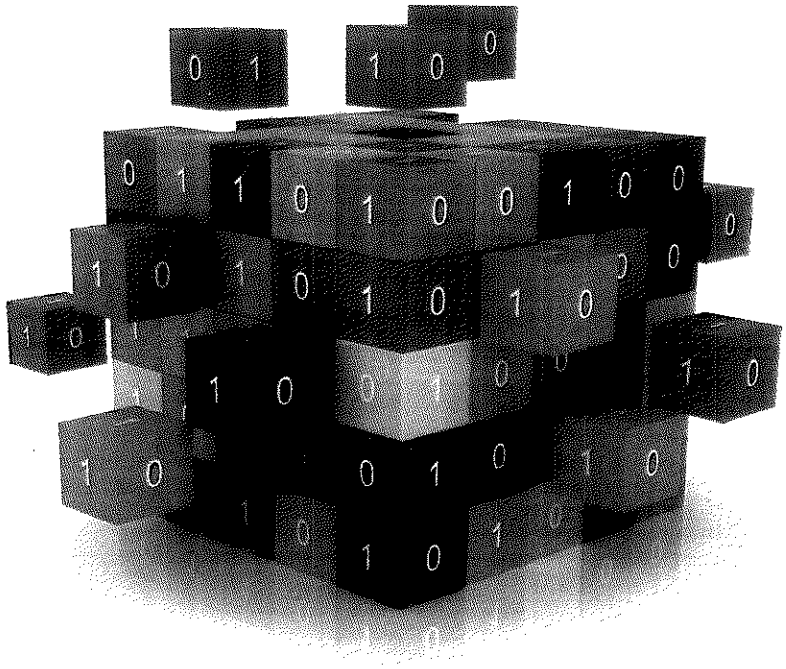
APOGEO

Introduzione alla teoria della computazione

Michael Sipser



MAGGIOLI EDITORE

Dal catalogo Apogeo Education
Informatica

Bolchini, Brandolese, Salice, Sciuto, *Reti logiche*, seconda edizione
Bruni, Corradini, Gervasi, *Programmazione in Java*, seconda edizione
Cabodi, Camurati, Pasini, Patti, Vendraminetto, *Dal problema al programma. Introduzione al problem-solving in linguaggio C*
Cabodi, Camurati, Pasini, Patri, Vendraminetto, *Ricorsione e problem-solving. Strategie algoritmiche in linguaggio C*
Collins, *Algoritmi e strutture dati in Java*
Coppola, Mizzaro, *Laboratorio di programmazione in Java*
Deitel, *C++ Fondamenti di programmazione*, seconda edizione
Deitel, *C++ Tecniche avanzate di programmazione*, seconda edizione
Della Mea, Di Gaspero, Scagnetto, *Programmazione web lato server*, seconda edizione
Di Noia, De Virgilio, Di Sciascio, Donini, *Semantic Web. Tra ontologie e Open Data*
Facchinetti, Larizza, Rubini, *Programmare in C. Concetti di base e tecniche avanzate*
Goodrich, Tamassia, Goldwasser, *Algoritmi e strutture dati in Java*
Hanly, Koffman, *Problem solving e programmazione in C*
Hennessy, Patterson, *Architettura degli elaboratori*
Horstmann, *Concetti di informatica e fondamenti di Java*, quinta edizione
Horstmann, Necaise, *Concetti di informatica e fondamenti di Python*
King, *Programmazione in C*
Laganà, Righi, Romani, *Informatica. Concetti e sperimentazioni*, seconda edizione
Lambert, *Programmazione in Python*
Lombardo, Valle, *Audio e multimedia*, quarta edizione
Malik, *Programmazione in C++*
Peterson, Davie, *Reti di calcolatori*, terza edizione
Schneider, Gersting, *Informatica*
Sklar, *Principi di Web Design*
Tarabella, *Musica informatica. Filosofia, storia e tecnologia della Computer Music*

APOGEO education

Introduzione alla teoria della computazione

Michael Sipser

Edizione italiana a cura di
Clelia De Felice, Luisa Gargano, Paolo D'Arco

MAGGIOLI EDITORE

Introduzione alla teoria della computazione
Autore: Michael Sipser

Titolo originale: Introduction to the Theory of Computation, 3rd edition

Traduzione e revisione: Clelia De Felice, Luisa Gargano, Paolo D'Arco
Impaginazione elettronica: Compomat srl

Maggioli Editore è un marchio di Maggioli S.p.A.
Azienda con sistema qualità certificato ISO 9001:2008

47822 Santarcangelo di Romagna (RN) • Via del Carpino, 8
Tel. 0541/628111 • Fax 0541/622595

www.maggiolieditore.it
e-mail: clienti.editore@maggioli.it

Finito di stampare nel mese di marzo 2016
nello stabilimento Maggioli S.p.A.
Santarcangelo di Romagna (RN)

INDICE

A Ina, Rachel ed Aaron

PREFAZIONE ALLA PRIMA EDIZIONE

Agli studenti

Benvenuti!

State per intraprendere lo studio di un argomento affascinante ed importante: la teoria della computazione. Esso comprende le proprietà matematiche fondamentali dell'hardware, del software e alcune loro applicazioni. Con lo studio di questo argomento, cerchiamo di determinare ciò che può e non può essere calcolato, in quanto tempo, con quanta memoria, e su quale tipo di modello computazionale. Questa materia ha evidenti connessioni con la pratica ingegneristica e, come in molte scienze, ha anche alcuni aspetti puramente filosofici.

So che molti di voi desiderano studiare questa materia, ma alcuni possono non essere qui per propria scelta. Forse volete conseguire una laurea in informatica o ingegneria, ed un corso di teoria è richiesto – Dio solo sa perché. Dopo tutto, non è vero che la teoria è arcana, noiosa, e peggio di tutto, irrilevante?

Per rendervi conto che la teoria non è né arcana né noiosa, ma invece del tutto comprensibile ed anche interessante, continuate a leggere. L'informatica teorica presenta molte idee affascinanti, ma ha anche molti piccoli dettagli che a volte la rendono faticosa. Imparare qualsiasi nuovo argomento è un lavoro duro, ma diventa più facile e più divertente se l'argomento viene correttamente presentato. Il mio obiettivo primario nello scrivere questo libro è quello di esporre gli aspetti veramente interessanti della teoria dei calcolatori, senza impantanarsi nella fatica dei dettagli. Naturalmente, l'unico modo per determinare se la teoria vi interessa è quello di provare a impararla.

La teoria è rilevante per la pratica. Fornisce strumenti concettuali che i professionisti utilizzano in Ingegneria Informatica. Lavorerete al progetto di un nuovo linguaggio di programmazione per un'applicazione specializzata? Quello che avrete imparato in questo corso sulle *grammatiche* vi ritornerà utile. Avrete un problema di ricerca su stringhe o di pattern matching? Ricorderete gli *automi finiti* e le *espressioni regolari*. Sarete di fronte ad un problema che sembra richiedere più tempo di calcolo di quanto vi potete permettere? Ripenserete a quello che avete imparato su *NP-completezza*.

Varie aree applicative, come i moderni protocolli crittografici, fanno affidamento sui principi teorici che imparerete qui. La teoria è anche importante per voi, perché vi mostra un lato nuovo, più semplice, e più elegante dei calcolatori, che normalmente consideriamo macchine complicate. I migliori progetti e le migliori applicazioni informatiche sono concepiti con l'eleganza in mente. Un corso teorico può aumentare il vostro senso estetico e vi aiuta a costruire sistemi più belli. Infine, la teoria è un bene per voi, perché il suo studio espande la vostra mente. La tecnologia informatica cambia rapidamente. Una preparazione tecnica specifica, oggi utile, può diventare obsoleta in pochi anni. Pensiamo invece alla capacità di pensare, di esprimersi in modo chiaro e preciso per la risoluzione dei problemi e di essere capaci di sapere quando un problema non è stato risolto. Queste abilità hanno un valore durevole. Studiare la teoria vi forma in questo ambito.

Considerazioni pratiche a parte, quasi tutti coloro che lavorano con i computer sono curiosi circa queste creazioni sorprendenti, le loro capacità, ed i loro limiti. Negli ultimi 30 anni, un nuovo ramo della matematica è nato e cresciuto per rispondere ad alcune domande fondamentali. Una domanda importante che rimane irrisolta è: se vi do un numero grande – per esempio, con 500 cifre – potete trovare i suoi fattori (i numeri che lo dividono) in un lasso di tempo ragionevole? Anche usando un supercomputer, nessuno attualmente sa come farlo in ogni caso *entro il ciclo di vita dell'universo!* Il problema della fattorizzazione è collegato ad alcuni codici segreti nella moderna crittografia. Trovate un metodo veloce per fattorizzare, e la fama sarà vostra!

Ai docenti

Questo libro è inteso per una laurea in informatica o come testo a livello introduttivo per dottorandi. Esso contiene un trattamento matematico dell'argomento, progettato intorno a teoremi e dimostrazioni. Ho fatto qualche sforzo per accogliere studenti con poca esperienza pregressa nelle dimostrazioni di teoremi, gli studenti più esperti saranno facilitati.

Il mio obiettivo primario nel presentare il materiale è stato quello di renderlo chiaro ed interessante. In tal modo, ho sottolineato l'intuizione ed il "quadro generale" dell'argomento rispetto ad alcuni dettagli di livello inferiore.

Ad esempio, anche se presento il metodo della dimostrazione per induzione nel Capitolo 0 insieme ad altri preliminari matematici, esso non gioca un ruolo importante nel seguito. In generale, non presento le solite prove per induzione della correttezza delle varie costruzioni relative ad automi. Se presentate in modo chiaro, queste costruzioni convincono e non hanno bisogno di ulteriori argomentazioni. Un'induzione può confondere piuttosto che illuminare perché l'induzione stessa è una tecnica piuttosto

sofisticata che molti trovano misteriosa. Elaborare l'ovvio con un'induzione rischia di insegnare agli studenti che una dimostrazione matematica è una manipolazione formale, invece di insegnare loro ciò che è e ciò che non è un'argomentazione convincente.

Un secondo esempio si ha nelle Parti Seconda e Terza, dove descrivo gli algoritmi in prosa invece che in pseudocodice. Non dedico molto tempo alla programmazione di macchine di Turing (o di qualsiasi altro modello formale). Gli studenti oggi vengono con una preparazione in programmazione e trovano la tesi di Church–Turing auto esplicativa. Quindi non presento simulazioni lunghe di un modello con un altro per stabilirne l'equivalenza.

Al di là del fornire l'intuizione in più e sopprimere alcuni dettagli, do quello che potrebbe essere chiamata una presentazione classica del materiale. La maggior parte dei teorici troveranno la scelta del materiale, la terminologia, e l'ordine di presentazione coerente con quella di altri libri di testo molto usati. Solo in alcuni punti, quando ho trovato la terminologia standard particolarmente oscura o confusa, ho introdotto una terminologia originale. Per esempio, presento il termine *riducibilità mediante funzioni* invece di *riducibilità molti-uno*.

La pratica attraverso la risoluzione dei problemi è essenziale per imparare qualsiasi argomento matematico. In questo libro, i problemi sono organizzati in due categorie principali chiamate *Esercizi* e *Problemi*. Gli esercizi ricapitolano definizioni e concetti. I problemi richiedono qualche abilità in più. I problemi contrassegnati con l'asterisco sono più difficili. Ho provato a fare di esercizi e problemi delle sfide interessanti.

La prima edizione

Introduzione alla teoria della computazione è apparso per la prima volta come un'edizione preliminare in formato economico. La prima edizione differisce dall'edizione preliminare in vari aspetti sostanziali. I tre capitoli finali sono nuovi: il Capitolo 8 sulla complessità di spazio; il Capitolo 9 sull'intrattabilità dimostrabile; ed il Capitolo 10 su argomenti avanzati di teoria della complessità. Il Capitolo 6 è stato ampliato per includere diversi argomenti avanzati della teoria della computabilità. Altri capitoli sono stati migliorati attraverso l'inserimento di ulteriori esempi ed esercizi.

I commenti degli istruttori e degli studenti che hanno utilizzato l'edizione preliminare sono stati utili per la rifinitura dei Capitoli 0-7. Naturalmente, gli errori riportati sono stati corretti in questa edizione.

I Capitoli 6 e 10 danno una visione generale di alcuni argomenti più avanzati nelle teorie della computabilità e della complessità. Essi non sono intesi come un'unità coesa nel modo in cui lo sono i restanti capitoli. Questi capitoli sono inclusi per consentire all'istruttore di selezionare argomenti opzionali che possono essere di interesse per lo studente motivato. Gli ar-

gomenti stessi spaziano ampiamente. Alcuni, come la *riducibilità di Turing* e l'*alternanza*, sono estensioni dirette di altri concetti nel libro. Altri, come le *teorie logiche decidibili* e la *crittografia*, sono brevi introduzioni a campi vasti.

Commenti per l'autore

Internet fornisce nuove possibilità di interazione tra autori e lettori. Ho ricevuto molte e-mail che offrono suggerimenti, lodi, e critiche, e che segnalano errori nell'edizione preliminare. Per favore continuate a scrivere! Cerco di rispondere ad ogni messaggio personalmente, appena il tempo lo permette. L'indirizzo e-mail per la corrispondenza relativa a questo libro è

sipserbook@math.mit.edu .

Ringraziamenti

Non avrei potuto scrivere questo libro senza l'aiuto di molti amici, colleghi e della mia famiglia.

Desidero ringraziare gli insegnanti che hanno contribuito a plasmare il mio punto di vista scientifico ed il mio stile educativo. Cinque di loro spiccano tra tutti. Al mio relatore di tesi, Manuel Blum, è dovuta una nota speciale per il suo modo unico di prendersi cura degli studenti e di ispirarli attraverso la chiarezza di pensiero e l'entusiasmo. Egli rappresenta un modello per me come per molti altri. Sono grato a Richard Karp per avermi fatto conoscere la teoria della complessità, a John Addison per avermi insegnato la logica e per aver assegnato quei meravigliosi compiti a casa, a Juris Hartmanis per avermi fatto conoscere la teoria della computazione, ed a mio padre per avermi fatto familiarizzare con la matematica, l'informatica e l'arte di insegnare.

Questo libro nasce dalle note da un corso che ho insegnato al MIT negli ultimi 15 anni. Alcuni studenti delle mie classi hanno preso le note delle mie lezioni. Spero che mi perdoneranno per non elencarli tutti. I miei assistenti alle lezioni nel corso degli anni – Avrim Blum, Thang Bui, Benny Chor, Andrew Chou, Stavros Cosmadakis, Aditi Dhagat, Wayne Goddard, Parry Husbands, Dina Kravets, Jakov Kučan, Brian O'Neill, Ioana Popescu, and Alex Russell – mi hanno aiutato a modificare ed ampliare queste note e hanno fornito alcuni dei problemi.

Circa tre anni fa, Tom Leighton mi ha convinto a scrivere un libro di testo sulla teoria della computazione. Avevo pensato di farlo per qualche tempo, ma ci è voluta la persuasione di Tom per trasformare la teoria in pratica. Apprezzo i suoi generosi consigli sulla scrittura del libro e su molte altre cose.

Voglio ringraziare Eric Bach, Peter Beebee, Cris Calude, Marek Chrobak, Anna Chefter, Guang-Ien Cheng, Elias Dahlhaus, Michael Fischer, Steve Fisk, Lance Fortnow, Henry J. Friedman, Jack Fu, Seymour Ginsburg, Oded Goldreich, Brian Grossman, David Harel, Micha Hofri, Dung T. Huynh, Neil Jones, H. Chad Lane, Kevin Lin, Michael Loui, Silvio Micali, Tadao Murata, Christos Papadimitriou, Vaughan Pratt, Daniel Rosenband, Brian Scassellati, Ashish Sharma, Nir Shavit, Alexander Shen, Ilya Shlyakhter, Matt Stallmann, Perry Susskind, Y. C. Tay, Joseph Traub, Osamu Watanabe, Peter Widmayer, David Williamson, Derick Wood e Charles Yang per i loro commenti, suggerimenti ed assistenza durante la scrittura.

Le seguenti persone hanno fornito ulteriori commenti che hanno migliorato questo libro: Isam M. Abdelhameed, Eric Allender, Shay Artzi, Michelle Atherton, Rolfe Blodgett, Al Briggs, Brian E. Brooks, Jonathan Buss, Jin Yi Cai, Steve Chapel, David Chow, Michael Ehrlich, Yaakov Eisenberg, Farzan Fallah, Shaun Flisakowski, Hjalmtyr Hafsteinsson, C. R. Hale, Maurice Herlihy, Vegard Holmedahl, Sandy Irani, Kevin Jiang, Rhys Price Jones, James M. Jowdy, David M. Martin Jr., Manrique Mata-Montero, Ryota Matsuura, Thomas Minka, Farooq Mohammed, Tadao Murata, Jason Murray, Hideo Nagahashi, Kazuo Ohta, Constantine Papageorgiou, Joseph Raj, Rick Regan, Rhonda A. Reumann, Michael Rintzler, Arnold L. Rosenberg, Larry Roske, Max Rozenoer, Walter L. Ruzzo, Sanatan Sahgal, Leonard Schulman, Steve Seiden, Joel Seiferas, Ambuj Singh, David J. Stucki, Jayram S. Thathachar, H. Venkateswaran, Tom Whaley, Christopher Van Wyk, Kyle Young e Kyoung Hwan Yun.

Robert Sloan ha utilizzato una prima versione del manoscritto di questo libro in una classe dove ha insegnato e mi ha fornito inestimabili commenti ed idee derivanti dalla sua esperienza. Mark Herschberg, Kazuo Ohta, e Latanya Sweeney hanno letto parti del manoscritto e suggerito ampi miglioramenti. Shafi Goldwasser mi ha aiutato con il materiale del Capitolo 10.

Ho ricevuto supporto tecnico da William Baxter di Superscript, che ha scritto il pacchetto LaTeX di macro che realizza il design del libro, e da Larry Nolan presso il dipartimento di matematica del MIT.

È stato un piacere lavorare con le persone della PWS Publishing per la creazione del prodotto finale. Cito Michael Sugarman, David Dietz, Elise Kaiser, Monique Calello, Susan Garland e Tanja Brull perché ho avuto più contatti con loro, ma so che molti altri sono stati coinvolti. Grazie a Jerry Moore quale curatore editoriale, a Diane Levy per il design della copertina e a Caterina Hawkes per il design dell'interno.

Sono grato alla National Science Foundation per il sostegno fornito con il fondo di ricerca CCR-9.503.322.

Mio padre, Kenneth Sipser, e mia sorella, Laura Sipser, hanno convertito diagrammi del libro in forma elettronica. L'altra mia sorella, Karen Fisch, ci ha salvato in varie emergenze con il computer e mia madre, Justine Sipser,

ha aiutato con consigli materni. Li ringrazio per il contributo fornitomi in circostanze difficili, comprese scadenze folli e software recalcitrante.

Infine, il mio amore va a mia moglie, Ina, e a mia figlia, Rachel. Grazie per aver sopportato tutto questo.

Cambridge, Massachusetts ***Michael Sipser***
Ottobre, 1996

PREFAZIONE ALLA SECONDA EDIZIONE

A giudicare dalle comunicazioni e-mail che ho ricevuto da tanti di voi, il più grande deficit della prima edizione è che non fornisce esempi di soluzioni per nessuno dei problemi. Quindi li ho aggiunti qui. Ogni capitolo contiene ora una nuova sezione *Soluzioni selezionate* che dà risposte ad un campione rappresentativo di esercizi e problemi del capitolo. Per compensare la perdita dei problemi risolti come interessanti sfide nei compiti a casa, ho aggiunto una serie di nuovi problemi. Il manuale per istruttori contiene altre soluzioni e può essere trovato sul sito web `www.cengage.com/international`.

Vari lettori avrebbero voluto una maggiore copertura di alcuni temi "standard", in particolare il Teorema di Myhill-Nerode ed il Teorema di Rice. Ho parzialmente accontentato questi lettori attraverso lo sviluppo di tali argomenti in problemi risolti. Non ho incluso il Teorema di Myhill-Nerode nel corpo principale del testo perché ritengo che questo corso dovrebbe fornire solo un'introduzione agli automi finiti e non un'indagine approfondita. A mio avviso, il ruolo degli automi a stati finiti qui è quello di far esplorare agli studenti un semplice modello formale di computazione come un preludio a modelli più potenti, e per fornire esempi utili per argomenti successivi. Naturalmente, alcune persone preferiscono un trattamento più approfondito, mentre altri ritengono che avrei dovuto omettere tutti i riferimenti agli (o almeno la dipendenza da) automi finiti. Non ho incluso il Teorema di Rice nel corpo principale del testo perché, pur essendo uno "strumento utile" per dimostrare l'indecidibilità, alcuni studenti potrebbero utilizzarlo meccanicamente senza realmente capire cosa stia succedendo. Utilizzare invece riduzioni per dimostrazioni di indecidibilità, dà una preparazione più utile per affrontare le riduzioni che si trovano nella teoria della complessità.

Sono in debito con i miei assistenti – Ilya Baran, Sergi Elizalde, Rui Fan, Jonathan Feldman, Venkatesan Guruswami, Prahladh Harsha, Christos Kapoutsis, Julia Khodor, Adam Klivans, Kevin Matulef, Ioana Popescu, April Rasala, Sofya Raskhodnikova e Iuliu Vasilescu – che mi hanno aiutato a scrivere alcuni dei nuovi problemi e soluzioni. Ching Law, Edmond Kayi Lee e Zulfikar Ramzan hanno contribuito alle soluzioni. Ringrazio Victor Shoup per aver suggerito un modo semplice per eliminare il divario nell'analisi dell'algoritmo di primalità probabilistico della prima edizione.

Apprezzo gli sforzi delle persone del Course Technology nello spronare me e le altre parti durante questo progetto, soprattutto Alyssa Pratt e Aimee Poirier. Molte grazie a Gerald Eisman, Weizhen Mao, Rupak Majumdar, Chris Umans, e Christopher Wilson per le loro recensioni. Sono in debito con Jerry Moore per il suo eccellente lavoro di editore ed a Laura Segel di ByteGraphics (`lauras@bytegraphics.com`) per le sue belle figure.

Il volume di e-mail che ho ricevuto è stato superiore a quanto mi aspettassi. Sentire da tanti di voi e da tanti luoghi è stato assolutamente delizioso, ho cercato di rispondere a tutti – le mie scuse a coloro che ho mancato. Ho elencato qui le persone che hanno fatto proposte che hanno avuto un impatto specifico su questa edizione, ma ringrazio tutti per i loro messaggi:

Luca Aceto, Arash Afkanpour, Rostom Aghanian, Eric Allender, Karun Bakshi, Brad Ballinger, Ray Bartkus, Louis Barton, Arnold Beckmann, Mihir Bellare, Kevin Trent Bergeson, Matthew Berman, Rajesh Bhatt, Somenath Biswas, Lenore Blum, Mauro A. Bonatti, Paul Bondin, Nicholas Bone, Ian Bratt, Gene Browder, Doug Burke, Sam Buss, Vladimir Bychkovsky, Bruce Carneal, Soma Chaudhuri, Rong-Jaye Chen, Samir Chopra, Benny Chor, John Clausen, Allison Coates, Anne Condon, Jeffrey Considine, John J. Crashell, Claude Crepeau, Shaun Cutts, Susheel M. Daswani, Geoff Davis, Scott Dexter, Peter Drake, Jeff Edmonds, Yaakov Eisenberg, Kurtcebe Eroglu, Georg Essl, Alexander T. Fader, Farzan Fallah, Faith Fich, Joseph E. Fitzgerald, Perry Fizzano, David Ford, Jeannie Fromer, Kevin Fu, Atsushi Fujioka, Michel Galley, K. Ganesan, Simson Garfinkel, Travis Gebhardt, Peymann Gohari, Ganesh Gopalakrishnan, Steven Greenberg, Larry Griffith, Jerry Grossman, Rudolf de Haan, Michael Halper, Nick Harvey, Mack Hendricks, Laurie Hiyakumoto, Steve Hockema, Michael Hoehle, Shahadat Hossain, Dave Isecke, Ghaith Issa, Raj D. Iyer, Christian Jacobi, Thomas Janzen, Mike D. Jones, Max Kanovitch, Aaron Kaufman, Roger Khazan, Sarfraz Khurshid, Kevin Killourhy, Seungjoo Kim, Victor Kuncak, Kanata Kuroda, Thomas Lasko, Suk Y. Lee, Edward D. Legenski, Li-Wei Lehman, Kong Lei, Zsolt Lengvarszky, Jeffrey Levetin, Baekjun Lim, Karen Livescu, Stephen Louie, TzerHung Low, Wolfgang Maass, Arash Madani, Michael Manapat, Wojciech Marchewka, David M. Martin Jr., Anders Martinson, Lyle McGeoch, Alberto Medina, Kurt Mehlhorn, Nihar Mehta, Albert R. Meyer, Thomas Minka, Mariya Minkova, Daichi Mizuguchi, G. Allen Morris III, Damon Mosk-Aoyama, Xiaolong Mou, Paul Muir, German Muller, Donald Nelson, Gabriel Nivasch, Mary Obelnicki, Kazuo Ohta, Thomas M. Oleson, Jr., Curtis Oliver, Owen Ozier, Rene Peralta, Alexander Perlis, Holger Petersen, Detlef Plump, Robert Prince, David Pritchard, Bina Reed, Nicholas Riley, Ronald Rivest, Robert Robinson, Christi Rockwell, Phil Rogaway, Max Rozenoer, John Rupf, Teodor Rus, Larry Ruzzo, Brian Sanders, Cem Say, Kim Schioett, Joel Seiferas, Joao Carlos Setubal, Geoff Lee Seyon, Mark Skandera, Bob Sloan, Geoff

Smith, Marc L. Smith, Stephen Smith, Alex C. Snoeren, Guy St-Denis, Larry Stockmeyer, Radu Stoleru, David Stucki, Hisham M. Sueyllam, Kenneth Tam, Elizabeth Thompson, Michel Toulouse, Eric Tria, Chittaranjan Tripathy, Dan Trubow, Hiroki Ueda, Giora Unger, Kurt L. Van Etten, Jesir Vargas, Bienvenido Velez-Rivera, Kobus Vos, Alex Vrenios, Sven Waibel, Marc Waldman, Tom Whaley, Anthony Widjaja, Sean Williams, Joseph N. Wilson, Chris Van Wyk, Guangming Xing, Vee Voon Yee, Cheng Yongxi, Neal Young, Timothy Yuen, Kyle Yung, Jinghua Zhang, Lilla Zollei.

Ringrazio Suzanne Balik, Matthew Kane, Kurt L. Van Etten, Nancy Lynch, Gregory Roberts e Cem Say per aver indicato errori nella prima ristampa.

Maggiormente, ringrazio la mia famiglia – Ina, Rachel, e Aaron – per la loro pazienza, la comprensione e l'amore quando sono stato seduto per interminabili ore davanti allo schermo del mio computer.

Cambridge, Massachusetts ***Michael Sipser***
Dicembre, 2004

PREFAZIONE ALLA TERZA EDIZIONE

La terza edizione contiene una nuova sezione sui linguaggi context-free deterministici. Ho scelto questo argomento per diversi motivi. Prima di tutto, si riempie una lacuna evidente nella mia precedente trattazione della teoria degli automi e dei linguaggi. Le edizioni precedenti introducevano automi a stati finiti e macchine di Turing nelle varianti deterministiche e non deterministiche, ma coprivano solo la variante non deterministica degli automi a pila. L'aggiunta di una discussione sugli automi a pila deterministici, fornisce un pezzo mancante del puzzle.

In secondo luogo, la teoria delle grammatiche context-free deterministiche è la base per le grammatiche LR(k), un'applicazione importante e non banale della teoria degli automi nei linguaggi di programmazione e nella progettazione di compilatori. Quest'applicazione mette insieme diversi concetti chiave, tra cui l'equivalenza di automi a stati finiti deterministici e non deterministici, e le conversioni tra grammatiche context-free ed automi a pila, per produrre un metodo bello ed efficiente per il parsing. Qui abbiamo un'interazione concreta tra teoria e pratica.

Infine, questo argomento non sembra trattato abbastanza nei libri di teoria esistenti, considerata la sua importanza come applicazione della teoria degli automi. Ho studiato le grammatiche LR(k) anni fa, ma senza capire completamente come lavorano, e senza vedere come si inseriscono bene nella teoria dei linguaggi context-free deterministici. Il mio obiettivo nello scrivere questo capitolo è quello di dare un'introduzione intuitiva ma rigorosa a quest'area rivolta sia a teorici che applicativi, e contribuire in tal modo ad un suo più ampio apprezzamento. Una nota di cautela, però: parte del materiale in questa sezione è piuttosto impegnativo, quindi un docente di un primo corso di base di teoria può preferire utilizzarlo solo come materiale supplementare. I capitoli successivi non dipendono da questo materiale.

Molte persone hanno contribuito direttamente o indirettamente allo sviluppo di questa edizione. Sono in debito ai revisori Christos Kapoutsis e Cem Say che hanno letto una bozza della nuova sezione e hanno fornito commenti preziosi. Diverse persone alla Cengage Learning hanno fornito assistenza durante la produzione, in particolare Alyssa Pratt e Jennifer Feltri-George. Suzanne Huizenga ha revisionato il testo e Laura Segel di ByteGraphics ha creato le nuove figure e modificato alcune delle figure più vecchie.

Desidero ringraziare i miei assistenti alle lezioni al MIT, Victor Chen, Andy Drucker, Michael Forbes, Elena Grigorescu, Brendan Juba, Christos Kapoutsis, Jon Kelner, Swastik Kopparty, Kevin Matulef, Amanda Redlich, Zack Remscrim, Ben Rossman, Shubhangi Saraf e Oren Weimann. Ognuno di loro mi ha aiutato, discutendo di nuovi problemi e delle rispettive soluzioni ed aiutando a comprendere fino a che punto i nostri studenti avevano capito i contenuti del corso. Mi ha fatto molto piacere lavorare con giovani così colmi di entusiasmo e di talento.

È stato gratificante ricevere email da tutto il mondo. Grazie a tutti per i vostri suggerimenti, domande ed idee. Di seguito elenco quelle persone i cui commenti hanno avuto un impatto su questa edizione:

Djihed Afifi, Steve Aldrich, Eirik Bakke, Suzanne Balik, Victor Bandur, Paul Beame, Elazar Birnbaum, Goutam Biswas, Rob Bittner, Marina Blanton, Rodney Bliss, Promita Chakraborty, Lewis Collier, Jonathan Deber, Simon Dexter, Matt Diephouse, Peter Dillinger, Peter Drake, Zhidian Du, Peter Fejer, Margaret Fleck, Atsushi Fujioka, Valerio Genovese, Evangelos Georgiadis, Joshua Grochow, Jerry Grossman, Andreas Guelzow, Hjalmtyr Hafsteinsson, Arthur Hall III, Cihat Imamoglu, Chinawat Isradisaikul, Kayla Jacobs, Flemming Jensen, Barbara Kaiser, Matthew Kane, Christos Kapoutsis, Ali Durlov Khan, Edwin Sze Lun Khoo, Yongwook Kim, Akash Kumar, Eleazar Leal, Zsolt Lengvarszky, Cheng-Chung Li, Xiangdong Liang, Vladimir Lifschitz, Ryan Lortie, Jonathan Low, Nancy Lynch, Alexis Maciel, Kevin Matulef, Nelson Max, Hans-Rudolf Metz, Mladen Mikŝa, Sara Miner More, Rajagopal Nagarajan, Marvin Nakayama, Jonas Nyrup, Gregory Roberts, Ryan Romero, Santhosh Samarthyam, Cem Say, Joel Seiferas, John Sieg, Marc Smith, John Steinberger, Nuri Taşdemir, Tamir Tassa, Mark Testa, Jesse Tjang, John Trammell, Hiroki Ueda, Jeroen Vaelen, Kurt L. Van Etten, Guillermo Vàzquez, Phanisekhar Botlaguduru Venkata, Benjamin Bing-Yi Wang, Lutz Warnke, David Warren, Thomas Watson, Joseph Wilson, David Wittenberg, Brian Wongchaowart, Kishan Yerubandi, Dai Yi.

Soprattutto, ringrazio la mia famiglia – mia moglie, Ina, ed i nostri figli, Rachel ed Aaron. Il tempo è finito e fugace. Il vostro amore è tutto.

Cambridge, Massachusetts ***Michael Sipser***
Aprile, 2012

PREFAZIONE ALL'EDIZIONE ITALIANA

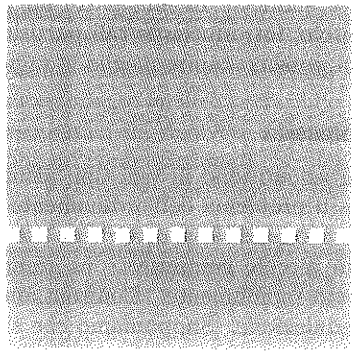
Il libro di Michael Sipser abbraccia argomenti fondazionali nell'ambito della teoria della computabilità e della complessità, partendo dalla teoria degli automi e dei linguaggi formali per giungere a trattare argomenti avanzati in importanti ambiti pratici, quali la crittografia. Alcuni di questi argomenti sono presenti in tutti i corsi di Computer Science italiani ed esteri. Il libro è stato pubblicato per la prima volta nel 1997; le successive edizioni hanno dimostrato la sua freschezza, attualità e importanza.

Introduction to the Theory of Computation è un testo adottato in numerosi corsi di laurea a vari livelli sia di Informatica che di Matematica e di Ingegneria. In Italia è utilizzato per l'insegnamento della Teoria della Computazione sia in corsi di laurea triennali che magistrali. Il motivo della scelta di tale testo, tra i tanti che affrontano questi argomenti, risiede probabilmente, e sicuramente per quanto ci riguarda, nel fatto che l'autore, consapevole delle difficoltà degli studenti nell'apprendimento della materia, fa precedere alla trattazione rigorosa degli argomenti una loro presentazione intuitiva ed illustrata da esempi illuminanti. La parte più formale è scritta con uno stile sintetico, escludendo volutamente i dettagli squisitamente tecnici. Le dimostrazioni sono tratteggiate negli aspetti più significativi, tralasciando le conclusioni a cui lo studente può giungere autonomamente o lasciandogli il compito di intuire il ragionamento formale che consente di giungere ad alcune affermazioni. Il testo è ricco di esercizi di vari livelli di difficoltà, alcuni dei quali anche con soluzione, che permettono agli studenti di confrontarsi con la materia e verificare il proprio grado di apprendimento. La trattazione degli argomenti, che in gran parte si trovano a cavallo tra la Matematica e l'Informatica, è chiaramente rivolta agli studenti di Informatica, con il continuo riferimento a esempi di interesse informatico, in cui i concetti presentati trovano applicazione.

Ci siamo impegnati nella traduzione italiana ritenendo che essa faciliti ulteriormente l'apprendimento degli argomenti trattati. Alcuni termini sono stati volutamente non tradotti poiché il termine anglosassone è ormai di uso comune in ambito informatico.

Clelia De Felice
Luisa Gargano
Paolo D'Arco

0



INTRODUZIONE

Iniziamo con una panoramica di quelle aree della teoria della computazione che presenteremo in questo corso. Nel seguito, avrete la possibilità di imparare e/o rivedere alcuni concetti matematici che vi saranno necessari nel prosieguo.

0.1
AUTOMI, COMPUTABILITÀ E COMPLESSITÀ

Questo libro si concentra su tre aree tradizionalmente centrali alla teoria della computazione: automi, computabilità e complessità. Tutte queste aree sono collegate dalla seguente domanda:

Quali sono le capacità e le limitazioni intrinseche dei calcolatori?

Questa domanda risale al 1930, quando gli studiosi di logica matematica iniziarono ad esplorare il significato della computazione. I successivi progressi tecnologici hanno aumentato notevolmente la nostra capacità di calcolare e hanno portato tale domanda da un ambito puramente teorico a quello delle applicazioni pratiche.

In ciascuna delle tre aree – automi, computabilità e complessità – la questione è stata interpretata in modo differente e le risposte variano in

base a tale interpretazione. Dopo questo capitolo introduttivo, esploreremo ognuna delle tre aree suddette in una parte ad essa dedicata di questo libro. In questo capitolo, introduciamo tali parti in ordine inverso, perché partendo dalla fine si possono capire meglio le ragioni dell'inizio.

Teoria della complessità

I problemi di computazione sono di vari tipi; alcuni sono facili, altri sono difficili. Ad esempio, il problema dell'ordinamento rientra tra quelli facili. Supponiamo che sia necessario disporre di un elenco di numeri in ordine crescente. Anche un piccolo computer è in grado di ordinare un milione di numeri piuttosto rapidamente. Confrontiamo questo problema con un problema di schedulazione. Supponiamo che sia necessario trovare un calendario delle lezioni per l'intera università che soddisfi alcuni vincoli, come ad esempio che non esistano due classi differenti allocate contemporaneamente nella stessa aula. Il problema di schedulazione sembra essere molto più difficile del problema di ordinamento. Se si hanno solo mille classi, trovare l'allocazione migliore può richiedere secoli, persino con un supercomputer.

Cosa rende alcuni problemi computazionalmente difficili ed altri facili?

Questa è la questione principale della teoria della complessità. Sorprendentemente, non conosciamo risposta a tale quesito, anche se è stato intensamente studiato per oltre 40 anni. In seguito, esploreremo questa affascinante domanda insieme ad alcune delle sue variazioni. In uno dei più importanti risultati della teoria della complessità finora ottenuti, i ricercatori hanno scoperto un elegante sistema per classificare i problemi in base alla difficoltà computazionale relativa. È una classificazione analoga a quella utilizzata per classificare gli elementi nella tavola periodica in base alle loro proprietà chimiche.

Utilizzando tale schema, possiamo dare un metodo che fornisce una prova a sostegno che alcuni problemi sono computazionalmente difficili, anche se siamo incapaci di dimostrarlo.

Vi sono varie opzioni quando si affronta un problema che appare computazionalmente difficile. In primo luogo, attraverso la comprensione della natura della difficoltà, può essere possibile modificare il problema in modo da renderlo più facilmente risolvibile. In secondo luogo, può essere possibile accontentarsi di una soluzione non esatta del problema. In taluni casi trovare soluzioni che sono solo approssimativamente esatte risulta relativamente facile. In terzo luogo, alcuni problemi sono difficili solo nel caso peggiore, ma facili nella maggior parte dei casi. A seconda dell'applicazione, si può essere soddisfatti con una procedura che risulta lenta in qualche caso, ma è di norma veloce. Infine, si possono prendere in considerazione

tipi alternativi di calcolo, come ad esempio il calcolo randomizzato, che può velocizzare alcuni compiti.

Un'area applicativa che è stata direttamente influenzata dalla teoria della complessità è l'antico campo della crittografia. Nella maggior parte dei casi, un problema computazionalmente facile è preferibile ad uno difficile perché quelli facili sono più economici da risolvere. La crittografia è particolare perché necessita di problemi computazionali che siano difficili da risolvere, piuttosto che semplici. I codici segreti devono essere difficili da violare senza chiave segreta o password. La teoria della complessità ha indirizzato i crittografi verso i problemi computazionalmente difficili, utilizzando i quali essi hanno progettato nuove codifiche rivoluzionarie.

Teoria della computabilità

Durante la prima metà del ventesimo secolo, alcuni matematici tra cui Kurt Gödel, Alan Turing ed Alonzo Church scoprirono l'esistenza di problemi che non possono essere risolti con i computer. Un esempio di questo fenomeno è il problema di stabilire se un enunciato matematico sia vero o falso. Questo compito è il pane quotidiano dei matematici. Sembra che esso si presti naturalmente ad una soluzione mediante computer, perché si trova decisamente all'interno della matematica. Tuttavia non esiste alcun algoritmo in grado di eseguire tale compito.

Una delle conseguenze di questo profondo risultato fu lo sviluppo di idee in materia di modelli teorici di calcolatore che alla fine avrebbero favorito la costruzione degli attuali computer. Le teorie della computabilità e della complessità sono strettamente correlate. Nella teoria della complessità, l'obiettivo è quello di classificare i problemi come facili o difficili, mentre nella teoria della computabilità la classificazione dei problemi è tra quelli che sono risolvibili e quelli che invece non lo sono. La teoria della computabilità introduce molti dei concetti utilizzati nella teoria della complessità.

Teoria degli automi

La teoria degli automi si occupa delle definizioni e delle proprietà dei modelli di calcolo matematici. Questi modelli hanno un ruolo all'interno di varie aree applicative dell'informatica. Un modello, chiamato *automa finito*, è utilizzato nell'ambito del trattamento automatico dei testi, dei compilatori e della progettazione hardware. Un altro modello, chiamato *grammatica context-free*, è utilizzato nell'ambito dei linguaggi di programmazione e dell' intelligenza artificiale. La teoria degli automi è un eccellente punto di inizio per lo studio della teoria della computazione. Le teorie della computabilità e della complessità richiedono una precisa definizione di *computer*. La teoria degli automi permette di esercitarsi con le definizioni formali di

calcolo in quanto introduce concetti rilevanti in altre branche non teoriche dell'informatica.

0.2 TERMINOLOGIA E NOTAZIONE MATEMATICA

Come per ogni argomento matematico, cominciamo introducendo gli oggetti matematici di base, gli strumenti e la notazione che ci aspettiamo di utilizzare nel seguito.

Insiemi

Un ***insieme*** è un gruppo di oggetti rappresentati in maniera unitaria. Gli insiemi possono contenere qualsiasi tipo di oggetto, inclusi numeri, simboli, e perfino altri insiemi. Gli oggetti in un insieme sono chiamati i suoi ***elementi*** o membri. Gli insiemi possono essere descritti formalmente in vari modi. Un modo consiste nell'elencare gli elementi dell'insieme all'interno di parentesi graffe. Quindi l'insieme

$$S = \{7, 21, 57\}$$

contiene gli elementi 7, 21 e 57. I simboli $\in$ e $\notin$ denotano l'appartenenza e la non appartenenza ad un insieme. Scriviamo $7 \in \{7, 21, 57\}$ e $8 \notin \{7, 21, 57\}$. Dati due insiemi A e B, diciamo che A è un ***sottoinsieme*** di B, e scriviamo $A \subseteq B$, se ogni elemento di A è anche un elemento di B. Diciamo che A è un ***sottoinsieme proprio*** di B, e scriviamo $A \subsetneq B$, se A è un sottoinsieme di B e non è uguale a B.

L'ordine degli elementi nel descrivere un insieme non è importante, così come la ripetizione dei suoi elementi. Otteniamo lo stesso insieme S se scriviamo $\{57, 7, 7, 7, 21\}$. Se vogliamo tener conto del numero di occorrenze degli elementi, allora chiamiamo il gruppo ***multinsieme*** invece di insieme. Così $\{7\}$ e $\{7, 7\}$ sono differenti come multinsiemi ma identici come insiemi. Un ***insieme infinito*** contiene infiniti elementi. Non possiamo scrivere una lista di tutti gli elementi di un insieme infinito, pertanto a volte usiamo la notazione "..." per indicare che "la sequenza continua all'infinito". Così scriviamo l'insieme dei ***numeri naturali*** $\mathcal{N}$ come

$$\{1, 2, 3, \ldots\}.$$

Scriviamo l'insieme degli ***interi*** $\mathcal{Z}$ come

$$\{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

L'insieme contenente zero elementi è chiamato ***insieme vuoto*** e viene indicato con $\emptyset$. Un insieme contenente un unico elemento è a volte chiamato insieme ***singleton*** ed un insieme contenente due elementi è chiamato anche ***coppia non ordinata***. Quando vogliamo descrivere un insieme contenente elementi in base ad una qualche regola, scriviamo $\{n |$ regola per $n\}$. Per esempio $\{n |\ n = m^2$ per qualche $m \in \mathcal{N}\}$ indica l'insieme dei quadrati perfetti.

Dati due insiemi A and B, l'***unione*** di A e B, scritta $A \cup B$, è l'insieme ottenuto combinando tutti gli elementi di A e B in un unico insieme. L'***intersezione*** di A e B, scritta $A \cap B$, è l'insieme di tutti gli elementi che sono sia in A che in B.

Il ***complemento*** di A, scritto $\overline{A}$, è l'insieme di tutti gli elementi in esame che *non* sono in A. Come spesso accade in matematica, un'immagine aiuta a chiarire meglio un concetto. Per gli insiemi, usiamo un tipo di immagine chiamato ***diagramma di Venn***. Esso rappresenta gli insiemi come regioni delimitate da linee circolari. Sia l'insieme START-t l'insieme di tutte le parole inglesi che iniziano con la lettera "t". Ad esempio, nella figura il cerchio rappresenta l'insieme START-t. Vari elementi di questo insieme sono simbolicamente rappresentati da punti all'interno del cerchio.

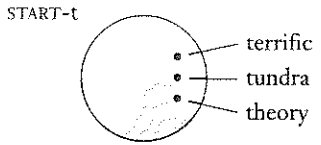


FIGURA 0.1
Diagramma di Venn per l'insieme delle parole inglesi che iniziano con la lettera "t"

Allo stesso modo, rappresentiamo l'insieme END-z delle parole inglesi che terminano con la lettera "z" nella figura seguente.

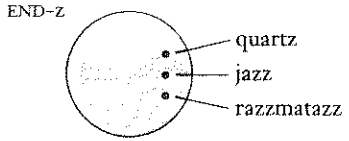


FIGURA 0.2
Diagramma di Venn per l'insieme di parole inglesi che terminano con la lettera "z"

Per rappresentare entrambi gli insiemi con uno stesso diagramma di Venn dobbiamo disegnarli in modo che si sovrappongano, indicando in tal modo

che essi condividono alcuni elementi, come illustrato nella figura seguente. Per esempio, la parola *topaz* appartiene ad entrambi gli insiemi. La figura contiene anche un cerchio per l'insieme START-j. Esso non si sovrappone al cerchio per le START-t, perché nessuna parola appartiene ad entrambi gli insiemi.

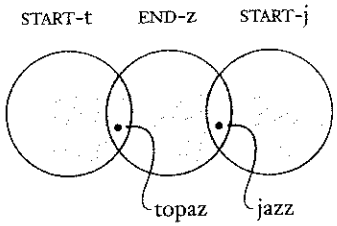


FIGURA 0.3
Cerchi sovrapposti indicano l'esistenza di elementi comuni

I due diagrammi di Venn seguenti raffigurano l'unione e intersezione degli insiemi A e B.

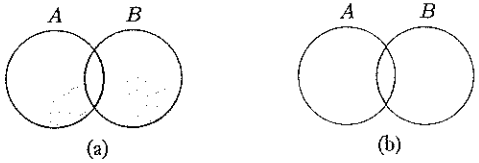


FIGURA 0.4
Diagrammi per (a) $A \cup B$ e (b) $A \cap B$

Sequenze e tuple

Una ***sequenza*** è una lista di oggetti in un qualche ordine. Di solito si indica una sequenza scrivendo la lista tra parentesi tonde. Ad esempio, la sequenza 7, 21, 57 è scritta come

$$(7, 21, 57).$$

In un insieme l'ordine non è importante, ma lo è in una sequenza. Quindi $(7, 21, 57)$ è una sequenza diversa da $(57, 7, 21)$. Analogamente, le ripetizioni hanno importanza in una sequenza, ma non in un insieme. Quindi $(7, 7, 21, 57)$ è diversa da entrambe le altre sequenze, mentre l'insieme $\{7, 21, 57\}$ è identico all'insieme $\{7, 7, 21, 57\}$.

Come per gli insiemi, le sequenze possono essere finite o infinite. Una sequenze finita è spesso chiamata ***tupla***. Una sequenza con k elementi è una ***k-tupla***. Quindi $(7, 21, 57)$ è una 3-tupla. Una 2-tupla è anche chiamata ***coppia ordinata***.

Insiemi e sequenze possono apparire come elementi di altri insiemi e sequenze. Per esempio, l'***insieme potenza*** di A è l'insieme di tutti i sottoinsiemi di A. Se A è l'insieme $\{0, 1\}$, l'insieme potenza di A è l'insieme $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$. L'insieme di tutte le coppie ordinate i cui elementi sono zeri ed uno è $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$.

Se A e B sono due insiemi, il ***prodotto cartesiano*** o ***prodotto vettoriale*** di A e B, denotato con $A \times B$, è l'insieme di tutte le coppie in cui il primo elemento è un elemento di A ed il secondo è un elemento di B.

ESEMPIO 0.5

Se $A = \{1, 2\}$ e $B = \{x, y, z\}$,

$$A \times B = \{(1, x), (1, y), (1, z), (2, x), (2, y), (2, z)\}.$$

Possiamo anche effettuare il prodotto Cartesiano di k insiemi, $A_1, A_2, \ldots, A_k$, scritto come $A_1 \times A_2 \times \cdots \times A_k$. Esso è l'insieme costituito da tutte le k-tuple $(a_1, a_2, \ldots, a_k)$ dove $a_i \in A_i$.

ESEMPIO 0.6

Se A e B sono gli insiemi dell'Esempio 0.5,

$$A \times B \times A = \{(1, x, 1), (1, x, 2), (1, y, 1), (1, y, 2), (1, z, 1), (1, z, 2), (2, x, 1), (2, x, 2), (2, y, 1), (2, y, 2), (2, z, 1), (2, z, 2)\}.$$

Se abbiamo il prodotto cartesiano di un insieme con se stesso, usiamo la notazione

$$\overbrace{A \times A \times \cdots \times A}^{k} = A^k.$$

ESEMPIO 0.7

L'insieme $\mathcal{N}^2$ equivale a $\mathcal{N} \times \mathcal{N}$. Si compone di tutte le coppie di numeri naturali. Possiamo anche scriverlo come $\{(i, j) \mid i, j \geq 1\}$.

Funzioni e relazioni

Le funzioni sono fondamentali in matematica. Una ***funzione*** è un oggetto che stabilisce una relazione input–output. Una funzione prende un input e produce un output. In ogni funzione, uno stesso input produce sempre

lo stesso output. Se f è una funzione il cui valore di output è b quando il valore di input è a, scriviamo

$$f(a) = b.$$

Una funzione è anche chiamata ***mappa*** e, se $f(a) = b$, diciamo che f mappa a in b.

Ad esempio, la funzione valore assoluto *abs* prende un numero x in input e restituisce x se x è positivo e $-x$ se x è negativo. Quindi $abs(2) = abs(-2) = 2$. L'addizione è un altro esempio di una funzione, scritta *add*. L'input della funzione *add* è una coppia di numeri e l'output è la somma di tali numeri.

L'insieme dei possibili input per la funzione è detto ***dominio***. L'output di una funzione forma un insieme chiamato ***codominio*** o ***range***. La notazione per indicare che f è una funzione con dominio D e range R è

$$f\colon D \longrightarrow R.$$

Nel caso della funzione *abs*, se lavoriamo con numeri interi allora dominio e codominio sono in $\mathcal{Z}$, quindi scriviamo $abs\colon \mathcal{Z} \longrightarrow \mathcal{Z}$. Nel caso della funzione di addizione per gli interi, il dominio è l'insieme delle coppie di interi $\mathcal{Z} \times \mathcal{Z}$ ed il range è $\mathcal{Z}$, quindi scriviamo $add\colon \mathcal{Z} \times \mathcal{Z} \longrightarrow \mathcal{Z}$. Si noti che una funzione può non utilizzare tutti gli elementi del codominio specificato. La funzione *abs* non prende mai il valore -1 anche se $-1 \in \mathcal{Z}$. Una funzione che fa uso di tutti gli elementi del codominio si dice ***suriettiva***.

Vi sono vari modi per descrivere una funzione. Una possibilità consiste nel fornire una procedura per calcolare un output da uno specifico input. Un altro modo è mediante una tabella che elenca tutti i possibili input e per ognuno fornisce l'output corrispondente.

ESEMPIO 0.8

Consideriamo la funzione $f\colon \{0,1,2,3,4\} \longrightarrow \{0,1,2,3,4\}$.

n	$f(n)$
0	1
1	2
2	3
3	4
4	0

Questa funzione aggiunge 1 all'input e quindi restituisce in output il risultato modulo 5. Un numero modulo m è il resto della divisione del numero per m. Per esempio, la lancetta dei minuti di un orologio conta modulo 60. Quando operiamo con l'aritmetica modulare definiamo $\mathcal{Z}_m = \{0, 1, 2, \ldots, m-1\}$. Con questa notazione, la funzione f introdotta prima si può scrivere come $f\colon \mathcal{Z}_5 \longrightarrow \mathcal{Z}_5$.

ESEMPIO 0.9

A volte, se il dominio della funzione è il prodotto cartesiano di due insiemi, si utilizza una tabella bidimensionale. Consideriamo la funzione $g\colon \mathcal{Z}_4 \times \mathcal{Z}_4 \longrightarrow \mathcal{Z}_4$. L'elemento corrispondente alla riga i ed alla colonna j della tabella è il valore $g(i, j)$.

g	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

La funzione g è la somma modulo 4.

Quando il dominio di una funzione f è $A_1 \times \cdots \times A_k$ per qualche $A_1, \ldots, A_k$, l'input di f è una k-tupla $(a_1, a_2, \ldots, a_k)$ e chiamiamo ogni a_i un ***argomento*** di f. Una funzione con k argomenti si dice ***funzione k-aria***, e k è chiamata ***arietà*** della funzione. Se k è 1, f ha un solo argomento ed è chiamata ***funzione unaria***. Se k è 2, f è una ***funzione binaria***. Alcune funzioni binarie sono scritte mediante ***notazione infissa***, con il simbolo della funzione tra i due argomenti, invece che in ***notazione prefissa***, con il simbolo che precede gli argomenti. Per esempio la funzione di addizione *add* è normalmente scritta mediante notazione infissa con il simbolo + tra i due argomenti, come in $a + b$, invece della notazione prefissa $add(a, b)$.

Un ***predicato*** o ***proprietà*** è una funzione il cui range è {VERO, FALSO}. Per esempio, assumendo che *pari* sia una proprietà che ha valore VERO se il suo input è un numero pari e FALSO se il suo input è un numero dispari. Quindi $pari(4) =$ VERO e $pari(5) =$ FALSO.

Una proprietà il cui dominio è un insieme di k-tuple $A \times \cdots \times A$ è detta ***relazione***, ***relazione k-aria***, o ***relazione k-aria su A***. Un caso comune è una relazione 2-aria, detta relazione binaria. Quando si scrive un'espressione che contiene una relazione binaria, utilizziamo abitualmente la notazione infissa. Ad esempio, "minore di" è una relazione di solito denotata con il simbolo di operazione infissa $<$. "Uguale", scritto con il simbolo $=$ è un'altra relazione familiare. Se R è una relazione binaria, l'affermazione aRb significa che $aRb =$ VERO. Allo stesso modo, se R è una relazione k-aria, la dichiarazione $R(a_1, \ldots, a_k)$ significa che $R(a_1, \ldots, a_k) =$ VERO.

ESEMPIO 0.10

In un gioco per bambini chiamato Sasso–Carta–Forbici, due giocatori scelgono contemporaneamente un elemento dell'insieme {SASSO, CARTA, FORBICI} e indicano le loro selezioni con segnali manuali. Se le due selezioni sono uguali, il gioco ricomincia. Se le selezioni sono diverse, un giocatore vince in base alla relazione *batte*.

batte	FORBICI	CARTA	SASSO
FORBICI	FALSO	VERO	FALSO
CARTA	FALSO	FALSO	VERO
SASSO	VERO	FALSO	FALSO

Da questa tabella si determina che FORBICI *batte* CARTA è VERO e che CARTA *batte* FORBICI è FALSO.

A volte risulta più conveniente descrivere i predicati mediante insiemi, invece di funzioni. Il predicato $P\colon D \longrightarrow \{\text{VERO}, \text{FALSO}\}$ può essere scritto (D, S), dove $S = \{a \in D | \ P(a) = \text{VERO}\}$, o, semplicemente, S se il dominio D è chiaro dal contesto. In tal modo la relazione *batte* può essere scritta

$$\{(\text{FORBICI, CARTA}),\ (\text{CARTA, SASSO}),\ (\text{SASSO, FORBICI})\}.$$

Un particolare tipo di relazione binaria, chiamata ***relazione di equivalenza***, coglie il concetto che due oggetti risultano uguali rispetto ad una qualche caratteristica. Una relazione binaria R è una relazione di equivalenza se R soddisfa tre condizioni:

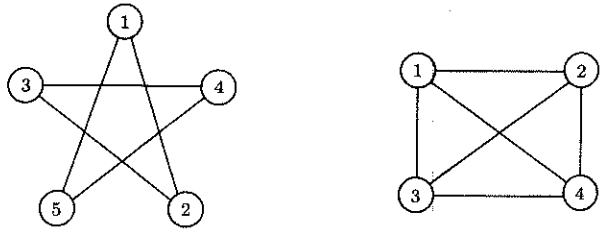
1. R è ***riflessiva*** se per ogni x, xRx;
2. R è ***simmetrica*** se per ogni x e y, xRy implica yRx; e
3. R è ***transitiva*** se per ogni x, y e z, xRy e yRz implica xRz.

ESEMPIO 0.11

Definiamo una relazione di equivalenza sui numeri naturali, denotata con $\equiv_7$. Per $i, j \in \mathcal{N}$ diciamo che $i \equiv_7 j$, se $i - j$ è un multiplo di 7. Questa è una relazione di equivalenza perché soddisfa le tre condizioni. È riflessiva, in quanto $i - i = 0$, che è un multiplo di 7. È simmetrica, in quanto $i - j$ è un multiplo di 7, se $j - i$ è un multiplo di 7. È transitiva, in quanto se $i - j$ e $j - k$ sono entrambi multipli di 7, allora $i - k = (i - j) + (j - k)$ è la somma di due multipli di 7 e quindi è anch'esso un multiplo di 7.

Grafi

Un ***grafo non orientato***, o semplicemente un ***grafo***, è un insieme di punti e di linee che connettono alcuni dei punti. I punti sono chiamati ***nodi*** o ***vertici***, e le linee sono chiamate ***archi***, come mostrato nella seguente figura.



(a) (b)

FIGURA 0.12
Esempi di grafi

Il numero di archi ad un particolare nodo è il ***grado*** di tale nodo. Nella Figura 0.12(a) tutti i nodi hanno grado 2. Nella Figura 0.12(b) tutti i nodi hanno grado 3. Non è permesso più di un arco tra due nodi dati. Possiamo, in alcuni casi, permettere l'esistenza di un arco da un nodo a se stesso, chiamato ***self-loop***.

In un grafo G che contiene i nodi i e j, la coppia (i, j) rappresenta l'arco che collega i e j. L'ordine di i e j non è importante in un grafo non orientato, quindi le coppie (i, j) e (j, i) rappresentano lo stesso arco. Talvolta descriviamo archi non orientati con coppie non ordinate utilizzando la notazione insiemistica $\{i, j\}$. Se V è l'insieme di nodi di G ed E è l'insieme degli archi, scriviamo $G = (V, E)$. Possiamo descrivere un grafo con un diagramma o più formalmente specificando V ed E. Per esempio, una descrizione formale del grafo in Figura 0.12(a) è

$$\big(\{1,2,3,4,5\},\ \{(1,2),\ (2,3),\ (3,4),\ (4,5),\ (5,1)\}\big),$$

ed una descrizione formale del grafo in Figura 0.12(b) è

$$\big(\{1,2,3,4\},\ \{(1,2),\ (1,3),\ (1,4),\ (2,3),\ (2,4),\ (3,4)\}\big).$$

I grafi sono frequentemente usati per la rappresentazione di dati. I nodi potrebbero essere città e gli archi delle autostrade di collegamento, oppure i nodi potrebbero essere persone e gli archi le amicizie tra di loro. A volte, per comodità, assegniamo un'etichetta (o label) ai nodi e/o agli archi di un grafo che poi è chiamato un ***grafo etichettato***. La Figura 0.13 illustra un grafo i cui nodi sono città ed i cui archi sono etichettati con la tariffa del volo diretto più economico per viaggiare tra quelle città, se un tale volo esiste.

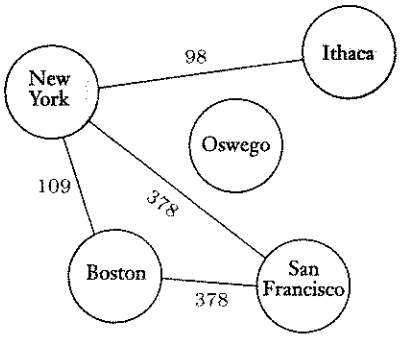


FIGURA 0.13
Tariffe più economiche dei voli diretti tra alcune città

Diciamo che il grafo G è un ***sottografo*** del grafo H se i nodi di G sono un sottoinsieme dei nodi di H, e gli archi di G sono anche archi di H sui nodi corrispondenti. La figura seguente mostra un grafo H e un suo sottografo G.

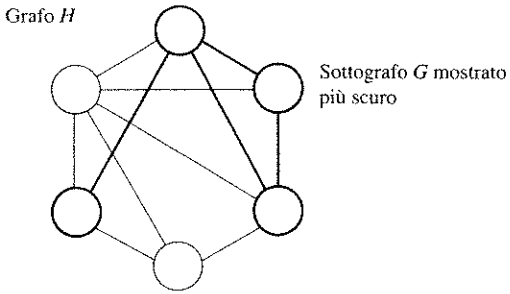


FIGURA 0.14
Il grafo G (più scuro) è un sottografo di H

Un ***cammino*** in un grafo è una sequenza di nodi collegati da archi. Un ***cammino semplice*** è un cammino che non contiene nodi ripetuti. Un grafo è ***connesso*** se per ogni coppia di nodi esiste un cammino tra di loro. Un cammino è un ***ciclo*** se inizia e termina nello stesso nodo. Un ***ciclo semplice*** è quello che contiene almeno tre nodi e ripete solo il primo e l'ultimo nodo. Un grafo è un ***albero*** se è connesso e non ha cicli semplici, come mostrato nella Figura 0.15. Un albero può contenere un nodo speciale chiamato ***radice***. I nodi di grado 1 in un albero, ad eccezione della radice, sono chiamati ***foglie*** dell'albero.

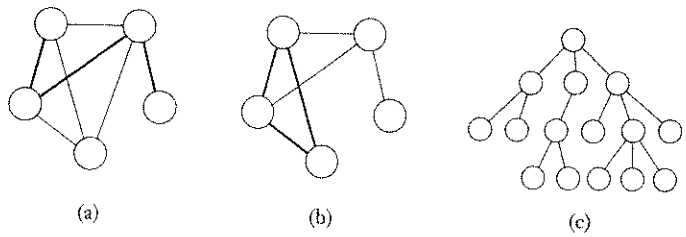


FIGURA 0.15
(a) Un cammino in un grafo, (b) un ciclo in un grafo, (c) un albero

Un ***grafo orientato*** o diretto ha frecce invece di linee, come mostrato nella seguente figura. Il numero delle frecce che escono da un particolare nodo è il ***grado uscente*** di quel nodo, ed il numero delle frecce che entrano in un particolare nodo è il ***grado entrante***.

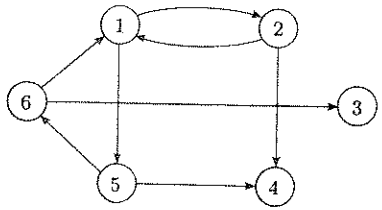


FIGURA 0.16
Un grafo orientato

In un grafo orientato rappresentiamo un arco da i a j come una coppia (i, j). La descrizione formale di un grafo orientato G è (V, E) dove V è l'insieme dei nodi ed E è l'insieme degli archi. La descrizione formale del grafo in Figura 0.16 è

$$(\{1,2,3,4,5,6\}, \{(1,2), (1,5), (2,1), (2,4), (5,4), (5,6), (6,1), (6,3)\}).$$

Un cammino in cui tutte le frecce puntano nella stessa direzione del cammino si definisce ***cammino orientato*** (o diretto). Un grafo orientato è ***fortemente connesso*** se per ogni coppia di vertici esiste almeno un cammino orientato che li collega. I grafi orientati sono utili per descrivere

relazioni binarie. Se R è una relazione binaria il cui dominio è $D \times D$, un grafo etichettato $G = (D, E)$ rappresenta R, dove $E = \{(x, y) | \, xRy\}$.

ESEMPIO 0.17

Il grafo diretto qui mostrato rappresenta la relazione data nell'Esempio 0.10.

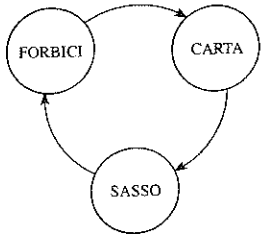


FIGURA 0.18
Il grafo della relazione *batte*

Stringhe e linguaggi

Le stringhe di caratteri sono i mattoni fondamentali dell'informatica. L'alfabeto su cui le stringhe sono definite può variare a seconda dell'applicazione. Per i nostri scopi, si definisce ***alfabeto*** un qualsiasi insieme finito non vuoto. Gli elementi dell'alfabeto sono i ***simboli*** dell'alfabeto. In genere usiamo le lettere greche maiuscole Σ e Γ per indicare alfabeti e un font di caratteri del tipo "typewriter" per i simboli di un alfabeto. Di seguito sono riportati alcuni esempi di alfabeti.

$$\Sigma_1 = \{0,1\}$$
$$\Sigma_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$$
$$\Gamma = \{0, 1, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$$

Una ***stringa su di un alfabeto*** è una sequenza finita di simboli dell'alfabeto, solitamente scritti uno di seguito all'altro e non separati da virgole. Se $\Sigma_1 = \{0,1\}$, allora 01001 è una stringa su Σ_1. Se $\Sigma_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$, allora abracadabra è una stringa su Σ_2. Se w è una stringa su Σ, la ***lunghezza*** di w, indicata con $|w|$, è il numero di simboli che essa contiene. La stringa di lunghezza zero è chiamata ***stringa vuota*** ed è indicata con ε. La stringa vuota svolge il ruolo dello 0 in un sistema numerico.

Se w ha lunghezza n, allora possiamo scrivere $w = w_1 w_2 \cdots w_n$ dove ogni $w_i \in \Sigma$. L'***inversa*** di w, indicato con $w^{\mathcal{R}}$, è la stringa ottenuta scrivendo w in ordine inverso (ovvero, $w_n w_{n-1} \cdots w_1$). La stringa z è una ***sottostringa*** di w se z appare consecutivamente in w. Ad esempio, cad è una sottostringa di abracadabra. Data una stringa x di lunghezza m ed una stringa y di lunghezza n, la ***concatenazione*** di x e y, scritta xy, è la stringa ottenuta aggiungendo y alla fine di x, come in $x_1 \cdots x_m y_1 \cdots y_n$. Per concatenare una stringa con se stessa più volte usiamo la notazione x^k: per indicare

$$\overbrace{xx \cdots x}^{k}.$$

L'***ordine lessicografico*** delle stringhe è lo stesso tipo di ordine del dizionario. Utilizzeremo in alcuni casi un ordine lessicografico modificato, detto semplicemente ***ordine per lunghezza***, che è identico all'ordine lessicografico, tranne per il fatto che stringhe più corte precedono quelle più lunghe. Quindi l'ordine per lunghezza di tutte le stringhe sull'alfabeto {0,1} è

$$(\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots).$$

Diciamo che x è un ***prefisso*** della stringa y se esiste una stringa z tale che $xz = y$, e diciamo che x è un ***prefisso proprio*** di y se risulta anche $x \neq y$.

Un ***linguaggio*** è un insieme di stringhe. Un linguaggio è ***prefisso*** se nessun elemento è prefisso di un altro.

Logica booleana

La ***logica booleana*** è un sistema matematico costruito intorno ai due valori VERO e FALSO. Anche se originariamente concepito come matematica pura, questo sistema è ormai considerato alla base dell'elettronica digitale e della progettazione di calcolatori. I valori VERO e FALSO sono chiamati ***valori booleani*** e sono spesso rappresentati dai valori 1 e 0. Utilizziamo i valori booleani in situazioni con due possibili stati, come un cavo che può avere un alto o un basso voltaggio, una proposizione che può essere vera o falsa, o una domanda che può avere risposta si o no. Siamo in grado di manipolare i valori booleani mediante le ***operazioni booleane***.

La più semplice di tali operazioni è l'operazione di ***negazione*** o **not** indicata con il simbolo $\neg$. La negazione di un valore booleano è il valore opposto. Così $\neg 0 = 1$ e $\neg 1 = 0$. La ***congiunzione*** o **and** è indicata con il simbolo $\wedge$. La congiunzione di due valori booleani è 1 se entrambi i valori sono 1.

La ***disgiunzione*** o **or** è indicata con il simbolo $\vee$. La disgiunzione di due valori booleani è 1 se almeno uno di questi valori è 1. Riassumiamo queste informazioni come segue.

$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$\neg 0 = 1$
$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$\neg 1 = 0$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	
$1 \wedge 1 = 1$	$1 \vee 1 = 1$	

Usiamo le operazioni booleane per combinare asserzioni semplici in espressioni booleane più complesse, esattamente come si fa con le operazioni aritmetiche + e × per costruire espressioni aritmetiche complesse. Per esempio se P è il valore booleano che rappresenta la veridicità dell'affermazione "il sole splende" e Q rappresenta la veridicità dell'affermazione "oggi è lunedì", possiamo scrivere $P \wedge Q$ per rappresentare la veridicità dell'affermazione "il sole splende *and* oggi è lunedì" ed allo stesso modo per $P \vee Q$ con *and* sostituito da *or*. I valori P e Q sono chiamati ***operandi*** dell'operazione.

Esistono anche altre operazioni booleane. L'operazione di ***or esclusivo***, o **xor**, è denotata con il simbolo $\oplus$ e dà valore 1 se esattamente uno dei due operandi è 1. L'operazione di ***uguaglianza***, denotata con $\leftrightarrow$, è 1 se entrambi gli operandi hanno lo stesso valore. Infine, l'operazione di ***implicazione***, denotata dal simbolo $\rightarrow$, è 0 se il suo primo operando ha valore 1 ed il suo secondo operando ha valore 0; altrimenti $\rightarrow$ è 1. Riassumiamo queste operazioni come segue.

$$\begin{array}{lll} 0 \oplus 0 = 0 & 0 \leftrightarrow 0 = 1 & 0 \rightarrow 0 = 1 \\ 0 \oplus 1 = 1 & 0 \leftrightarrow 1 = 0 & 0 \rightarrow 1 = 1 \\ 1 \oplus 0 = 1 & 1 \leftrightarrow 0 = 0 & 1 \rightarrow 0 = 0 \\ 1 \oplus 1 = 0 & 1 \leftrightarrow 1 = 1 & 1 \rightarrow 1 = 1 \end{array}$$

Possiamo stabilire varie relazioni tra queste operazioni. Infatti, siamo in grado di esprimere tutte le operazioni booleane in termini di operazioni AND e NOT come mostrato dalle seguenti uguaglianze. Le due espressioni in ogni riga sono equivalenti. Ogni riga esprime l'operazione nella colonna di sinistra in termini delle operazioni delle righe sovrastanti e delle operazioni AND e NOT.

$$\begin{array}{ll} P \vee Q & \neg(\neg P \wedge \neg Q) \\ P \rightarrow Q & \neg P \vee Q \\ P \leftrightarrow Q & (P \rightarrow Q) \wedge (Q \rightarrow P) \\ P \oplus Q & \neg(P \leftrightarrow Q) \end{array}$$

Nel manipolare espressioni booleane risulta utile la ***legge distributiva*** per AND e OR. Essa è simile alla legge distributiva per addizione e moltiplicazione, che stabilisce che $a \times (b+c) = (a \times b) + (a \times c)$. La versione booleana è disponibile in due forme:

- $P \wedge (Q \vee R)$ è uguale a $(P \wedge Q) \vee (P \wedge R)$,
- $P \vee (Q \wedge R)$ è uguale a $(P \vee Q) \wedge (P \vee R)$.

Indice dei termini matematici

Albero	Un grafo connesso senza cicli semplici
Alfabeto	Un insieme finito di oggetti chiamati simboli
Arco	Una linea in un grafo
Argomento	Un input di una funzione
Complemento	Un'operazione su un insieme, che produce l'insieme di tutti gli elementi non presenti
Concatenazione	Un'operazione che unisce stringhe di un insieme con stringhe di un altro insieme
Congiunzione	Operazione booleana AND
Grafo connesso	Un grafo con un cammino per ogni coppia di nodi
Ciclo	Un cammino che inizia e termina nel medesimo nodo
Coppia ordinata	Una lista di due elementi, detta anche 2-tupla
Coppia non ordinata	Un insieme di due elementi
Cammino	Una sequenza di nodi di un grafo connessi da archi
Cammino semplice	Un cammino senza ripetizioni
Codominio	L'insieme che contiene l'output di una funzione
Disgiunzione	Operazione booleana OR
Dominio	L'insieme dei possibili input di una funzione
Elemento	Un oggetto in un insieme
Funzione	Un'operazione che trasforma un input in un output
Grafo	Una collezione di punti e linee che collegano coppie di punti
Grafo orientato	Una collezione di punti e frecce che collegano coppie di punti
Insieme	Un gruppo di oggetti
Insieme vuoto	L'insieme senza elementi
Insieme singleton	Un insieme contenente un solo elemento
Intersezione	Un'operazione su insiemi che produce l'insieme degli elementi comuni
k-tupla	Una lista di k oggetti
Linguaggio	Un insieme di stringhe
Membro	Un oggetto in un insieme
Nodo	Un punto in un grafo
Operazione booleana	Un'operazione su valori booleani
Prodotto cartesiano	Un'operazione tra insiemi che produce l'insieme di tutte le tuple di elementi dei rispettivi insiemi.
Predicato	Una funzione il cui codominio è {VERO, FALSO}
Proprietà	Un predicato
Relazione	Un predicato, in genere quando il dominio è un insieme di k-tuple
Relazione binaria	Una relazione il cui dominio è un insieme di coppie
Relazione di equivalenza	Una relazione binaria che è: riflessiva, simmetrica e transitiva
Sequenza	Una lista di oggetti
Simbolo	Un elemento di un alfabeto
Stringa	Una lista finita di simboli di un alfabeto
Stringa vuota	Stringa di lunghezza zero
Unione	Un'operazione su insiemi che combina tutti gli elementi in un unico insieme
Valore booleano	I valori VERO o FALSO, spesso rappresentati da 1 o 0
Vertice	Un punto in un grafo

0.3
DEFINIZIONI, TEOREMI E DIMOSTRAZIONI

Teoremi e dimostrazioni sono l'anima e il corpo della matematica e le definizioni ne sono lo spirito. Questi tre elementi sono al centro di ogni argomento matematico, compresi i nostri. Le ***definizioni*** descrivono gli oggetti e le nozioni che utilizziamo. Una definizione può essere semplice, come quella di *insieme* data precedentemente in questo capitolo, o complessa come la definizione di *sicurezza* in un sistema crittografico. La precisione è essenziale per qualsiasi definizione matematica. Quando si definisce un oggetto dobbiamo chiarire ciò che costituisce tale oggetto e ciò che non lo costituisce. Dopo aver definito vari oggetti e nozioni, di solito formuliamo degli ***enunciati matematici*** su di essi. Un enunciato afferma tipicamente che un oggetto ha una certa proprietà. L'enunciato può essere vero o meno, ma come una definizione, deve essere preciso. Non ci deve essere nessuna ambiguità sul suo significato. Una ***dimostrazione*** è un ragionamento logico convincente che un enunciato è vero. In matematica un ragionamento deve essere ineccepibile, il che vuol dire, convincente in senso assoluto. Nella vita di tutti i giorni o in giurisprudenza, l'onere della prova è inferiore. Un processo per omicidio esige una prova "oltre ogni ragionevole dubbio". Il peso delle prove può convincere la giuria ad accettare l'innocenza o la colpevolezza dell'indagato. Tuttavia, l'evidenza non ha alcun ruolo in una dimostrazione matematica. Un matematico esige la dimostrazione oltre *qualsiasi* dubbio.

Un ***teorema*** è un enunciato matematico che si dimostra essere vero. Generalmente ci riserviamo l'uso del termine per le affermazioni di particolare interesse. Occasionalmente dimostriamo enunciati interessanti solo perché servono alla dimostrazione di un altro enunciato più significativo. Un tale enunciato è chiamato ***lemma***. A volte un teorema o la sua dimostrazione può consentire di concludere facilmente che altri enunciati correlati sono veri. Questi enunciati sono chiamati ***corollari*** del teorema.

Dimostrazioni

L'unico modo per determinare la verità o falsità di un enunciato matematico è mediante una dimostrazione matematica. Purtroppo, trovare dimostrazioni non è sempre facile. Non è riducibile ad un semplice insieme di regole o processi. Durante questo corso, vi sarà chiesto di esibire dimostrazioni di vari enunciati. Non disperatevi al pensiero! Anche se nessuno ha una ricetta per la produzione di dimostrazioni, un valido aiuto ci viene dall'esistenza di strategie generali. Come prima cosa, leggete attentamente l'enunciato che si desidera provare. Avete capito tutta la notazione? Riscrivete l'enunciato con parole vostre. Scomponetelo e considerate ogni parte separatamente.

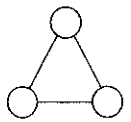
Talvolta le parti di un'enunciato composito non sono immediatamente chiare. Un tipo frequente di enunciato composto da più parti è della forma "P se e solo se Q", spesso scritto "P sse Q", dove sia P che Q sono enunciati matematici. Questa notazione è un'abbreviazione di un enunciato formato da due parti. La prima parte è "P solo se Q," che significa: se P è vero, allora Q è vero, scritta $P \Rightarrow Q$. La seconda è "P se Q," che significa: se Q è vero, allora P è vero, scritta $P \Leftarrow Q$. La prima di queste due parti indica la ***direzione diretta*** dell'enunciato originale e la seconda è la ***direzione inversa***. Scriviamo "P se e solo se Q" come $P \iff Q$. Per dimostrare un enunciato di questo tipo è necessario dimostrare ciascuna delle due direzioni. Spesso, una di queste direzioni risulta più semplice da dimostrare dell'altra.

Un altro tipo di enunciato composito afferma che due insiemi A e B sono uguali. La prima parte afferma che A è un sottoinsieme di B, e la seconda parte afferma che B è un sottoinsieme di A. Quindi un modo comune per dimostrare che $A = B$ consiste nel dimostrare che ogni elemento di A è anche un elemento di B e che ogni elemento di B è anche un elemento di A.

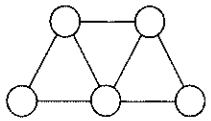
Inoltre, quando volete dimostrare un enunciato o una sua parte, cercate di avere un intuizione del perché dovrebbe essere vero. Sperimentare con esempi è particolarmente utile. Per esempio, se un enunciato dice che tutti gli oggetti di un certo tipo hanno una particolare proprietá, scegliete un paio di oggetti di quel tipo e osservate che effettivamente hanno quella proprietà. Fatto questo, cercate di trovare un oggetto che non riesce ad avere la proprietà – si dice ***controesempio***. Se l'enunciato è vero, non sarà possibile trovare un controesempio. Capire dove sorgono le difficoltà nel determinare un controesempio può aiutare a capire perché l'enunciato è vero.

ESEMPIO 0.19

Si supponga di voler dimostrare il seguente enunciato: *per ogni grafo G, la somma dei gradi di tutti i nodi in G è un numero pari.* In primo luogo, scegliete un paio di grafi ed osservate l'applicazione di questo enunciato. Ecco due esempi.

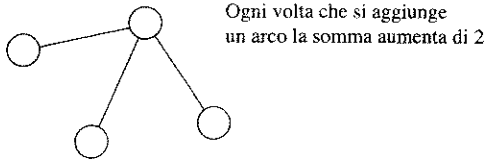


somma = 2+2+2 = 6



somma = 2+3+4+3+2 = 14

Quindi, cercate di trovare un controesempio, cioè un grafo in cui la somma è un numero dispari.



Potete iniziare a capire perché l'enunciato è vero e come provarlo?

Se siete ancora bloccati nel tentativo di dimostrare un enunciato, provate qualcosa di più facile. Tentate di dimostrare un caso particolare dell'enunciato. Per esempio, se si sta cercando di dimostrare che qualche proprietà è vera per ogni $k > 0$, prima cercate di dimostrarlo per $k = 1$. Se ci riuscite, provatelo per $k = 2$, e così via fino ad arrivare a capire il caso generale. Se un caso speciale è difficile da dimostrare, provate un caso speciale diverso o anche un caso speciale del caso speciale. Infine, quando pensate di aver trovato la dimostrazione, è necessario scrivere tutto con precisione. Una prova ben scritta è una sequenza di enunciati, in cui ognuno segue per semplice ragionamento dai precedenti nella sequenza. Scrivere con attenzione una dimostrazione è importante, sia per consentire a chi legge di capirla, sia per essere sicuri che sia esente da errori.

Di seguito sono riportati alcuni suggerimenti per produrre una dimostrazione.

- *Essere pazienti.* Trovare prove richiede tempo. Se non riuscite a farlo subito, non vi preoccupate. I ricercatori a volte lavorano per settimane o anche anni per trovare una sola dimostrazione.
- *Ritornarci su.* Guardate oltre l'enunciato che volete dimostrare, pensate un po', lasciate, e poi ritornateci su dopo pochi minuti o anche ore. Lasciate che l'inconscio, la parte intuitiva della vostra mente abbia la possibilità di lavorare.
- *Essere precisi.* Quando state elaborando la vostra intuizione per l'enunciato che state tentando di dimostrare, utilizzate semplici immagini e/o testi. State tentando di trasformare la vostra intuizione in un'affermazione, la mancanza di precisione si interpone al vostro intuito. Inoltre, quando si sta scrivendo una soluzione che un'altra persona dovrà leggere, la chiarezza aiuterà quella persona a capire.
- *Essere concisi.* La sintesi aiuta ad esprimere le proprie idee ad un livello superiore senza perdersi in dettagli. Una buona notazione matematica è utile per esprimere le idee in modo conciso. Ma siate sicuri di includere abbastanza del vostro ragionamento nello scrivere una dimostrazione in modo che il lettore possa facilmente capire che cosa state tentando di dire.

Per far pratica, cerchiamo di dimostrare una delle leggi di DeMorgan.

TEOREMA 0.20

Per ogni coppia di insiemi A e B, $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

In primo luogo, il significato di questo teorema è chiaro? Se non si capisce il significato dei simboli $\cup$ o $\cap$ o la barra superiore, rivedete la discussione a pagina 5.

Per dimostrare questo teorema dobbiamo dimostrare che i due insiemi $\overline{A \cup B}$ e $\overline{A} \cap \overline{B}$ sono uguali. Ricordate che possiamo dimostrare che due insiemi sono uguali dimostrando che ogni elemento di un insieme è anche elemento dell'altro e viceversa. Prima di guardare la dimostrazione seguente, prendete in considerazione qualche esempio e poi cercate di provarlo da soli.

DIMOSTRAZIONE. Questo teorema afferma che i due insiemi $\overline{A \cup B}$ e $\overline{A} \cap \overline{B}$ sono uguali. Dimostriamo questa affermazione dimostrando che ogni elemento di uno è anche un elemento dell'altro e viceversa.

Supponiamo che x sia un elemento di $\overline{A \cup B}$. Quindi x non è in $A \cup B$, dalla definizione di complemento di un insieme. Quindi x non è in A e x non è in B, dalla definizione di unione di insiemi. In altre parole, x è in $\overline{A}$ ed x è in $\overline{B}$. Quindi la definizione di intersezione di insiemi mostra che x è in $\overline{A} \cap \overline{B}$.

Per l'altro verso, supponiamo che x sia in $\overline{A} \cap \overline{B}$. Allora x è sia in $\overline{A}$ che in $\overline{B}$. Quindi x non è in A e x non è in B, e quindi neanche nell'unione di questi due insiemi. Quindi x è nel complemento dell'unione di questi insiemi; in altre termini, x è in $\overline{A \cup B}$, il che completa la dimostrazione del teorema.

Dimostriamo ora l'enunciato dell'Esempio 0.19.

TEOREMA 0.21

Per ogni grafo G, la somma dei gradi di tutti i nodi di G è un numero pari.

DIMOSTRAZIONE. Ogni arco in G è connesso a due nodi. Ogni arco contribuisce di 1 al grado di ogni nodo a cui esso è collegato. Quindi, ogni arco contribuisce di 2 alla somma dei gradi di tutti i nodi. Questo implica

che se G contiene e archi, allora la somma dei gradi di tutti i nodi di G è $2e$, che è un numero pari.

0.4
TIPI DI DIMOSTRAZIONI

Vari tipi di ragionamenti trovano posto nelle dimostrazioni matematiche. Qui, ne descriviamo alcuni che si trovano spesso nella teoria della computazione. Si noti che una dimostrazione può contenere più di un tipo di ragionamento, perché la dimostrazione può contenere al suo interno diverse sotto-dimostrazioni.

Dimostrazioni per costruzione

Molti teoremi affermano l'esistenza di un particolare tipo di oggetto. Un modo per dimostrare un tale teorema consiste nel mostrare come costruire l'oggetto. Questa tecnica è detta ***dimostrazione costruttiva***. Usiamo una dimostrazione per costruzione per dimostrare il seguente teorema. Diciamo che un grafo è ***k-regolare*** se ogni nodo del grafo ha grado k.

TEOREMA 0.22

Per ogni numero pari n maggiore di 2, esiste un grafo 3-regolare con n nodi.

DIMOSTRAZIONE. Sia n un numero pari maggiore di 2. Costruiamo un grafo $G = (V, E)$ con n vertici come segue. L'insieme dei vertici di G è $V = \{0, 1, \ldots, n-1\}$ e l'insieme di archi di G è l'insieme:

$$E = \{\{i, i+1\} \mid \text{per } 0 \le i \le n-2\} \cup \{\{n-1, 0\}\}$$
$$\cup \{\{i, i+n/2\} \mid \text{per } 0 \le i \le n/2-1\}.$$

Immaginate i vertici di questo grafo scritti consecutivamente lungo la circonferenza di un cerchio. In tal caso, gli archi descritti nella riga superiore di E connettono coppie di vertici adiacenti intorno al cerchio. Gli archi descritti nella riga inferiore di E connettono coppie di vertici situati sui lati opposti del cerchio. Questa immagine costruita mentalmente mostra chiaramente che ogni vertice di G ha grado 3.

Dimostrazioni per assurdo

Frequentemente per dimostrare un teorema, assumiamo che il teorema sia falso per poi dimostrare che questa ipotesi porta ad una conseguenza chiaramente falsa, detta contraddizione o assurdo. Usiamo questo tipo di ragionamento anche nella vita di tutti i giorni, come nel seguente esempio.

ESEMPIO 0.23

Jack vede Jill, che è appena rientrato. Osservando che lei è completamente asciutta, sa che non piove. La sua "dimostrazione" che non piove è la seguente: *se piovesse* (l'assunto che l'affermazione è falsa), *Jill dovrebbe essere bagnata* (la conseguenza, ovviamente falsa). Quindi, non piove.

Ora, cerchiamo di dimostrare per assurdo che la radice quadrata di 2 è un numero irrazionale. Un numero è ***razionale*** se è una frazione $\frac{m}{n}$, dove m ed n sono numeri interi, in altre parole, un numero razionale è il rapporto di due interi. Per esempio, $\frac{2}{3}$ è ovviamente un numero razionale. Un numero è ***irrazionale*** se non è razionale.

TEOREMA 0.24

$\sqrt{2}$ è irrazionale

DIMOSTRAZIONE. In primo luogo, assumiamo al fine di ottenere una contraddizione, che $\sqrt{2}$ è razionale. Quindi

$$\sqrt{2} = \frac{m}{n},$$

dove m ed n sono interi. Se m ed n sono entrambi divisibili per uno stesso numero intero maggiore di 1, allora dividiamoli entrambi per il più grande di tali divisori. In questo modo non cambia il valore della frazione. Ora, almeno uno tra m ed n deve essere un numero dispari. Moltiplichiamo entrambi i lati dell'equazione per n e otteniamo

$$n\sqrt{2} = m.$$

Eleviamo al quadrato entrambi i lati dell'equazione e otteniamo

$$2n^2 = m^2.$$

Poiché m^2 è 2 volte n^2, allora m^2 è pari. Quindi anche m è pari, poiché il quadrato di un numero dispari è sempre dispari. Quindi possiamo scrivere $m = 2k$ per un qualche intero k. Sostituendo $2k$ ad m, otteniamo

$$2n^2 = (2k)^2$$
$$= 4k^2.$$

Dividendo entrambi il lati per 2, otteniamo

$$n^2 = 2k^2.$$

Ma questo risultato dimostra che n^2 è pari e quindi che n è pari. In tal modo abbiamo stabilito che sia m che n sono pari. Però avevamo prima ridotto m ed n in modo che *non* fossero entrambi pari, quindi siamo giunti ad una contraddizione.

Dimostrazioni per induzione

Una dimostrazione per induzione è un metodo avanzato utilizzato per mostrare che tutti gli elementi di un insieme infinito godono di una data proprietà. Per esempio, possiamo utilizzare una dimostrazione per induzione per dimostrare che un'espressione aritmetica calcola una quantità desiderata per ogni assegnamento alle sue variabili o che un programma funziona correttamente per qualsiasi input. Per illustrare come funziona una dimostrazione per induzione, prendiamo come insieme infinito quello dei numeri naturali, $\mathcal{N} = \{1, 2, 3, \ldots\}$, e chiamiamo $\mathcal{P}$ la proprietà. Il nostro obiettivo è dimostrare che $\mathcal{P}(k)$ è vera per ogni numero naturale k. In altre parole, vogliamo dimostrare che $\mathcal{P}(1)$ è vera, così come $\mathcal{P}(2)$, $\mathcal{P}(3)$, $\mathcal{P}(4)$, e così via.

Ogni dimostrazione per induzione consiste di due parti, la ***base*** ed il ***passo induttivo***. Ogni parte è una dimostrazione a se stante. La base induttiva dimostra che $\mathcal{P}(1)$ è vera. Il passo induttivo mostra che per ogni $i \geq 1$, se $\mathcal{P}(i)$ vera, allora lo è anche $\mathcal{P}(i+1)$.

Quando abbiamo provato entrambe le parti, abbiamo raggiunto il risultato desiderato, e cioè che $\mathcal{P}(i)$ è vera per ogni i. Perché? In primo luogo, sappiamo che $\mathcal{P}(1)$ è vera perché la base da sola lo dimostra. In secondo luogo, sappiamo che $\mathcal{P}(2)$ è vera, perché il passo induttivo dimostra che, se $\mathcal{P}(1)$ è vera, allora $\mathcal{P}(2)$ è vera, e noi sappiamo che $\mathcal{P}(1)$ è vera. In terzo luogo, sappiamo che $\mathcal{P}(3)$ è vera, perché il passo induttivo dimostra che, se $\mathcal{P}(2)$ è vera, allora $\mathcal{P}(3)$ è vera, e noi ora sappiamo che $\mathcal{P}(2)$ è vera. Questo processo continua per tutti i numeri naturali, mostrando che $\mathcal{P}(4)$ è vera, $\mathcal{P}(5)$ è vera, e così via.

Una volta capito il paragrafo precedente, si possono facilmente comprendere varianti e generalizzazioni della stessa idea. Per esempio, la base non deve necessariamente iniziare con 1, ma può iniziare con qualsiasi valore b. In tal caso la prova per induzione mostra che $\mathcal{P}(k)$ è vera per ogni k che è almeno b.

Nel passo induttivo l'assunzione che $\mathcal{P}(i)$ è vera è detta ***ipotesi induttiva***. A volte può essere utile avere l'ipotesi induttiva più forte che $\mathcal{P}(j)$ è vera per ogni $j \leq i$. La prova per induzione rimane valida perché, quando vogliamo dimostrare che $\mathcal{P}(i+1)$ è vera, abbiamo già dimostrato che $\mathcal{P}(j)$ è vera per ogni $j \leq i$. Il formato per scrivere una dimostrazione per induzione è il seguente.

Base: Provare che $\mathcal{P}(1)$ è vera.

⋮

Passo induttivo: Per ogni $i \geq 1$, assumere che $\mathcal{P}(i)$ è vera ed utilizzare tale assunzione per mostrare che $\mathcal{P}(i+1)$ è vera.

⋮

Cerchiamo ora di dimostrare per induzione la correttezza della formula utilizzata per calcolare le dimensioni dei pagamenti mensili dei mutui-casa. Quando acquistano una casa, molte persone prendono in prestito parte del denaro necessario per l'acquisto, per poi rimborsare il prestito in un certo numero di anni. In genere, i termini di tali rimborsi stabiliscono che ogni mese viene pagata una rata che copre gli interessi e una parte della somma originale, in modo che il totale sia restituito, per esempio, in 30 anni. La formula per calcolare il valore della rata mensile sembra avvolta dal mistero, ma in realtà è abbastanza semplice. Riguardando la vita di molte persone, conoscerla dovrebbe risultarvi interessante. Useremo l'induzione per dimostrare che la formula funziona, fornendo così un buon esempio di tale tecnica.

In primo luogo, diamo nome e significato alle varie variabili. Sia P il *prestito iniziale*, che rappresenta l'importo del mutuo. Sia $I > 0$ il tasso di *interesse* annuo del prestito, dove I = 0.06 indica un tasso del 6% di interesse. Sia Y la rata mensile. Per comodità definiamo un'altra variabile M, dipendente da I, come moltiplicatore mensile. Rappresenta il tasso secondo il quale il prestito cambia ogni mese a causa degli interessi che gravino su di esso. Seguendo il linguaggio bancario definiamo il tasso di interesse mensile pari ad un dodicesimo di quello annuo, cosí che $M = 1 + I/12$. Gli interessi sono pagati su base mensile. Sono dunque due le cose che accadono ogni mese. Primo, l'importo del prestito tende ad aumentare per effetto del moltiplicatore mensile. Secondo, l'importo del prestito tende a diminuire per effetto del pagamento effettuato mensilmente. Fissiamo P_t come l'importo del mutuo rimanente dopo il t-esimo mese. Quindi $P_0 = P$ è l'importo iniziale del prestito, $P_1 = MP_0 - Y$ è l'importo del prestito dopo un mese, $P_2 = MP_1 - Y$ è l'importo del prestito dopo due mesi, e così via. Ora siamo pronti ad enunciare e dimostrare un teorema per induzione su t che fornisce una formula per il valore di P_t.

TEOREMA 0.25

Per ogni $t \geq 0$,

$$P_t = PM^t - Y\left(\frac{M^t - 1}{M - 1}\right).$$

DIMOSTRAZIONE.

Base: Dimostriamo che la formula è vera per $t = 0$. Se $t = 0$, allora la formula diventa

$$P_0 = PM^0 - Y\left(\frac{M^0 - 1}{M - 1}\right).$$

Possiamo semplificare il lato destro osservando che $M^0 = 1$. Quindi otteniamo

$$P_0 = P,$$

che è vera perché abbiamo definito P_0 pari a P. Quindi abbiamo dimostrato che la base induttiva è vera.

Passo induttivo: Per ogni $k \geq 0$, assumiamo che la formula sia vera per $t = k$ e dimostriamo che essa è vera per $t = k + 1$. L'ipotesi induttiva stabilisce che

$$P_k = PM^k - Y\left(\frac{M^k - 1}{M - 1}\right).$$

Vogliamo dimostrare che

$$P_{k+1} = PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right).$$

Lo facciamo con i passi seguenti. Innanzitutto, dalla definizione di P_{k+1} in termini di P_k, sappiamo che

$$P_{k+1} = P_k M - Y.$$

Quindi, usando l'ipotesi induttiva per calcolare P_k,

$$P_{k+1} = \left[PM^k - Y\left(\frac{M^k - 1}{M - 1}\right)\right] M - Y.$$

Effettuando la moltiplicazione per M e riscrivendo Y, otteniamo

$$\begin{aligned} P_{k+1} &= PM^{k+1} - Y\left(\frac{M^{k+1} - M}{M - 1}\right) - Y\left(\frac{M - 1}{M - 1}\right) \\ &= PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right). \end{aligned}$$

Quindi la formula è corretta per $t = k + 1$, il che dimostra il teorema.

Nel problema 0.14 si chiede di utilizzare la formula precedente per il calcolo effettivo del mutuo.

ESERCIZI

0.1 Esaminare le seguenti descrizioni formali di insiemi in modo da stabilire gli elementi che essi contengono. Scrivere una breve descrizione informale in italiano di ogni insieme.

- **a.** $\{1, 3, 5, 7, \ldots\}$
- **b.** $\{\ldots, -4, -2, 0, 2, 4, \ldots\}$
- **c.** $\{n \mid n = 2m$ per qualche m in $\mathcal{N}\}$
- **d.** $\{n \mid n = 2m$ per qualche m in $\mathcal{N}$, e $n = 3k$ per qualche k in $\mathcal{N}\}$
- **e.** $\{w \mid w$ è una stringa di 0 e 1 e w è l'inversa di $w\}$
- **f.** $\{n \mid n$ è un intero e $n = n + 1\}$

0.2 Scrivere una descrizione formale dei seguenti insiemi.

- **a.** L'insieme contenente i numeri 1, 10, e 100
- **b.** L'insieme contenente tutti i numeri interi maggiori di 5
- **c.** L'insieme che contiene tutti i numeri naturali che sono minori di 5
- **d.** L'insieme che contiene la stringa aba
- **e.** L'insieme che contiene la stringa vuota
- **f.** L'insieme che non contiene niente

0.3 Siano A l'insieme $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ e B l'insieme $\{\mathtt{x}, \mathtt{y}\}$.

- **a.** A è un sottoinsieme di B?
- **b.** B è un sottoinsieme di A?
- **c.** Qual é $A \cup B$?
- **d.** Qual é $A \cap B$?
- **e.** Qual é $A \times B$?
- **f.** Qual é l'insieme potenza di B?

0.4 Se A ha a elementi e B ha b elementi, quanti elementi contiene $A \times B$? Motivare la risposta.

0.5 Se C ha c elementi, quanti elementi contiene l'insieme potenza di C? Motivare la risposta.

0.6 Siano X l'insieme $\{1, 2, 3, 4, 5\}$ e Y l'insieme $\{6, 7, 8, 9, 10\}$. Le tabelle seguenti definiscono la funzione unaria $f: X \longrightarrow Y$ e la funzione binaria $g: X \times Y \longrightarrow Y$

n	$f(n)$
1	6
2	7
3	6
4	7
5	6

g	6	7	8	9	10
1	10	10	10	10	10
2	7	8	9	10	6
3	7	7	8	8	9
4	9	8	7	6	10
5	6	6	6	6	6

a. Qual è il valore di $f(2)$?

b. Quali sono il dominio ed il codominio di f?

c. Qual è il valore di $g(2, 10)$?

d. Quali sono il dominio ed il codominio di g?

e. Qual è il valore di $g(4, f(4))$?

0.7 Per ognuno dei seguenti punti, date una relazione che soddisfi la condizione.

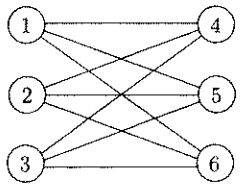
a. Riflessiva e simmetrica, ma non transitiva

b. Riflessiva e transitiva, ma non simmetrica

c. Simmetrica e transitiva, ma non riflessiva

0.8 Si consideri il grafo orientato $G = (V, E)$ dove V, l'insieme dei vertici, è $\{1, 2, 3, 4\}$ ed E, l'insieme degli archi è $\{\{1, 2\}, \{2, 3\}, \{1, 3\}, \{2, 4\}, \{1, 4\}\}$. Disegnare il grafo G. Qual è il grado del nodo 1? Qual è il grado del nodo 3? Indicare un cammino dal nodo 3 al nodo 4 sul disegno di G.

0.9 Scrivere una descrizione formale del seguente grafo.



PROBLEMI

0.10 Mostrare che ogni grafo avente due o più nodi contiene due nodi aventi lo stesso grado.

0.11 Trovare l'errore nella seguente dimostrazione che tutti i cavalli sono dello stesso colore.
AFFERMAZIONE: In un qualsiasi insieme di h cavalli, tutti i cavalli sono dello stesso colore
DIMOSTRAZIONE: Per induzione su h.

Base: Per $h = 1$. In ogni insieme contenente un solo cavallo, tutti i cavalli sono chiaramente dello stesso colore.

Passo induttivo: Per $k \geq 1$, supponiamo che l'affermazione sia vera per $h = k$ e dimostriamola vera per $h = k + 1$. Prendiamo qualsiasi insieme H di $k + 1$ cavalli. Mostriamo che tutti i cavalli di questo insieme sono dello stesso colore. Rimuoviamo un cavallo da questo insieme per ottenere l'insieme H_1 contenente solo k cavalli. Per ipotesi induttiva, tutti i cavalli in H_1 sono dello stesso colore. Ora sostituiamo il cavallo rimosso e rimuoviamone un altro in modo da ottenere l'insieme H_2. Per lo stesso motivo, tutti i cavalli in H_2 sono dello stesso colore. Pertanto tutti i cavalli in H devono essere dello stesso colore, e la dimostrazione è completa.

0.12 Sia $S(n) = 1 + 2 + \cdots + n$ la somma dei primi n numeri naturali e sia $C(n) = 1^3 + 2^3 + \cdots + n^3$ la somma dei primi n cubi. Dimostrare le seguenti uguaglianze per induzione su n, per giungere alla curiosa conclusione che $C(n) = S^2(n)$ per ogni n.

a. $S(n) = \frac{1}{2}n(n + 1)$.

b. $C(n) = \frac{1}{4}(n^4 + 2n^3 + n^2) = \frac{1}{4}n^2(n + 1)^2$.

0.13 Trovare l'errore nella seguente dimostrazione che $2 = 1$.
Si consideri l'equazione $a = b$. Moltiplicare entrambi i lati per a per ottenere $a^2 = ab$. Sottrarre b^2 da entrambi i lati per ottenere $a^2 - b^2 = ab - b^2$. Ora fattorizzare ciascun lato, $(a + b)(a - b) = b(a - b)$, e dividere ogni lato per $(a - b)$, per ottenere $a + b = b$. Infine, porre a e b pari a 1, che mostra che $2 = 1$.

^A**0.14** Usare il Teorema 0.25 per ottenere una formula per il calcolo della rata mensile per un mutuo in termini di un valore iniziale P, un tasso di interesse I ed il numero t di pagamenti. Si supponga che dopo t pagamenti, l'importo del prestito si è ridotto a 0. Utilizzare la formula per calcolare l'importo dell'ammontare in dollari di ogni rata mensile per un mutuo di 30 anni con 360 rate mensili su un importo iniziale di \$100000 con un tasso di interesse annuo del 5% .

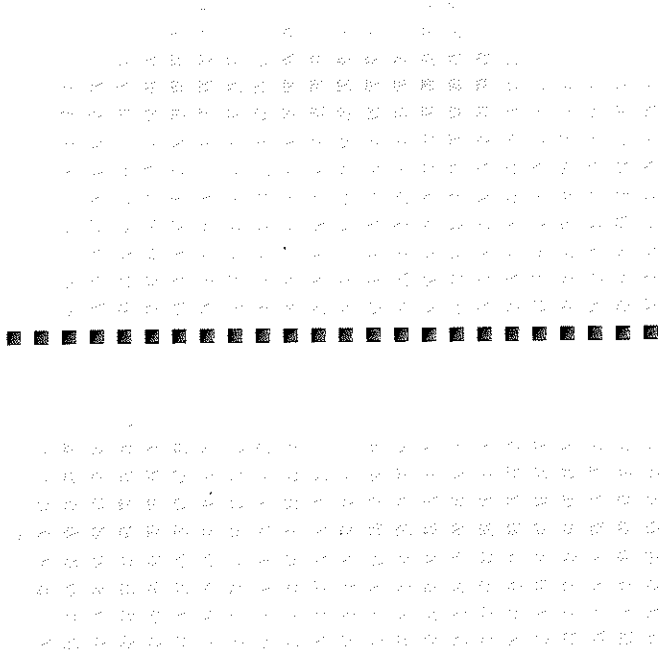
^A*^**0.15** **Teorema di Ramsey.** Sia G un grafo. Una ***clique*** in G è un sottografo in cui ogni coppia di nodi è connessa da un arco. Un'***anti-clique***, anche chiamata ***insieme indipendente***, è un sottografo in cui ogni coppia di nodi non sono connessi da un arco. Mostrare che ogni grafo con n nodi contiene una clique oppure un'anti-clique di $\frac{1}{2}\log_2 n$ nodi.

SOLUZIONI SELEZIONATE

0.14 Poniamo $P_t = 0$ e risolviamo rispetto ad Y per ottenere la formula: $Y = PM^t(M - 1)/(M^t - 1)$. Per $P = \$100{,}000$, $I = 0.05$ e $t = 360$, abbiamo $M = 1 + (0.05)/12$. Usiamo una calcolatrice per scoprire che $Y \approx \$536.82$ è la rata mensile.

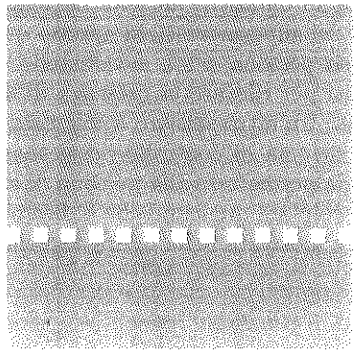
0.15 Fare spazio per due mucchi di nodi: A e B. Poi, a partire dall'intero grafo, aggiungere ripetutamente ogni nodo rimanente x ad A se il suo grado è maggiore della metà del numero di nodi rimasti ed a B altrimenti, e scartare tutti i nodi a cui x non è (è) collegato se esso è stato inserito in A (B). Continuare fino a quando non resta alcun nodo. Ad ogni passo viene scartata al più la metà dei nodi, quindi saranno necessari almeno $\log_2 n$ passi prima che il processo termini. Ogni passaggio aggiunge un nodo ad uno dei mucchi, così uno dei mucchi conterrà alla fine almeno $\frac{1}{2}\log_2 n$ nodi. Il mucchio A contiene i nodi di una clique ed il mucchio B contiene i nodi di un anti-clique.

PARTE PRIMA



AUTOMI E LINGUAGGI

1



LINGUAGGI REGOLARI

La teoria della computazione inizia con una domanda: Cos'è un computer? Forse è una domanda sciocca, poiché tutti sanno che questo oggetto su cui scrivo è un computer. Ma i computer reali sono piuttosto complicati – troppo per permetterci di sviluppare una teoria matematica direttamente su di essi e che sia maneggevole. Invece, usiamo un computer ideale, chiamato ***modello di computazione***. Come con ogni modello nelle scienze, un modello di computazione può essere accurato in alcuni aspetti ma forse non rispetto ad altri. Quindi useremo modelli di computazione diversi, a seconda delle caratteristiche su cui vogliano concentrarci. Iniziamo con il modello più semplice, chiamato ***macchina a stati finiti*** o ***automa finito***.

1.1
AUTOMI FINITI

Gli automi finiti sono un buon modello per computer con una quantità estremamente limitata di memoria. Che cosa può fare un computer con una tale piccola memoria? Molte cose utili! In effetti, noi interagiamo sempre con tali computer, poiché essi sono alla base di vari dispositivi elettromeccanici.

Il sistema di controllo per una porta automatica è un esempio di un tale dispositivo. Le porte automatiche ad anta battente, che si trovano spesso alle entrate e uscite dei supermercati, si aprono quando il sistema di controllo avverte che una persona si sta avvicinando. Una porta automatica ha un sensore davanti per rilevare la presenza di una persona che sta per attraversare la soglia. Un altro sensore è collocato dall'altro lato della soglia in modo che il sistema di controllo possa mantenere aperta la porta abbastanza a lungo da permettere alla persona di attraversarla e anche per evitare che la porta colpisca qualcuno che è dietro di essa quando si apre. Questa disposizione è mostrata nella figura seguente.

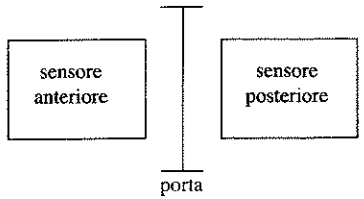


FIGURA 1.1
Vista dall'alto di una porta automatica

Il sistema di controllo è in uno dei due stati: "OPEN" o "CLOSED," che rappresentano la condizione corrispondente della porta. Come mostrato nelle figure seguenti, ci sono quattro possibili condizioni di input: "FRONT" (il cui significato è che una persona è sul sensore davanti alla soglia), "REAR" (significa che una persona è sul sensore dietro alla soglia), "BOTH" (significa che vi sono persone su entrambi i sensori), e "NEITHER" (significa che nessuno è su alcun sensore).

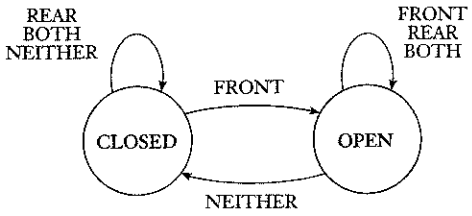


FIGURA 1.2
Diagramma di stato per un sistema di controllo di una porta automatica

		input			
		NEITHER	FRONT	REAR	BOTH
stato	CLOSED	CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

FIGURA 1.3
Tabella delle transizioni di stato per il sistema di controllo di una porta automatica

Il sistema di controllo passa da uno stato a un altro, in base all'input che riceve. Quando è nello stato CLOSED e riceve l'input NEITHER o REAR, resta nello stato CLOSED. Inoltre, se riceve l'input BOTH, resta in CLOSED perché se aprisse la porta rischierebbe di urtare qualcuno sul sensore posteriore. Ma se riceve l'input FRONT, passa nello stato OPEN. Nello stato OPEN, se riceve l'input FRONT, REAR, o BOTH resta in OPEN. Se riceve l'input NEITHER, ritorna a CLOSED.

Per esempio, un sistema di controllo potrebbe iniziare nello stato CLOSED e ricevere la sequenza di segnali di input FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, REAR, e NEITHER. Allora esso passerebbe attraverso la sequenza di stati CLOSED (iniziale), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED, CLOSED e CLOSED.

Vedere un sistema di controllo di una porta automatica come un automa finito è utile perché suggerisce modi standard di rappresentazione come nelle Figure 1.2 e 1.3. Questo sistema di controllo è un computer che ha soltanto un singolo bit di memoria, in grado di memorizzare in quale dei due stati è il sistema di controllo. Altri dispositivi comuni hanno sistemi di controllo con memorie un pò più grandi. In un sistema di controllo di un ascensore, uno stato può rappresentare il piano a cui è l'ascensore e gli input potrebbero essere i segnali ricevuti dai pulsanti. Il computer potrebbe aver bisogno di diversi bit per mantenere traccia di questa informazione. I sistemi di controllo di svariati elettrodomestici come lavastoviglie e termostati elettronici, oltre a orologi e calcolatrici digitali, sono ulteriori esempi di computer con memorie limitate. Il progetto di tali dispositivi richiede di ricordare la metodologia e la terminologia degli automi finiti.

Gli automi finiti e la loro controparte probabilistica, le ***catene di Markov*** sono utili strumenti quando cerchiamo di riconoscere pattern in dati. Questi dispositivi sono usati nell'elaborazione vocale e nel riconoscimento ottico dei caratteri. Le catene di Markov sono state perfino usate per modellare e predire i cambiamenti dei prezzi nei mercati finanziari.

Ora esamineremo più attentamente gli automi finiti da una prospettiva matematica. Svilupperemo una definizione precisa di automa finito, una terminologia per descrivere e usare gli automi finiti, e risultati teorici che descrivano il loro potere e i loro limiti. Oltre a darti una comprensione più chiara di cosa sono gli automi finiti e cosa essi possano e non possano fare,

questo sviluppo teorico ti permetterà di esercitarti e di stare più a proprio agio con definizioni matematiche, teoremi e dimostrazioni in un contesto relativamente semplice.

Inizialmente descriviamo la teoria matematica degli automi finiti in astratto, senza riferirci ad alcuna particolare applicazione. La figura seguente rappresenta un automa finito chiamato M_1.

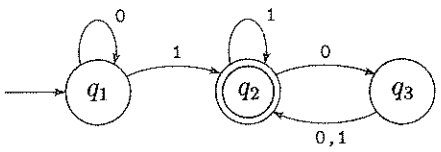


FIGURA 1.4
Un automa finito, chiamato M_1, che ha tre stati

La Figura 1.4 è chiamata il ***diagramma di stato*** di M_1. Esso ha tre ***stati***, etichettati q_1, q_2 e q_3. Lo ***stato iniziale***, q_1, è indicato dall'arco entrante in esso e non uscente da uno stato. Lo ***stato accettante***, q_2, è quello con un doppio cerchio. Gli archi che vanno da uno stato a un altro sono chiamati ***transizioni***.

Quando l'automa riceve una stringa in input come 1101, esso elabora questa stringa e produce un output. L'output è ***accetta*** o ***rifiuta***. Per semplicità, considereremo per ora soltanto questo tipo di output sì/no. L'elaborazione inizia nello stato iniziale di M_1. L'automa riceve i simboli della stringa di input uno a uno da sinistra a destra. Dopo aver letto ciascun simbolo, M_1 si muove da uno stato a un altro lungo la transizione che ha quel simbolo come etichetta. Quando legge l'ultimo simbolo, M_1 produce il suo output. L'output è *accetta* se M_1 è, in quel momento, in uno stato accettante e *rifiuta* se non lo è.

Per esempio, quando forniamo la stringa di input 1101 alla macchina M_1 nella Figura 1.4, l'elaborazione procede come segue:

1. Inizia nello stato q_1.
2. Legge 1, effettua la transizione da q_1 a q_2.
3. Legge 1, effettua la transizione da q_2 a q_2.
4. Legge 0, effettua la transizione da q_2 a q_3.
5. Legge 1, effettua la transizione da q_3 a q_2.
6. *Accetta* perché M_1 è in uno stato accettante q_2 alla fine dell'input.

I risultati di test, con varie stringhe di input, su questa macchina, mostrano che essa accetta le stringhe 1, 01, 11, e 0101010101. In effetti, M_1 accetta una qualsiasi stringa che termina con un 1, poiché va nel suo stato accettante q_2 ogni volta che legge il simbolo 1. Inoltre, accetta le stringhe 100, 0100, 110000, e 0101000000, e una qualsiasi stringa che termina con un numero pari di 0 dopo l'ultimo 1. Rifiuta altre stringhe, come 0, 10, 101000. Sei in grado di descrivere il linguaggio che consiste di tutte le stringhe che M_1 accetta? Lo faremo a breve.

Definizione formale di automa finito

Nella sezione precedente, abbiamo usato i diagrammi di stato per introdurre gli automi finiti. Ora definiremo formalmente gli automi finiti. Sebbene i diagrammi di stato siano intuitivamente più facili da capire, abbiamo bisogno anche della definizione formale per due ragioni specifiche.

Innanzitutto, una definizione formale è precisa. Essa risolve qualsiasi incertezza su cosa è permesso in un automa finito. Se fossi incerto sulla possibilità di avere 0 stati di accettazione in un automa finito o se esso deve avere esattamente una transizione uscente da ogni stato e per ogni possibile simbolo di input, potresti consultare la definizione formale e verificare che la risposta è sì in entrambi i casi. In secondo luogo, una definizione formale fornisce notazioni. Una buona notazione ti aiuta a pensare e a esprimere chiaramente i tuoi pensieri.

Il linguaggio di una definizione formale è piuttosto arcano e ha qualche somiglianza con il linguaggio di un documento giuridico. Entrambi devono essere precisi e ogni dettaglio deve essere espresso chiaramente.

Un automa finito ha diverse componenti. Esso ha un insieme di stati e regole per andare da uno stato a un altro, a seconda del simbolo di input. Ha un alfabeto di simboli input che indica i simboli di input permessi. Ha uno stato iniziale e un insieme di stati accettanti. La definizione formale afferma che un automa finito è una lista di questi cinque oggetti: insieme degli stati, alfabeto di simboli di input, regole per i cambiamenti di stato, stato iniziale e stati accettanti. Nel linguaggio matematico, una lista di cinque elementi è spesso chiamata una quintupla. Quindi definiamo un automa finito come una quintupla che consiste di queste cinque componenti.

Usiamo qualcosa il cui nome è ***funzione di transizione***, frequentemente denotata con δ, per definire le regole per il cambiamento di stato. Se l'automa finito ha un arco da uno stato x a uno stato y etichettato con il simbolo di input 1, questo significa che se l'automa è nello stato x quando legge un 1, allora si muove nello stato y. Possiamo indicare la stessa cosa con la funzione di transizione dicendo che $\delta(x, 1) = y$. Questa notazione è una specie di abbreviazione matematica. Mettendo tutto questo insieme, arriviamo alla definizione formale di automa finito.

DEFINIZIONE **1.5**

Un ***automa finito*** è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove

1. Q è un insieme finito chiamato l'insieme degli ***stati***,
2. Σ è un insieme finito chiamato l'***alfabeto***,
3. $\delta\colon Q \times \Sigma \longrightarrow Q$ è la ***funzione di transizione***,[1]
4. $q_0 \in Q$ è lo ***stato iniziale***, ed
5. $F \subseteq Q$ è l'***insieme degli stati accettanti***.[2]

La definizione formale descrive precisamente cosa intendiamo per automa finito. Per esempio, tornando alla domanda precedente, se sia consentito avere 0 stati accettanti, puoi vedere che porre F uguale all'insieme vuoto $\emptyset$ implica 0 stati accettanti, che è quindi consentito. Inoltre, la funzione di transizione δ specifica esattamente uno stato successivo per ogni possibile combinazione di uno stato e un simbolo di input. Questo risponde affermativamente all'altra nostra domanda, mostrando che da ogni stato esce esattamente un arco di transizione per ogni possibile simbolo di input.

Possiamo usare la notazione della definizione formale per descrivere automi finiti particolari, specificando ciascuna delle cinque componenti elencate nella Definizione 1.5. Per esempio, torniamo all'automa finito M_1 che abbiamo discusso prima, ridisegnato qui per comodità.

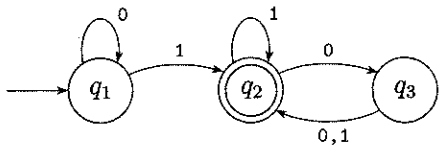


FIGURA 1.6
L'automa finito M_1

Possiamo descrivere M_1 formalmente ponendo $M_1 = (Q, \Sigma, \delta, q_1, F)$, dove

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,

[1] Fai riferimento indietro alla pagina 8 se sei incerto sul significato di $\delta\colon Q \times \Sigma \longrightarrow Q$.
[2] Gli stati accettanti a volte sono chiamati ***stati finali***.

3. δ è descritta come segue

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

4. q_1 è lo stato iniziale ed
5. $F = \{q_2\}$.

Se A è l'insieme di tutte le stringhe che la macchina M accetta, diciamo che A è il ***linguaggio della macchina M*** e scriviamo $L(M) = A$. Diciamo che ***M riconosce A*** o che ***M accetta A***. Poiché il termine *accetta* ha significati diversi quando ci riferiamo a macchine che accettano stringhe e macchine che accettano linguaggi, per evitare confusione preferiamo per i linguaggi il termine *riconosce*.

Una macchina può accettare diverse stringhe, ma riconosce sempre soltanto un linguaggio. Se la macchina non accetta alcuna stringa, essa riconosce ancora un linguaggio – ossia, il linguaggio vuoto $\emptyset$.

Nel nostro esempio, sia

$$A = \{w \mid w \text{ contiene almeno un } 1 \text{ e un numero pari di } 0 \text{ segue l'ultimo } 1\}.$$

Allora $L(M_1) = A$, o equivalentemente, M_1 riconosce A.

Esempi di automi finiti

ESEMPIO 1.7

Ecco il diagramma di stato di un automa finito M_2.

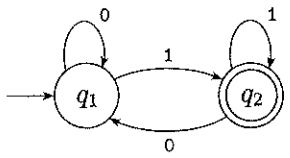


FIGURA 1.8
Diagramma di stato dell'automa finito M_2 con due stati

Nella descrizione formale, M_2 è $(\{q_1, q_2\}, \{0,1\}, \delta, q_1, \{q_2\})$. La funzione di transizione δ è

	0	1
q_1	q_1	q_2
q_2	q_1	q_2.

Ricorda che il diagramma di stato di M_2 e la descrizione formale di M_2 contengono le stesse informazioni, solo in forme differenti. Puoi sempre andare dall'una all'altra se necessario.

Un buon modo per iniziare a comprendere una qualsiasi macchina è provarne il comportamento su alcune stringhe di input. Quando fai questi "esperimenti" per vedere come opera la macchina, il suo modo di funzionare spesso diventa chiaro. Sulla stringa campione 1101, la macchina M_2 inizia nel suo stato iniziale q_1 e va prima nello stato q_2 dopo aver letto il primo 1, e poi negli stati q_2, q_1 e q_2 dopo aver letto 1, 0 e 1. La stringa è accettata perché q_2 è uno stato accettante. Ma la stringa 110 lascia M_2 nello stato q_1, quindi essa è rifiutata. Dopo aver provato alcuni altri esempi, dovresti aver capito che M_2 accetta tutte le stringhe che terminano con un 1. Quindi $L(M_2) = \{w | \ w \text{ termina con un } 1\}$.

ESEMPIO 1.9

Considera l'automa finito M_3.

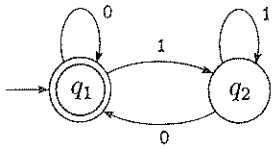


FIGURA 1.10
Diagramma di stato dell'automa finito M_3 con due stati

La macchina M_3 è simile a M_2 tranne nella posizione dello stato accettante. Come al solito, la macchina accetta tutte le stringhe che la lasciano in uno stato accettante quando ne ha finito la lettura. Nota che, poiché lo stato iniziale è anche uno stato accettante, M_3 accetta la stringa vuota ε. Non appena una macchina inizia a leggere la stringa vuota, ne ha finito la lettura; quindi se lo stato iniziale è uno stato accettante, ε è accettata. Oltre alla stringa vuota, questa macchina accetta ogni stringa che termina con 0. In questo caso,

$$L(M_3) = \{w | \ w \text{ è la stringa vuota } \varepsilon \text{ o termina con } 0\}.$$

ESEMPIO 1.11

La figura seguente mostra una macchina M_4 con cinque stati.

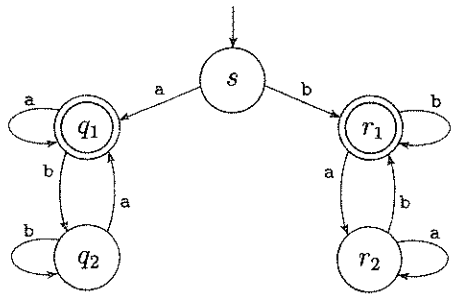


FIGURA 1.12
L'automa finito M_4

La macchina M_4 ha due stati accettanti, q_1 ed r_1, e opera sull'alfabeto $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Alcuni tentativi mostrano che essa accetta le stringhe a, b, aa, bb, e bab, ma non le stringhe ab, ba, or bbba. Questa macchina inizia nello stato s e dopo aver letto il primo simbolo in input, va a sinistra negli stati q o a destra negli stati r. In entrambi i casi, non può mai ritornare nello stato iniziale (a differenza degli esempi precedenti), poiché non ha alcun modo per ritornare a s da un qualsiasi altro stato. Se il primo simbolo nella stringa di input è a, allora va a sinistra e accetta quando la stringa termina con a. Analogamente, se il primo simbolo è b, la macchina va a destra e accetta quando la stringa termina con b. Quindi M_4 accetta tutte le stringhe che iniziano e terminano con a o che iniziano e terminano con b. In altre parole, M_4 accetta le stringhe che iniziano e terminano con lo stesso simbolo.

ESEMPIO 1.13

La Figura 1.14 mostra la macchina M_5 con tre stati e alfabeto di simboli di input con quattro simboli, $\Sigma = \{\langle\text{RESET}\rangle, 0, 1, 2\}$. Trattiamo $\langle\text{RESET}\rangle$ come un singolo simbolo.

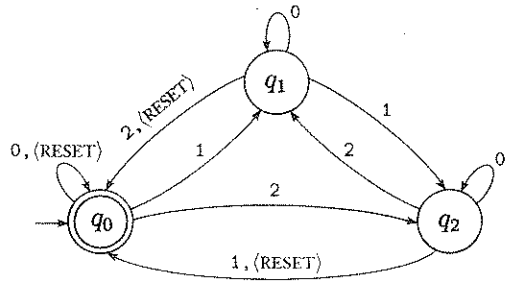


FIGURA 1.14
L'automa finito M_5

La macchina M_5 conserva un calcolo parziale della somma dei simboli di input numerici che legge, modulo 3. Ogni volta che riceve il simbolo ⟨RESET⟩, riporta il conto a 0. Accetta se la somma è 0 modulo 3, o in altre parole, se la somma è un multiplo di 3.

Descrivere un automa finito mediante il diagramma di stato non è possibile in alcuni casi. Questo può accadere quando il diagramma sarebbe troppo grande da disegnare o, come nell'esempio successivo, se la descrizione dipende da alcuni parametri non specificati. In questi casi, ci serviamo della descrizione formale per specificare la macchina.

ESEMPIO 1.15

Consideriamo una generalizzazione dell'Esempio 1.13, che usa lo stesso alfabeto a quattro simboli Σ. Per ogni $i \geq 1$ sia A_i il linguaggio di tutte le stringhe tali che la somma dei numeri è un multiplo di i, tranne che la somma è riportata a 0 ogni volta che appare il simbolo ⟨RESET⟩. Per ogni A_i definiamo un automa finito B_i, che riconosce A_i. Descriviamo la macchina B_i formalmente come segue: $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, dove Q_i è l'insieme degli i stati $\{q_0, q_1, q_2, \ldots, q_{i-1}\}$, e definiamo la funzione di transizione δ_i in modo che per ogni j, se B_i è in q_j, la somma parziale sia j, modulo i. Per ogni q_j sia

$$\delta_i(q_j, 0) = q_j,$$
$$\delta_i(q_j, 1) = q_k, \text{dove } k = j + 1 \text{ modulo } i,$$
$$\delta_i(q_j, 2) = q_k, \text{dove } k = j + 2 \text{ modulo } i, \text{ e}$$
$$\delta_i(q_j, \langle\text{RESET}\rangle) = q_0.$$

Definizione formale di computazione

Finora abbiamo descritto gli automi finiti informalmente, usando diagrammi di stato, e con una definizione formale, come una quintupla. La descrizione informale è più facile da comprendere all'inizio, ma la definizione formale è utile per rendere la nozione precisa, risolvendo ogni ambiguità che può essere presente nella descrizione informale. Di seguito, facciamo lo stesso per la computazione di un automa finito. Abbiamo già un'idea informale del modo in cui esso computa, e ora lo formalizziamo matematicamente.

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un automa finito e sia $w = w_1 w_2 \cdots w_n$ una stringa dove ogni w_i è un elemento dell'alfabeto Σ. Allora M ***accetta*** w se esiste una sequenza di stati $r_0, r_1, \ldots, r_n$ in Q con tre condizioni:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, per $i = 0, \ldots, n-1$, e
3. $r_n \in F$.

La condizione 1 afferma che la macchina inizia nello stato iniziale. La condizione 2 afferma che la macchina passa da stato a stato in base alla funzione di transizione. La condizione 3 afferma che la macchina accetta il suo input se termina la lettura in uno stato accettante. Diciamo che M ***riconosce il linguaggio*** A se $A = \{w | M \text{ accetta } w\}$.

DEFINIZIONE 1.16

Un linguaggio è chiamato un ***linguaggio regolare*** se un automa finito lo riconosce.

ESEMPIO 1.17

Considera la macchina M_5 nell'Esempio 1.13. Sia w la stringa

$$10\langle\text{RESET}\rangle22\langle\text{RESET}\rangle012.$$

Allora M_5 accetta w in base alla definizione di computazione perché la sequenza di stati che attraversa quando computa su w è

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

che soddisfa le tre condizioni. Il linguaggio di M_5 è

$$L(M_5) = \{w | \text{ la somma dei simboli in } w \text{ è 0 modulo 3, tranne per il fatto che } \langle\text{RESET}\rangle \text{ riporta la somma a 0}\}.$$

Poiché M_5 riconosce questo linguaggio, esso è un linguaggio regolare.

Progettare automi finiti

La progettazione, che sia di un automa o di un'opera d'arte, è un processo creativo. Come tale, essa non può essere ridotta a una semplice ricetta o formula. Comunque, nel progettare i vari tipi di automi, potresti trovare utile un particolare approccio. Cioè, mettere *te stesso* al posto della macchina che stai provando a progettare e poi vedere come affronteresti l'esecuzione del compito della macchina. Immaginare di essere la macchina è un'abilità mentale che aiuta a coinvolgere l'intera mente nel processo del progetto.

Progettiamo un automa finito usando il metodo "lettore uguale automa" appena descritto. Supponi che ti venga dato un linguaggio e vuoi progettare un automa finito che lo riconosce. Immaginando di essere l'automa, ricevi una stringa in input e devi determinare se è un elemento del linguaggio che l'automa deve riconoscere. Potrai vedere i simboli nella stringa uno a uno. Dopo ciascun simbolo, devi decidere se la stringa vista fino ad allora è nel

linguaggio. Il motivo è che tu, come la macchina, non sai se sei alla fine della stringa, quindi devi essere sempre pronto con la risposta.

Innanzitutto, per prendere queste decisioni, devi capire cosa devi ricordare della stringa mentre la stai leggendo. Perché non ricordare semplicemente tutto quello che hai visto? Considera che tu stai agendo come se fossi un automa finito e che questo tipo di macchina ha solo un numero finito di stati, il che significa una memoria finita. Immagina che l'input sia estremamente lungo – diciamo, da qui alla luna – così che tu non potresti verosimilmente ricordare l'intero oggetto. Tu hai una memoria finita – diciamo, un singolo foglio di carta – che ha una capacità di memorizzazione limitata. Fortunatamente per molti linguaggi non hai bisogno di ricordare tutto l'input. Devi ricordare solo alcune informazioni essenziali. Esattamente quali informazioni sono essenziali dipende dal particolare linguaggio considerato.

Per esempio, supponiamo che l'alfabeto sia $\{0,1\}$ e che il linguaggio consista di tutte le stringhe con un numero dispari di simboli uguali a 1. Vuoi costruire un automa finito E_1 che riconosca questo linguaggio. Immaginando di essere l'automa, inizi a ricevere la stringa input di simboli 0 e 1 un simbolo alla volta. Devi ricordare l'intera stringa vista finora per determinare se il numero dei simboli uguali 1 è dispari? Naturalmente no. Ricorda semplicemente se il numero dei simboli uguali a 1 visti fino ad allora è pari o dispari e mantieni traccia di questa informazione mentre leggi nuovi simboli. Se leggi un 1, cambia la risposta; ma se leggi uno 0, lascia la risposta com'è.

Ma come questo può aiutarti a progettare E_1? Una volta che hai determinato l'informazione necessaria da ricordare circa la stringa mentre la stai leggendo, rappresenti questa informazione come una lista finita di possibilità. In questo caso, le possibilità sarebbero

1. finora pari e
2. finora dispari.

Poi assegni uno stato a ciascuna delle possibilità. Questi sono gli stati di E_1, come mostrato di seguito.

FIGURA 1.18
I due stati q_{even} e q_{odd}

Successivamente, assegni le transizioni vedendo come passare da una possibilità all'altra subito dopo aver letto un simbolo. Quindi, se lo stato q_{even} rappresenta la possibilità pari e lo stato q_{odd} rappresenta la possibilità dispari, potresti definire le transizioni per cambiare lo stato ricevendo un 1 e stare fermo ricevendo uno 0, come mostrato di seguito.

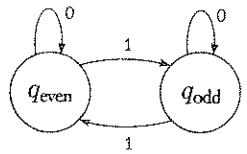


FIGURA 1.19
Le transizioni mostrano come le possibilità si riorganizzano

Poi, stabilisci che lo stato iniziale sia lo stato corrispondente alla possibilità associata con l'aver visto 0 simboli finora (la stringa vuota ε). In questo caso, lo stato iniziale corrisponde allo stato q_{even} poiché 0 è un numero pari. Infine, stabilisci che gli stati accettanti siano quelli corrispondenti alle possibilità in cui vuoi accettare la stringa di input . Stabilisci che q_{odd} sia uno stato accettante poiché vuoi accettare quando hai visto un numero dispari di simboli uguali a 1. Queste integrazioni sono mostrate nella figura seguente.

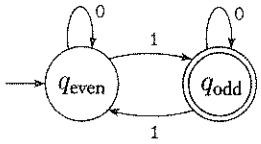


FIGURA 1.20
Aggiunta dello stato iniziale e degli stati accettanti

ESEMPIO 1.21

Questo esempio mostra come progettare un automa finito E_2 per riconoscere il linguaggio regolare di tutte le stringhe che contengono la stringa 001 come sottostringa. Per esempio, 0010, 1001, 001, e 11111110011111 sono tutte nel linguaggio, ma 11 e 0000 non vi appartengono. Come riconosceresti questo linguaggio se tu immaginassi di essere E_2? Quando arrivano i simboli, inizialmente dovresti ignorare tutti i simboli uguali a 1. Se ricevi uno 0, allora osservi che puoi avere appena visto il primo dei tre simboli nel pattern 001 che stai cercando. Se a questo punto vedi un 1, c'erano troppo pochi 0, quindi riprendi a ignorare i simboli uguali a 1. Ma se a questo punto vedi uno 0, dovresti ricordare che hai appena visto due sim-

boli del pattern. Ora devi semplicemente continuare a scorrere l'input fino a quando vedi un 1. Se lo trovi, ricorda che sei riuscito a trovare il pattern e continua a leggere la stringa di input fino alla fine.

Quindi ci sono quattro possibilità:

1. non hai visto alcun simbolo iniziale del pattern,
2. hai appena visto uno 0,
3. hai appena visto 00, o
4. hai appena visto l'intero pattern 001.

Assegna gli stati q, q_0, q_{00} e q_{001} a queste possibilità. Puoi assegnare le transizioni osservando che da q leggendo un 1 resti in q, ma leggendo uno 0 passi in q_0. In q_0 leggendo 1 ritorni in q, ma leggendo uno 0 passi in q_{00}. In q_{00} leggendo un 1 passi in q_{001}, ma leggendo uno 0 resti in q_{00}. Infine, in q_{001} leggendo uno 0 o un 1 resti in q_{001}. Lo stato iniziale è q, e il solo stato accettante è q_{001}, come mostrato nella Figura 1.22.

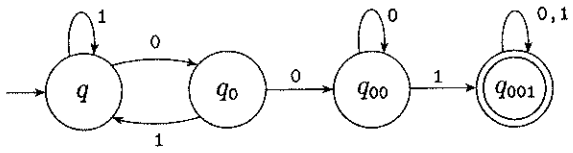


FIGURA 1.22
Accetta le stringhe contenenti 001

Le operazioni regolari

Nelle precedenti due sezioni, abbiamo introdotto e definito gli automi finiti e i linguaggi regolari. Ora iniziamo a studiare le loro proprietà. Questo aiuterà a sviluppare una serie di tecniche per progettare automi che riconoscono particolari linguaggi. La serie includerà anche modi di provare che alcuni altri linguaggi non sono regolari (cioè, oltre la capacità degli automi finiti).

In aritmetica, gli oggetti di base sono i numeri e gli strumenti sono le operazioni per trattarli, come + e ×. Nella teoria della computazione, gli oggetti sono i linguaggi e gli strumenti includono operazioni specificamente progettate per trattarli. Definiamo tre operazioni sui linguaggi, chiamate ***operazioni regolari***, e le usiamo per studiare le proprietà dei linguaggi regolari.

DEFINIZIONE **1.23**

Siano A e B linguaggi. Definiamo le operazioni regolari ***unione***, ***concatenazione*** e ***star*** come segue:

- **Unione**: $A \cup B = \{x | \, x \in A \text{ o } x \in B\}$.
- **Concatenazione**: $A \circ B = \{xy | \, x \in A \text{ e } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \ldots x_k | \, k \geq 0 \text{ e ogni } x_i \in A\}$.

Tu hai già familiarità con l'operazione di unione. Essa semplicemente prende tutte le stringhe sia in A che in B e le raggruppa insieme in un linguaggio.

L'operazione di concatenazione è un pó più complicata. Essa antepone una stringa di A ad una stringa di B in tutti i modi possibili per ottenere le stringhe nel nuovo linguaggio.

L'operazione star è un pó differente dalle altre due poichè si applica a un singolo linguaggio piuttosto che a due differenti linguaggi. Cioè, l'operazione star è un'***operazione unaria*** invece che un'***operazione binaria***. Essa opera concatenando un numero qualsiasi di stringhe in A insieme per ottenere una stringa nel nuovo linguaggio. Poiché "un numero qualsiasi" include 0 come possibilità, la stringa vuota ε è sempre un elemento di A^*, indipendentemente da chi sia A.

ESEMPIO 1.24

Sia Σ l'alfabeto standard a 26 lettere $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. Se $A = \{\mathtt{good}, \mathtt{bad}\}$ e $B = \{\mathtt{boy}, \mathtt{girl}\}$, allora

$A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\}$,

$A \circ B = \{\mathtt{goodboy}, \mathtt{goodgirl}, \mathtt{badboy}, \mathtt{badgirl}\}$, e

$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$.

Sia $\mathcal{N} = \{1, 2, 3, \ldots\}$ l'insieme dei numeri naturali. Quando diciamo che $\mathcal{N}$ è *chiuso rispetto alla moltiplicazione*, intendiamo che per ogni x e y in $\mathcal{N}$, il prodotto $x \times y$ è ancora in $\mathcal{N}$. Invece $\mathcal{N}$ non è chiuso rispetto alla divisione, poiché 1 e 2 sono in $\mathcal{N}$ ma $1/2$ non lo è. In generale, una classe di oggetti è ***chiusa*** rispetto a un'operazione se l'applicazione di questa operazione a elementi della classe restituisce un oggetto ancora nella classe. Mostriamo che la classe dei linguaggi regolari è chiusa rispetto a tutte e tre le operazioni regolari. Nella Sezione 1.3, mostriamo che ci sono strumenti

utili per trattare i linguaggi regolari e capire la potenza degli automi finiti. Iniziamo con l'operazione unione.

TEOREMA 1.25

La classe dei linguaggi regolari è chiusa rispetto all'operazione di unione.

In altre parole, se A_1 e A_2 sono linguaggi regolari, lo è anche $A_1 \cup A_2$.

IDEA. Abbiamo due linguaggi regolari, A_1 e A_2, e vogliamo mostrare che anche $A_1 \cup A_2$ è regolare. Poiché A_1 e A_2 sono regolari, sappiamo che un automa finito M_1 riconosce A_1 e un automa finito M_2 riconosce A_2. Per provare che $A_1 \cup A_2$ è regolare, esibiamo un automa finito, M, che riconosce $A_1 \cup A_2$.

Questa è una prova costruttiva. Costruiamo M da M_1 ed M_2. Per riconoscere il linguaggio unione, la macchina M deve accettare il suo input esattamente quando M_1 o M_2 lo accetterebbero. Opera *simulando* sia M_1 che M_2 e accettando se l'una o l'altra delle simulazioni accetta.

Come possiamo far sí che la macchina M simuli M_1 ed M_2? Magari prima simulando M_1 sull'input e poi simulando M_2 sull'input. Ma qui dobbiamo fare attenzione! Una volta che i simboli dell'input sono stati letti e usati per simulare M_1, noi non possiamo "riavvolgere il nastro di input" per provare la simulazione su M_2. Abbiamo bisogno di un'altra strategia.

Immagina che tu sia M. Come ricevi i simboli input, uno a uno, tu simuli sia M_1 che M_2 contemporaneamente. In questo modo, è necessario scorrere l'input solo una volta. Ma come puoi tenere traccia di entrambe le simulazioni con una memoria finita? Tutto quello che hai bisogno di ricordare è lo stato in cui ciascuna macchina sarebbe se avesse letto l'input fino a quel punto. Quindi, devi ricordare una coppia di stati. Quante possibili coppie ci sono? Se M_1 ha k_1 stati ed M_2 ha k_2 stati, il numero delle coppie di stati, uno in M_1 e l'altro in M_2, è il prodotto $k_1 \times k_2$. Questo prodotto sarà il numero degli stati in M, uno per ogni coppia. Le transizioni di M vanno da coppia a coppia, aggiornando lo stato corrente sia per la componente in M_1 che per quella in M_2. Gli stati accettanti di M sono quelle coppie tali che M_1 o M_2 è in uno stato accettante.

DIMOSTRAZIONE

Supponiamo che M_1 riconosca A_1, dove $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, ed
M_2 riconosca A_2, dove $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Costruiamo M che riconosce $A_1 \cup A_2$, dove $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) | \, r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
   Questo insieme è il ***prodotto cartesiano*** degli insiemi Q_1 e Q_2 ed è denotato con $Q_1 \times Q_2$. È l'insieme di tutte le coppie di stati, il primo in Q_1 e il secondo in Q_2.
2. Σ, l'alfabeto, è lo stesso di M_1 ed M_2. In questo teorema e in tutti i successivi teoremi simili, assumiamo per semplicità che sia M_1 che M_2 abbiano lo stesso alfabeto di simboli di input Σ. Il teorema resta vero se hanno alfabeti diversi, Σ_1 e Σ_2. Dopo dovremmo modificare la prova ponendo $\Sigma = \Sigma_1 \cup \Sigma_2$.
3. δ, la funzione di transizione, è definita come segue. Per ogni $(r_1, r_2) \in Q$ e ogni $a \in \Sigma$, sia
   $$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$
   Quindi δ riceve uno stato di M (che in realtà è una coppia di stati, uno in M_1 e uno in M_2), con un simbolo di input, e restituisce lo stato successivo di M.
4. q_0 è la coppia (q_1, q_2).
5. F è l'insieme delle coppie in cui l'uno o l'altro elemento è uno stato accettante di M_1 o M_2. Possiamo scriverlo come
   $$F = \{(r_1, r_2) | \, r_1 \in F_1 \text{ o } r_2 \in F_2\}.$$
   Questa espressione è uguale a $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Nota che *non* è uguale a $F = F_1 \times F_2$. Cosa ci darebbe invece questo?[3])

Questo conclude la costruzione dell'automa finito M che riconosce l'unione di A_1 e A_2. Questa costruzione è piuttosto semplice, e quindi la sua correttezza è evidente dalla strategia descritta nell'idea della prova. Costruzioni più complicate richiedono un'ulteriore discussione per provare la correttezza. Una prova formale di correttezza per una costruzione di questo tipo di solito procede per induzione. Per un esempio di una costruzione di cui si prova la correttezza, vedi la prova del Teorema 1.54. La maggior parte delle costruzioni che incontrerai in questo corso sono piuttosto semplici e quindi non richiedono una prova formale di correttezza.

Abbiamo appena mostrato che l'unione di due linguaggi regolari è regolare, provando in tal modo che la classe dei linguaggi regolari è chiusa rispetto all'operazione di unione. Ora rivolgiamo l'attenzione all'operazione di concatenazione e tentiamo di mostrare che la classe dei linguaggi regolari è chiusa anche rispetto a questa operazione.

[3] Quest'espressione definirebbe gli stati accettanti di M come quelli per i quali *entrambi* gli elementi della coppia sono stati accettanti. In questo caso, M accetterebbe una stringa solo se M_1 *ed* M_2 l'accettano, quindi il linguaggio corrispondente sarebbe l'*intersezione* e non l'unione. In effetti, questo risultato mostra che la classe dei linguaggi regolari è chiusa rispetto all'intersezione.

TEOREMA 1.26

La classe dei linguaggi regolari è chiusa rispetto all'operazione di concatenazione.

In altre parole, se A_1 e A_2 sono linguaggi regolari, allora lo è anche $A_1 \circ A_2$.

Per provare questo teorema, proviamo qualcosa di simile alla prova del caso dell'unione. Come prima, possiamo iniziare con gli automi finiti M_1 ed M_2 che riconoscono i linguaggi regolari A_1 e A_2. Ma ora, invece di costruire l'automa M che accetta il suo input se M_1 o M_2 lo accetta, esso deve accettare se il suo input può essere diviso in due parti, tali che M_1 accetta la prima parte ed M_2 accetta la seconda parte. Il problema è che M non sa dove dividere il suo input (cioè, dove finisce la prima parte e inizia la seconda). Per risolvere questo problema, introduciamo una nuova tecnica chiamata non determinismo.

1.2
NON DETERMINISMO

Il non determinismo è un concetto utile che ha avuto un grande impatto sulla teoria della computazione. Finora nella nostra discussione, ogni passo di una computazione seguiva univocamente dal passo precedente. Quando la macchina è in un dato stato e legge il simbolo di input successivo, sappiamo qual è lo stato successivo – esso è univocamente determinato. In questo caso parliamo di computazione ***deterministica***. In una macchina ***non deterministica***, possono esistere diverse scelte per lo stato successivo in ogni punto.

Il non determinismo è una generalizzazione del determinismo, quindi ogni automa finito deterministico è automaticamente un automa finito non deterministico. Come mostra la Figura 1.27, gli automi finiti non deterministici possono avere ulteriori caratteristiche.

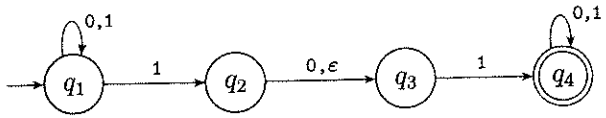


FIGURA 1.27
L'automa finito non deterministico N_1

La differenza tra un automa finito deterministico, indicato con l'acronimo DFA, e un automa finito non deterministico, indicato con l'acronimo NFA, è subito evidente. In primo luogo, ogni stato di un DFA ha sempre esattamente un arco di transizione uscente per ogni simbolo nell'alfabeto. L'NFA mostrato nella Figura 1.27 viola questa regola. Lo stato q_1 ha un arco uscente per 0, ma ne ha due per 1; q_2 ha un arco per 0, ma non ne ha alcuno per 1. In un NFA, uno stato può avere zero, uno, o più archi uscenti per ogni simbolo dell'alfabeto.

Inoltre, in un DFA, le etichette sugli archi di transizione sono simboli dell'alfabeto. Questo NFA ha un arco con etichetta ε. In generale, un NFA può avere archi etichettati con elementi dell'alfabeto o ε. Zero, uno, o più archi possono uscire da ciascuno stato con l'etichetta ε.

Come computa un NFA? Supponiamo di eseguire un NFA su una stringa di input e di giungere in uno stato con più modi per procedere. Per esempio, supponiamo di essere nello stato q_1 dell'NFA N_1 e che il simbolo di input successivo sia un 1. Dopo aver letto questo simbolo, la macchina si divide in più copie di sé stessa e segue *tutte* le possibilità in parallelo. Ogni copia della macchina prende una delle possibili direzioni per procedere e continua come prima. Se ci sono scelte successive, la macchina si divide di nuovo. Se il simbolo di input successivo non compare su alcuno degli archi uscenti dallo stato occupato da una copia della macchina, quella copia della macchina "muore", insieme con il ramo della computazione a essa associato. Infine, se *una qualunque* di queste copie della macchina è in uno stato accettante alla fine dell'input, l'NFA accetta la stringa di input .

Se ci imbattiamo in uno stato con un simbolo ε su un arco uscente, accade qualcosa di simile. Senza leggere alcun input, la macchina si divide in più copie multiple, una che segue ciascun arco uscente etichettato con ε e una che resta nello stato corrente. Poi la macchina procede non deterministicamente come prima.

Il non determinismo può essere visto come una specie di computazione parallela dove "processi" multipli indipendenti o "threads" possono essere eseguiti contemporaneamente. Quando l'NFA si divide per seguire diverse scelte, questo corrisponde a un processo di "separazione" in diversi figli, ognuno dei quali procede per conto proprio. Se almeno uno di questi processi accetta, allora l'intera computazione accetta.

Un altro modo per considerare una computazione non deterministica è vederla come un albero di possibilità. La radice dell'albero corrisponde all'inizio della computazione. Ogni punto di ramificazione nell'albero corrisponde a un punto nella computazione in cui la macchina ha scelte multiple. La macchina accetta se almeno uno dei rami della computazione termina in uno stato accettante, come mostrato nella Figura 1.28.

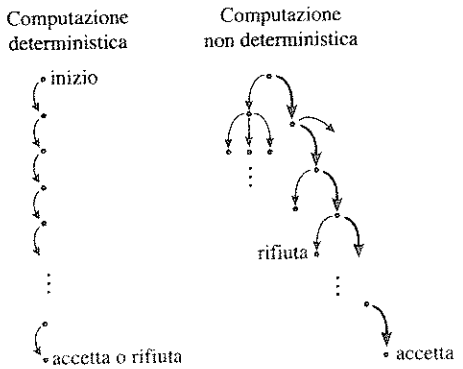


FIGURA 1.28
Computazioni deterministica e non deterministica con un ramo accettante

Consideriamo alcuni esempi di esecuzione dell'NFA N_1 mostrato in Figura 1.27. La computazione di N_1 sull'input 010110 è rappresentata nella figura seguente.

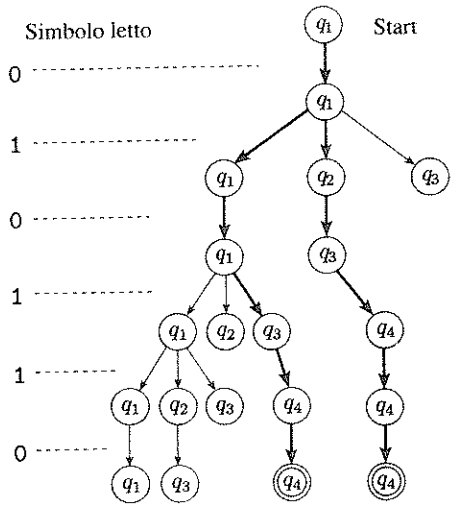


FIGURA 1.29
La computazione di N_1 sull'input 010110

Sull'input 010110, iniziamo nello stato iniziale q_1 e leggiamo il primo simbolo 0. Da q_1 c'è un solo posto dove andare su uno 0 – ossia, indietro su q_1 – quindi restiamo là. Poi, leggiamo il secondo simbolo 1. In q_1 su un 1 ci sono due scelte: restare in q_1 o moversi in q_2. La macchina si divide in due per seguire ciascuna scelta, non deterministicamente. Mantieni traccia delle possibilità mettendo un dito su ciascuno stato in cui una macchina potrebbe essere. Quindi tu ora hai dita sugli stati q_1 e q_2. Un ε-arco esce dallo stato q_2 quindi la macchina si divide di nuovo; mantieni un dito su q_2, e muovi l'altro in q_3. Ora hai dita su q_1, q_2 e q_3.

Quando viene letto il terzo simbolo 0, considera ogni dito, uno alla volta. Mantieni il dito su q_1, muovi il dito su q_2 in q_3, e togli il dito che stava su q_3. Quest'ultimo dito non ha archi su 0 da seguire e corrisponde a un processo che semplicemente "muore." A questo punto, tu hai dita sugli stati q_1 e q_3.

Quando è letto il quarto simbolo 1, dividi il dito su q_1 in dita sugli stati q_1 e q_2, poi dividi ulteriormente il dito su q_2 per seguire l'ε-arco entrante in q_3, e muovi il dito che era su q_3 in q_4. Ora hai un dito su ciascuno dei quattro stati.

Quando è letto il quinto simbolo 1, le dita su q_1 e q_3 diventano dita sugli stati q_1, q_2, q_3, e q_4, come hai visto con il quarto simbolo. Il dito sullo stato q_2 viene tolto da esso. Il dito che era su q_4 resta su q_4. Ora tu hai due dita su q_4, quindi togline uno poiché devi solo ricordare che q_4 è un possibile stato in questo punto, non che è possibile per più ragioni.

Quando è letto il sesto e finale simbolo 0, mantieni il dito su q_1, muovi quello su q_2 in q_3, rimuovi quello che era su q_3, e lascia dov'è quello su q_4. Ora sei alla fine della stringa, e accetti se qualche dito è su uno stato accettante. Tu hai dita sugli stati q_1, q_3, e q_4; e poiché q_4 è uno stato accettante, N_1 accetta questa stringa.

Cosa fa N_1 sull'input 010? Inizia con un dito su q_1. Dopo aver letto lo 0, hai ancora un dito solo su q_1; ma dopo l'1 ci sono dita su q_1, q_2, e q_3 (non dimenticare l'ε arco). Dopo il terzo simbolo 0, togli il dito su q_3 da esso, muovi il dito su q_2 in q_3, e lascia il dito su q_1 dov'è. A questo punto sei alla fine dell'input; e siccome nessun dito è su uno stato accettante, N_1 rifiuta questo input.

Continuando a provare in questo modo, capirai che N_1 accetta tutte le stringhe che contengono 101 o 11 come sottostringa.

Gli automi finiti non deterministici sono utili da vari punti di vista. Come mostreremo, ogni NFA può essere trasformato in un DFAequivalente, e costruire un NFA è a volte più facile che costruire direttamente un DFA. Un NFA può essere molto più piccolo della sua controparte deterministica, o il suo funzionamento può essere più facile da capire. Il non determinismo negli automi finiti è anche una buona introduzione al non determinismo in modelli di computazione più potenti perché gli automi finiti sono particolarmente facili da capire. Ora passiamo a considerare diversi esempi di NFA.

ESEMPIO 1.30

Sia A il linguaggio che consiste di tutte le stringhe su {0,1} contenenti un 1 nella terza posizione dalla fine (ad esempio, 000100 è in A ma 0011 non lo è). Il seguente NFA N_2, con quattro stati, riconosce A.

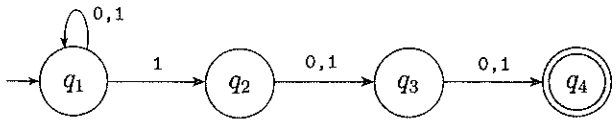


FIGURA 1.31
L'NFA N_2 che riconosce A

Un buon modo di vedere la computazione di questo NFA è dire che esso resta nello stato iniziale q_1 finché "ipotizza" che è a tre posizioni dalla fine. A questo punto, se il simbolo di input è un 1, passa nello stato q_2 e usa q_3 e q_4 per "controllare" se la sua ipotesi era corretta.

Come menzionato, ogni NFA può essere trasformato in un DFA equivalente; ma a volte questo DFA può avere molti più stati. Il più piccolo DFA per A contiene otto stati. Inoltre, capire il funzionamento dell'NFA è molto più facile, come si può vedere esaminando la figura seguente del DFA.

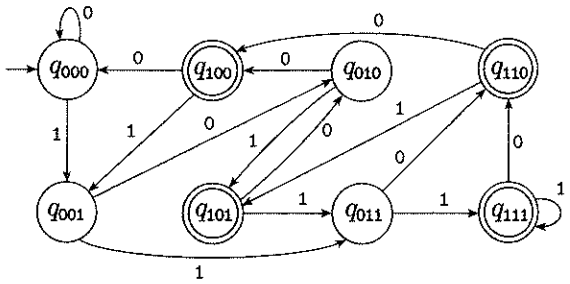


FIGURA 1.32
Un DFA che riconosce A

Supponiamo di aver aggiunto ε alle etichette sugli archi che vanno da q_2 a q_3 e da q_3 a q_4 nella macchina N_2 nella Figura 1.31. Quindi entrambi gli archi avrebbero allora le etichette $0, 1, \varepsilon$ invece di $0, 1$ soltanto. Quale linguaggio N_2 riconoscerebbe con questa variazione? Prova a modificare il DFA nella Figura 1.32 per riconoscere questo linguaggio.

ESEMPIO 1.33

Il seguente NFA N_3 ha un alfabeto {0} di simboli input che consiste di un solo simbolo. Un alfabeto contenente solo un simbolo è chiamato un ***alfabeto unario***.

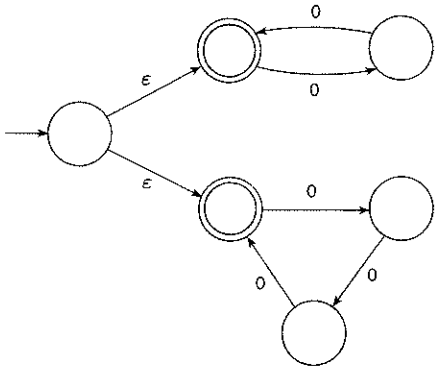


FIGURA 1.34
L'NFA N_3

Questa macchina mostra l'utilità di avere ε archi. Essa accetta tutte le stringhe della forma 0^k dove k è un multiplo di 2 o 3. (Ricorda che l'esponente denota ripetizione, non elevazione a potenza numerica.) Per esempio, N_3 accetta le stringhe ε, 00, 000, 0000, e 000000, ma non 0 o 00000.

Puoi vedere che la macchina opera inizialmente provando a indovinare se testare per un multiplo di 2 o un multiplo di 3, andando nel ciclo superiore o nel ciclo inferiore, e poi verificando se la propria ipotesi era corretta. Naturalmente, potremmo sostituire questa macchina con una che non ha ε-archi o perfino del tutto priva di non determinismo, ma la macchina mostrata è quella più facile da capire per questo linguaggio.

ESEMPIO 1.35

Diamo un altro esempio di un NFA in Figura 1.36. Esercitati con esso per convincerti che accetta le stringhe ε, a, baba, e baa, ma che non accetta le stringhe b, bb, e babba. Più tardi useremo questa macchina per illustrare la procedura per trasformare gli NFA in DFA.

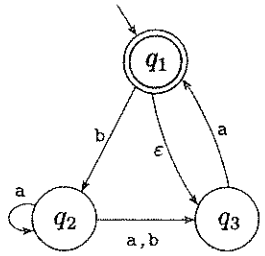


FIGURA 1.36
L'NFA N_4

Definizione formale di automa finito non deterministico

La definizione formale di automa finito non deterministico è simile a quella di automa finito deterministico. Entrambi hanno stati, un alfabeto di simboli di input, una funzione di transizione, uno stato iniziale, e una collezione di stati accettanti. Comunque, essi differiscono in un aspetto essenziale: nel tipo di funzione di transizione. In un DFA, la funzione di transizione prende uno stato e un simbolo di input e produce lo stato successivo. In un NFA, la funzione di transizione prende uno stato e un simbolo di input *o la stringa vuota* e produce *l'insieme dei possibili stati successivi*. Per scrivere la definizione formale, dobbiamo introdurre alcune notazioni aggiuntive. Per un qualsiasi insieme Q denotiamo con $\mathcal{P}(Q)$ la collezione di tutti i sottoinsiemi di Q. In questo caso $\mathcal{P}(Q)$ è chiamato l'***insieme potenza*** di Q. Per ogni alfabeto Σ scriviamo Σ_ε per denotare $\Sigma \cup \{\varepsilon\}$. Ora possiamo descrivere formalmente il tipo di funzione di transizione in un NFA come $\delta\colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$.

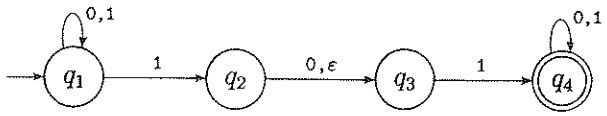
DEFINIZIONE **1.37**

Un ***automa finito non deterministico*** è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove

1. Q è un insieme finito di stati,
2. Σ è un alfabeto finito,
3. $\delta\colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ è la funzione di transizione,
4. $q_0 \in Q$ è lo stato iniziale, ed
5. $F \subseteq Q$ è l'insieme degli stati di accettazione.

ESEMPIO 1.38

Ricorda l'NFA N_1:



La descrizione formale di N_1 è $(Q, \Sigma, \delta, q_1, F)$, dove

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ è data da

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

4. q_1 è lo stato iniziale, ed
5. $F = \{q_4\}$.

La definizione formale di computazione per un NFA è simile a quella per un DFA. Sia $N = (Q, \Sigma, \delta, q_0, F)$ un NFA e sia w una stringa sull'alfabeto Σ. Diciamo che N ***accetta*** w se possiamo scrivere w come $w = y_1 y_2 \cdots y_m$, dove ciascun y_i è un elemento di Σ_ε ed esiste una sequenza di stati $r_0, r_1, \ldots, r_m$ in Q con tre condizioni:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m-1$, ed
3. $r_m \in F$.

La condizione 1 dice che la macchina inizia nello stato iniziale. La condizione 2 dice che lo stato r_{i+1} è uno dei possibili stati successivi quando N è nello stato r_i e sta leggendo y_{i+1}. Osserva che $\delta(r_i, y_{i+1})$ è l'*insieme* dei possibili stati successivi e quindi diciamo che r_{i+1} è un elemento di quell'insieme. Infine, la condizione 3 dice che la macchina accetta il suo input se l'ultimo stato è uno stato accettante.

Equivalenza tra gli NFA e i DFA

Gli automi finiti deterministici e quelli non deterministici riconoscono la stessa classe di linguaggi. Questa equivalenza è sia sorprendente che utile. È sorprendente perché gli NFA sembrano avere maggior potere computazionale dei DFA, quindi uno potrebbe aspettarsi che gli NFA riconoscono più linguaggi. È utile perché descrivere un NFA per un dato linguaggio a volte è molto più semplice che descrivere un DFA per quel linguaggio.

Diciamo che due macchine sono ***equivalenti*** se esse riconoscono lo stesso linguaggio.

TEOREMA 1.39

Per ogni automa finito non deterministico esiste un automa finito deterministico equivalente.

IDEA. Se un linguaggio è riconosciuto da un NFA, allora dobbiamo mostrare l'esistenza di un DFA che lo riconosce. L'idea è trasformare l'NFA in un DFA equivalente che simula l'NFA.

Ricorda la strategia "lettore come automa" per progettare automi finiti. Come simuleresti l'NFA se stessi immaginando di essere un DFA? Cosa devi memorizzare dell'elaborazione della stringa di input? Negli esempi degli NFA, mantenevi traccia dei vari rami della computazione ponendo un dito su ogni stato che potrebbe essere attivo in specificati punti nell'input. Aggiornavi la simulazione muovendo, aggiungendo e rimuovendo dita in base al modo in cui l'NFA opera. Tutto quello di cui dovevi tenere traccia era l'insieme degli stati che avevano dita su essi.

Se k è il numero degli stati dell'NFA, esso ha 2^k sottoinsiemi di stati. Ogni sottoinsieme corrisponde a una delle possibilità che il DFA deve ricordare, quindi il DFA che simula l'NFA avrà 2^k stati. Ora noi dobbiamo capire quali saranno lo stato iniziale e gli stati accettanti del DFA, e quale sarà la sua funzione di transizione. Possiamo discutere di questo più facilmente dopo aver introdotto alcune notazioni formali.

DIMOSTRAZIONE Sia $N = (Q, \Sigma, \delta, q_0, F)$ l'NFA che riconosce un linguaggio A. Costruiamo un DFA $M = (Q', \Sigma, \delta', q_0', F')$ che riconosce A. Prima di fare la costruzione completa, consideriamo inizialmente il caso più semplice in cui N non ha ε-archi. In seguito considereremo gli ε-archi.

1. $Q' = \mathcal{P}(Q)$.
   Ogni stato di M è un insieme di stati di N. Ricorda che $\mathcal{P}(Q)$ è l'insieme dei sottoinsiemi di Q.
2. Per $R \in Q'$ e $a \in \Sigma$, sia $\delta'(R, a) = \{q \in Q | \, q \in \delta(r, a)$ per qualche $r \in R\}$.
   Se R è uno stato di M, esso è anche un insieme di stati di N. Quando M legge un simbolo a nello stato R, mostra dove a porta ogni stato in R. Poiché da ogni stato si può andare in un insieme di stati, prendiamo l'unione di tutti questi insiemi. Un altro modo per scrivere questa espressione è
   $$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$[4]

[4] La notazione $\bigcup_{r \in R} \delta(r, a)$ significa: l'unione degli insiemi $\delta(r, a)$ per ogni possibile r in R.

3. $q_0' = \{q_0\}$.
   M inizia nello stato corrispondente alla collezione che contiene solo lo stato iniziale di N.
4. $F' = \{R \in Q' | \, R$ contiene uno stato accettante di $N\}$.
   La macchina M accetta se uno dei possibili stati in cui N potrebbe essere a quel punto è uno stato accettante.

Ora dobbiamo considerare gli ε-archi. Per farlo, introduciamo qualche ulteriore notazione. Per ogni stato R di M, definiamo $E(R)$ come la collezione di stati che possono essere raggiunti dagli elementi di R proseguendo solo con ε-archi, includendo gli stessi elementi di R. Formalmente, per $R \subseteq Q$ sia

$$E(R) = \{q | \, q \text{ può essere raggiunto da } R \text{ attraverso 0 o più } \varepsilon\text{-archi}\}.$$

Poi modifichiamo la funzione di transizione di M ponendo dita supplementari su tutti gli stati che possono essere raggiunti proseguendo attraverso ε-archi dopo ogni passo. Sostituendo $\delta(r, a)$ con $E(\delta(r, a))$ realizziamo questo effetto. Quindi

$$\delta'(R, a) = \{q \in Q | \, q \in E(\delta(r, a)) \text{ per qualche } r \in R\}.$$

Inoltre, dobbiamo modificare lo stato iniziale di M per muovere inizialmente le dita su tutti i possibili stati che possono essere raggiunti dallo stato iniziale di N attraverso gli ε-archi. Cambiare q_0' in $E(\{q_0\})$ realizza questo risultato. Ora abbiamo completato la costruzione del DFA M che simula l'NFA N.

Ovviamente la costruzione di M funziona correttamente. A ogni passo nella computazione di M su un input, chiaramente entra in uno stato che corrisponde al sottoinsieme di stati in cui N potrebbe essere a quel punto. Quindi la nostra prova è completa.

Il Teorema 1.39 afferma che ogni NFA può essere trasformato in un DFA equivalente. Quindi gli automi finiti non deterministici forniscono un modo alternativo di caratterizzare i linguaggi regolari. Esprimiamo questo fatto come corollario del Teorema 1.39.

COROLLARIO 1.40

Un linguaggio è regolare se e solo se qualche automa finito non deterministico lo riconosce.

Una direzione della condizione "se e solo se" afferma che un linguaggio è regolare se qualche NFA lo riconosce. Il Teorema 1.39 mostra che ogni NFA può essere trasformato in un DFA equivalente. Conseguentemente, se un NFA riconosce un linguaggio, lo fa anche qualche DFA, e quindi il linguaggio

è regolare. L'altra direzione della condizione "se e solo se" afferma che un linguaggio è regolare solo se qualche NFA lo riconosce. Cioè, se un linguaggio è regolare, qualche NFA deve riconoscerlo. Ovviamente, questa condizione è vera perché un linguaggio regolare ha un DFA che lo riconosce e un DFA è anche un NFA.

ESEMPIO 1.41

Illustriamo la procedura che abbiamo dato nella prova del Teorema 1.39 per trasformare un NFA in un DFA usando la macchina N_4 che appare (presente) nell'Esempio 1.35. Per chiarezza, abbiamo rinominato gli stati di N_4 in $\{1,2,3\}$. Quindi nella descrizione formale di $N_4 = (Q, \{\mathtt{a},\mathtt{b}\}, \delta, 1, \{1\})$, l'insieme degli stati Q è $\{1,2,3\}$ come mostrato nella Figura 1.42.

Per costruire un DFA D equivalente a N_4, innanzitutto determiniamo gli stati di D. N_4 ha tre stati, $\{1,2,3\}$, quindi costruiamo D con otto stati, uno per ogni sottoinsieme dell'insieme degli stati di N_4. Etichettiamo ognuno degli stati di D con il corrispondente sottoinsieme. Quindi l'insieme degli stati di D è

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

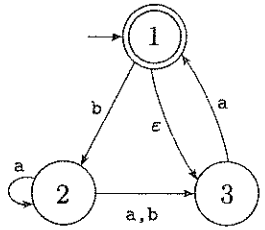


FIGURA 1.42
L'NFA N_4

Poi, determiniamo lo stato iniziale e gli stati accettanti di D. Lo stato iniziale è $E(\{1\})$, l'insieme degli stati che sono raggiungibili da 1 passando attraverso ε-archi, più 1 stesso. Un ε-arco va da 1 a 3, quindi $E(\{1\}) = \{1,3\}$. I nuovi stati accettanti sono quelli che contengono lo stato accettante di N_4; quindi $\{\{1\},\{1,2\},\{1,3\},\{1,2,3\}\}$.

Infine, determiniamo la funzione di transizione di D. Ciascuno degli stati di D va in una posizione sull'input a e in una posizione sull'input b. Illustriamo il processo di determinare la posizione degli archi di transizione di D con pochi esempi.

In D, lo stato $\{2\}$ va in $\{2,3\}$ sull'input a perché in N_4, lo stato 2 va sia in 2 che in 3 sull'input a e non possiamo andare più lontano da 2 o 3 attraverso ε-archi. Lo stato $\{2\}$ va nello stato $\{3\}$ sull'input b perché in N_4, lo stato 2 va solo nello stato 3 sull'input b e da 3 non possiamo andare più lontano attraverso ε archi.

Lo stato $\{1\}$ va in $\emptyset$ su a perché nessun arco etichettato con a esce da esso. Va in $\{2\}$ su b. Nota che la procedura nel Teorema 1.39 specifica che seguiamo gli ε archi *dopo* ogni simbolo di input che viene letto. Una procedura alternativa basata sul seguire gli ε archi prima di leggere ogni simbolo funziona ugualmente bene, ma questo metodo non è illustrato in questo esempio.

Lo stato $\{3\}$ va in $\{1,3\}$ su a perché in N_4, lo stato 3 va in 1 su a e 1 a sua volta va in 3 con un ε arco. Lo stato $\{3\}$ su b va in $\emptyset$.

Lo stato $\{1,2\}$ su a va in $\{2,3\}$ perché 1 non punta ad alcuno stato con a archi, 2 punta sia a 2 che a 3 con a archi, e nessuno dei due punta altrove con ε archi. Lo stato $\{1,2\}$ su b va in $\{2,3\}$. Continuando in questo modo, otteniamo il diagramma per D nella Figura 1.43.

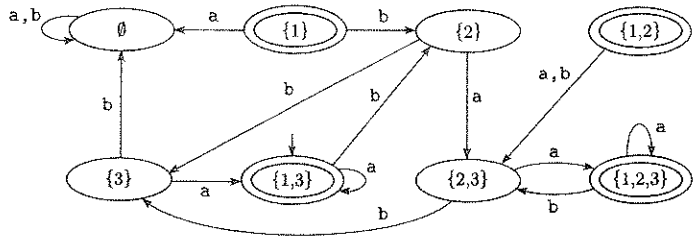


FIGURA 1.43
Un DFA D che è equivalente all'NFA N_4

Possiamo semplificare questa macchina osservando che nessun arco punta agli stati $\{1\}$ e $\{1,2\}$, quindi essi possono essere tolti senza ripercussioni sulla prestazione della macchina. Facendo così si ottiene la figura seguente.

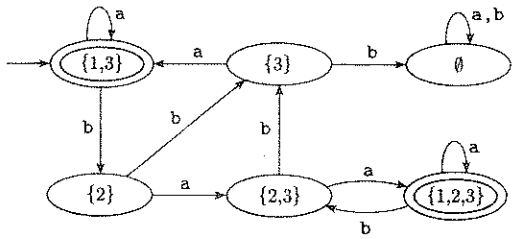


FIGURA 1.44
Il DFA D dopo aver tolto gli stati non necessari

Chiusura rispetto alle operazioni regolari

Ora torniamo alla chiusura della classe dei linguaggi regolari rispetto alle operazioni regolari che abbiamo iniziato nella Sezione 1.1. Il nostro scopo è

provare che l'unione, la concatenazione, e lo star di linguaggi regolari sono ancora regolari. Abbandonammo il tentativo originario di farlo quando trattare l'operazione di concatenazione risultava troppo complicato. L'uso del non determinismo rende le prove molto più semplici.

Innanzitutto, consideriamo di nuovo la chiusura rispetto all'unione. Prima abbiamo provato la chiusura rispetto all'unione simulando deterministicamente e simultaneamente entrambe le macchine, attraverso la costruzione di un prodotto cartesiano. Ora diamo una nuova prova per illustrare la tecnica del non determinismo. Riesaminare la prima prova, che è a pagina 48, può essere utile per vedere quanto più facile e più intuitiva sia la nuova prova.

TEOREMA 1.45

La classe dei linguaggi regolari è chiusa rispetto all'operazione di unione.

IDEA. Abbiamo i linguaggi regolari A_1 e A_2 e vogliamo provare che $A_1 \cup A_2$ è regolare. L'idea è prendere due NFA, N_1 ed N_2 per A_1 e A_2, e comporli in un nuovo NFA, N.

La macchina N deve accettare il suo input se N_1 o N_2 accetta questo input. La nuova macchina ha un nuovo stato iniziale che si dirama negli stati iniziali delle vecchie macchine con ε archi. In questo modo, la nuova macchina ipotizza non deterministicamente quale delle due macchine accetta l'input. Se una di esse accetta l'input, anche N lo accetterà.

Rappresentiamo questa costruzione nella figura seguente. Sulla sinistra, indichiamo lo stato iniziale e gli stati accettanti delle macchine N_1 ed N_2 con cerchi grandi e alcuni ulteriori stati con cerchi piccoli. Sulla destra, mostriamo come comporre N_1 ed N_2 in N aggiungendo archi di transizione supplementari.

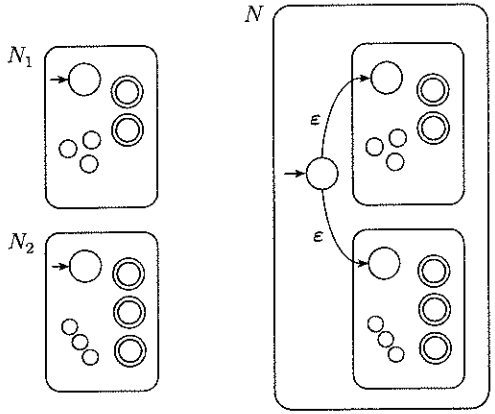


FIGURA 1.46
Costruzione di un NFA N per riconoscere $A_1 \cup A_2$

DIMOSTRAZIONE

Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ che riconosce A_1 ed
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ che riconosce A_2.

Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ per riconoscere $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
   Gli stati di N sono tutti gli stati di N_1 ed N_2, con l'aggiunta di un nuovo stato iniziale q_0.
2. Lo stato q_0 è lo stato iniziale di N.
3. L'insieme degli stati accettanti $F = F_1 \cup F_2$.
   Gli stati accettanti di N sono tutti gli stati accettanti di N_1 ed N_2. In questo modo, N accetta se N_1 accetta o N_2 accetta.
4. Definiamo δ in modo che per ogni $q \in Q$ e ogni $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon. \end{cases}$$

Ora possiamo provare la chiusura rispetto alla concatenazione. Ricorda che in precedenza, senza il non determinismo, completare la prova sarebbe stato difficile.

TEOREMA 1.47

La classe dei linguaggi regolari è chiusa rispetto all'operazione di concatenatione.

IDEA. Abbiamo i linguaggi regolari A_1 e A_2 e vogliamo provare che $A_1 \circ A_2$ è regolare. L'idea è prendere due NFA, N_1 ed N_2 per A_1 ed A_2, e combinarli in un nuovo NFA N come abbiamo fatto nel caso dell'unione, ma questa volta in un modo diverso, come mostrato nella Figura 1.48.

Poniamo come stato iniziale di N lo stato iniziale di N_1. Gli stati accettanti di N_1 hanno degli ulteriori ε-archi che non deterministicamente permettono di diramarsi in N_2 ogni volta che N_1 è in uno stato accettante, indicando che ha trovato un pezzo iniziale dell'input che costituisce una stringa in A_1. Gli stati accettanti di N sono solo gli stati accettanti di N_2. Quindi, esso accetta quando l'input può essere diviso in due parti, la prima accettata da N_1 e la seconda da N_2. Possiamo pensare che N non deterministicamente ipotizza dove effettuare la divisione.

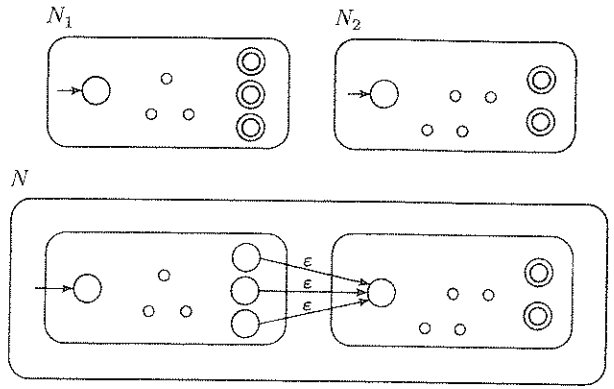


FIGURA 1.48
Costruzione di N per riconoscere $A_1 \circ A_2$

DIMOSTRAZIONE

Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ che riconosce A_1 ed
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ che riconosce A_2.

Costruiamo $N = (Q, \Sigma, \delta, q_1, F_2)$ per riconoscere $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.
   Gli stati di N sono tutti gli stati di N_1 ed N_2.
2. Lo stato q_1 è uguale allo stato iniziale di N_1.
3. Gli stati accettanti F_2 sono uguali agli stati accettanti di N_2.
4. Definiamo δ in modo che per ogni $q \in Q$ e ogni $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ e } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2. \end{cases}$$

TEOREMA 1.49

La classe dei linguaggi regolari è chiusa rispetto all'operazione star.

IDEA. Abbiamo un linguaggio regolare A_1 e vogliamo provare che anche A_1^* è regolare. Prendiamo un NFA N_1 per A_1 e lo modifichiamo per riconoscere A_1^*, come mostrato nella figura seguente. L'NFA N risultante accetterà il suo input quando esso può essere diviso in varie parti ed N_1 accetta ogni parte.

Possiamo costruire N come N_1 con ε-archi supplementari che dagli stati accettanti ritornano allo stato iniziale. In questo modo, quando l'elaborazione giunge alla fine di una parte che N_1 accetta, la macchina N ha la scelta di tornare indietro allo stato iniziale per provare a leggere un'altra parte che N_1 accetta. Inoltre, dobbiamo modificare N in modo che accetti ε, che è sempre un elemento di A_1^*. Un'idea (leggermente cattiva) è semplicemente aggiungere lo stato iniziale all'insieme degli stati accettanti. Questa strategia certamente aggiunge ε al linguaggio riconosciuto, ma può anche aggiungere altre stringhe non volute. L'Esercizio 1.15 chiede un esempio che mostri l'insuccesso di questa idea. Il modo per correggerla è aggiungere un nuovo stato iniziale, che è anche uno stato accettante, e che ha un ε-arco entrante nel vecchio stato iniziale. Questa soluzione ha l'effetto desiderato di aggiungere ε al linguaggio senza aggiungere qualcos'altro.

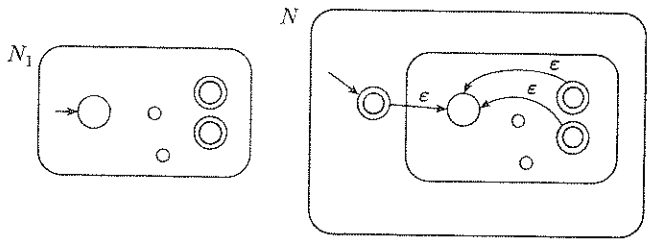


FIGURA 1.50
Costruzione di N per riconoscere A^*

DIMOSTRAZIONE Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ che riconosce A_1.
Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ per riconoscere A_1^*.

1. $Q = \{q_0\} \cup Q_1$.
   Gli stati di N sono gli stati di N_1 più un nuovo stato iniziale.
2. Lo stato q_0 è il nuovo stato iniziale.
3. $F = \{q_0\} \cup F_1$.
   Gli stati accettanti sono i vecchi stati accettanti più il nuovo stato iniziale.
4. Definiamo δ in modo che per ogni $q \in Q$ e ogni $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ e } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon. \end{cases}$$

1.3 ESPRESSIONI REGOLARI

In aritmetica, possiamo usare le operazioni + e × per costruire espressioni come

$$(5 + 3) \times 4 .$$

Analogamente, possiamo usare le operazioni regolari per costruire espressioni che descrivono linguaggi, che sono chiamate ***espressioni regolari***. Un esempio è:

$$(0 \cup 1)0^* .$$

Il valore dell'espressione aritmetica è il numero 32. Il valore di un'espressione regolare è un linguaggio. In questo caso, il valore è il linguaggio che consiste di tutte le stringhe che iniziano con uno 0 o un 1 seguito da un qualsiasi numero di simboli uguali a 0. Otteniamo questo risultato dividendo l'espressione nelle sue parti. In primo luogo, i simboli 0 e 1 sono abbreviazioni per gli insiemi $\{0\}$ e $\{1\}$. Quindi $(0 \cup 1)$ significa $(\{0\} \cup \{1\})$. Il valore di questa parte è il linguaggio $\{0,1\}$. La parte 0^* denota $\{0\}^*$, e il suo valore è il linguaggio che consiste di tutte le stringhe contenenti un numero qualsiasi di simboli uguali a 0. Inoltre, come il simbolo × in algebra, il simbolo della concatenazione $\circ$ è spesso sottinteso nelle espressioni regolari. Quindi $(0 \cup 1)0^*$ è in realtà un'abbreviazione per $(0 \cup 1) \circ 0^*$. La concatenazione connette le stringhe dalle due parti per ottenere il valore dell'intera espressione.

Le espressioni regolari hanno un ruolo importante nelle applicazioni dell'informatica. Nelle applicazioni che coinvolgono testo, gli utenti possono voler cercare stringhe che soddisfano alcuni schemi. Le espressioni regolari forniscono un metodo potente per descrivere tali schemi. Alcune utility come awk e grep in UNIX, linguaggi di programmazione moderni come Perl, ed editor di testo, tutti forniscono meccanismi per la descrizione di schemi usando espressioni regolari.

ESEMPIO 1.51

Un altro esempio di espressione regolare è

$$(0 \cup 1)^* .$$

Essa inizia con il linguaggio $(0 \cup 1)$ a cui applica l'operazione $*$. Il valore di questa espressione è il linguaggio che consiste di tutte le possibili stringhe di simboli 0 e 1. Se $\Sigma = \{0,1\}$, possiamo usare Σ come abbreviazione per l'espressione regolare $(0 \cup 1)$. Più in generale, se Σ è un qualsiasi alfabeto, l'espressione regolare Σ descrive il linguaggio che consiste di tutte le stringhe di lunghezza 1 su questo alfabeto e Σ^* descrive il linguaggio che consiste di tutte le stringhe su quell'alfabeto. Analogamente, $\Sigma^* 1$ è il linguaggio che contiene tutte le stringhe che terminano con 1. Il linguaggio $(0\Sigma^*) \cup (\Sigma^* 1)$ consiste di tutte le stringhe che iniziano con uno 0 o terminano con un 1.

In aritmetica, diciamo che × ha precedenza su + per intendere che, quando c'è una scelta, eseguiamo prima l'operazione ×. Quindi in $2 + 3 \times 4$, la moltiplicazione 3×4 è eseguita prima dell'addizione. Per far sì che l'addizione sia eseguita prima, dobbiamo aggiungere delle parentesi per ottenere $(2+3) \times 4$. Nelle espressioni regolari, l'operazione star è eseguita per prima, seguita dalla concatenazione e infine dall'unione, a meno che delle parentesi non cambino l'ordine usuale.

Definizione formale di espressione regolare

DEFINIZIONE **1.52**

Diciamo che R è un'***espressione regolare*** se R è

1. a per qualche a nell'alfabeto Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, dove R_1 ed R_2 sono espressioni regolari,
5. $(R_1 \circ R_2)$, dove R_1 ed R_2 sono espressioni regolari, o
6. (R_1^*), dove R_1 è un'espressione regolare.

Nei punti 1 e 2, le espressioni regolari a e ε rappresentano i linguaggi $\{a\}$ e $\{\varepsilon\}$, rispettivamente. Nel punto 3, l'espressione regolare $\emptyset$ rappresenta il linguaggio vuoto. Nei punti 4, 5, e 6, le espressioni rappresentano i linguaggi ottenuti prendendo l'unione o la concatenazione dei linguaggi R_1 ed R_2, o lo star del linguaggio R_1, rispettivamente.

Non confondere le espressioni regolari ε e $\emptyset$. L'espressione ε rappresenta il linguaggio che contiene una sola stringa – ossia, la stringa vuota – mentre $\emptyset$ rappresenta il linguaggio che non contiene alcuna stringa.

Apparentemente, sembra che stiamo definendo la nozione di espressione regolare in termini di sé stessa. Se fosse vero, avremmo una ***definizione circolare***, che non sarebbe valida. Comunque, R_1 ed R_2 sono sempre più piccole di R. Quindi in realtà stiamo definendo le espressioni regolari in termini di espressioni regolari più piccole, evitando in tal modo la circolarità. Una definizione di questo tipo è chiamata una ***definizione induttiva***.

Le parentesi in un'espressione possono essere omesse. Se lo sono, la valutazione è fatta nell'ordine della precedenza: star, poi concatenazione, poi unione.

Per comodità, usiamo R^+ come abbreviazione per RR^*. In altre parole, mentre R^* contiene tutte le stringhe che sono concatenazione di 0 o più stringhe di R, il linguaggio R^+ contiene tutte le stringhe che sono concatenazione di 1 o più stringhe di R. Quindi $R^+ \cup \varepsilon = R^*$. Inoltre, denotiamo con R^k l'abbreviazione della concatenazione di k copie di R insieme.

Quando vogliamo distinguere tra un'espressione regolare R e il linguaggio che descrive, denotiamo con $L(R)$ il linguaggio di R.

ESEMPIO 1.53

Negli esempi seguenti, assumiamo che l'alfabeto Σ sia $\{0,1\}$.

1. $0^*10^* = \{w \mid w \text{ contiene un solo } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ ha almeno un } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contiene la stringa } 001 \text{ come sottostringa}\}$.
4. $1^*(01^+)^* = \{w \mid \text{ogni } 0 \text{ in } w \text{ è seguito da almeno un } 1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ è una stringa di lunghezza pari}\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{la lunghezza di } w \text{ è un multiplo di } 3\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ inizia e termina con lo stesso simbolo}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   L'espressione $0 \cup \varepsilon$ descrive il linguaggio $\{0, \varepsilon\}$, quindi l'operazione di concatenazione aggiunge 0 o ε prima di ogni stringa in 1^*.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
    Concatenando l'insieme vuoto a un qualsiasi insieme si ottiene l'insieme vuoto.
12. $\emptyset^* = \{\varepsilon\}$.
    L'operazione star concatena un numero qualsiasi di stringhe del linguaggio per ottenere una stringa nel risultato. Se il linguaggio è vuoto, l'operazione star può concatenare 0 stringhe, dando solo la stringa vuota.

Se R è un'espressione regolare qualsiasi, abbiamo le seguenti identità. Esse sono un buon test per controllare se hai capito la definizione.

[5] La ***lunghezza*** di una stringa è il numero di simboli che essa contiene.

$R \cup \emptyset = R$.
Aggiungere il linguaggio vuoto a un qualsiasi altro linguaggio non lo cambia.

$R \circ \varepsilon = R$.
Concatenare la stringa vuota a una qualsiasi stringa non la cambia.

Tuttavia, scambiare $\emptyset$ e ε nelle precedenti identità può far sì che le uguaglianze non siano vere.

$R \cup \varepsilon$ può non essere uguale a R.
Per esempio, se $R = 0$, allora $L(R) = \{0\}$ ma $L(R \cup \varepsilon) = \{0, \varepsilon\}$.

$R \circ \emptyset$ può non essere uguale a R.
Per esempio, se $R = 0$, allora $L(R) = \{0\}$ ma $L(R \circ \emptyset) = \emptyset$.

Le espressioni regolari sono un utile strumento nel progetto dei compilatori per i linguaggi di programmazione. Gli oggetti elementari in un linguaggio di programmazione, chiamati ***token***, come i nomi delle variabili e le costanti, possono essere descritti con espressioni regolari. Per esempio, una costante numerica che può avere una parte decimale e/o un segno può essere descritta come un elemento del linguaggio

$$(+ \cup - \cup \varepsilon)\left(D^+ \cup D^+.D^* \cup D^*.D^+\right)$$

dove $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ è l'alfabeto delle cifre decimali. Esempi di stringhe generate sono: `72`, `3.14159`, `+7.`, e `-.01`.

Una volta che la sintassi di un linguaggio di programmazione è stata descritta con un'espressione regolare in termini dei suoi token, dei sistemi automatici possono generare l'***analizzatore lessicale***, la parte di un compilatore che elabora inizialmente il programma input.

Equivalenza con gli automi finiti

Le espressioni regolari e gli automi finiti sono equivalenti rispetto alla loro potenza descrittiva. Questo fatto è sorprendente perché gli automi finiti e le espressioni regolari a prima vista sembrano essere piuttosto differenti. Tuttavia, ogni espressione regolare può essere trasformata in un automa finito che riconosce il linguaggio che essa descrive, e viceversa. Ricorda che un linguaggio regolare è un linguaggio che è riconosciuto da qualche automa finito.

TEOREMA 1.54

Un linguaggio è regolare se e solo se qualche espressione regolare lo descrive.

Questo teorema deve essere dimostrato in entrambe le direzioni. Noi enunciamo e proviamo ciascuna direzione in un lemma separato.

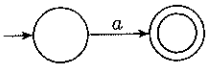
LEMMA 1.55

Se un linguaggio è descritto da un'espressione regolare, allora esso è regolare.

IDEA. Supponiamo di avere un'espressione regolare R che descrive un linguaggio A. Mostriamo come trasformare R in un NFA che riconosce A. Per il Corollario 1.40, se un NFA riconosce A allora A è regolare.

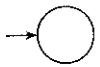
DIMOSTRAZIONE Trasformiamo R in un NFA N. Consideriamo i sei casi nella definizione di espressione regolare.

1. $R = a$ per qualche $a \in \Sigma$. Allora $L(R) = \{a\}$ e il seguente NFA riconosce $L(R)$.



   Nota che questa macchina soddisfa la definizione di NFA ma non quella di DFA perché ha qualche stato con nessun arco uscente per ogni possibile simbolo di input. Naturalmente, qui avremmo potuto presentare un DFA equivalente; ma un NFA è tutto quello di cui abbiamo bisogno per ora, ed è più facile da descrivere.
   Formalmente, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, dove descriviamo δ dicendo che $\delta(q_1, a) = \{q_2\}$ e $\delta(r, b) = \emptyset$ per $r \neq q_1$ o $b \neq a$.
2. $R = \varepsilon$. Allora $L(R) = \{\varepsilon\}$ e il seguente NFA riconosce $L(R)$.

   Formalmente, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, dove $\delta(r, b) = \emptyset$ per ogni r e b.
3. $R = \emptyset$. Allora $L(R) = \emptyset$, e il seguente NFA riconosce $L(R)$.



   Formalmente, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, dove $\delta(r, b) = \emptyset$ per ogni r e b.
4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

Per gli ultimi tre casi, usiamo le costruzioni date nelle prove che la classe dei linguaggi regolari è chiusa rispetto alle operazioni regolari. In altre parole, costruiamo l'NFA per R dagli NFA per R_1 ed R_2 (o solo R_1 nel caso 6) e mediante l'appropriata costruzione della chiusura.

Questo termina la prima parte della prova del Teorema 1.54, dando la direzione più facile della condizione "se e solo se". Prima di passare all'altra direzione, consideriamo alcuni esempi in cui usiamo questa procedura per trasformare un'espressione regolare in un NFA.

ESEMPIO 1.56

Trasformiamo l'espressione regolare (ab ∪ a)* in un NFA in una sequenza di passi. Costruiamo l'automa dalle sottoespressioni più piccole alle sottoespressioni più grandi finché abbiamo un NFA per l'espressione iniziale, come mostrato nel diagramma seguente. Nota che questa procedura generalmente non dà l'NFA con il minor numero di stati. In questo esempio, la procedura dà un NFA con otto stati, ma il più piccolo NFA equivalente ha solo due stati. Riesci a trovarlo?

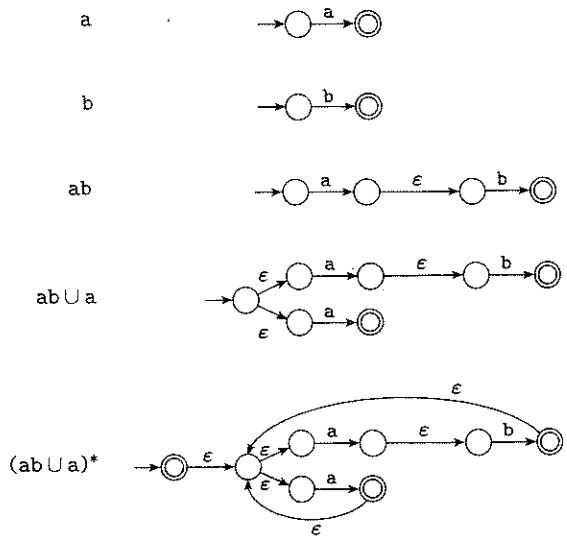


FIGURA 1.57
Costruzione di un NFA dall'espressione regolare (ab ∪ a)*

ESEMPIO 1.58

Nella Figura 1.59, trasformiamo l'espressione regolare $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$ in un NFA. Alcuni dei passi meno importanti non sono mostrati.

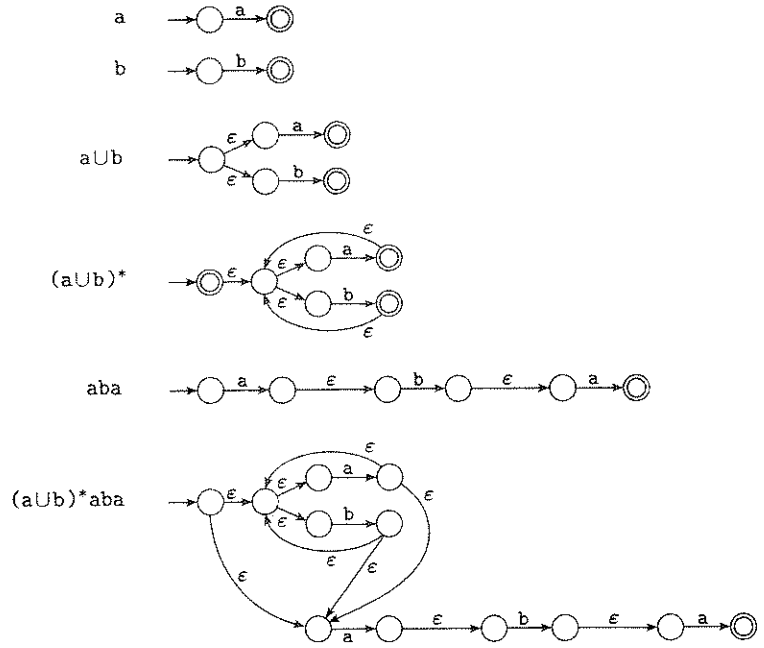


FIGURA 1.59
Costruzione di un NFA dall'espressione regolare $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$

Ora veniamo all'altra direzione della prova del Teorema 1.54.

LEMMA 1.60

Se un linguaggio è regolare, allora è descritto da un'espressione regolare.

IDEA. Dobbiamo mostrare che se un linguaggio A è regolare, allora un'espressione regolare lo descrive. Poiché A è regolare, esso è accettato da un DFA. Descriviamo una procedura per trasformare i DFA in espressioni regolari equivalenti.

Dividiamo questa procedura in due parti, usando un nuovo tipo di automa finito chiamato ***automa finito non deterministico generalizzato***,

GNFA. Prima mostriamo come trasformare un DFA in un GNFA e poi come trasformare un GNFA in un'espressione regolare.

Gli automi finiti non deterministici generalizzati sono semplicemente automi finiti non deterministici nei quali gli archi delle transizioni possono avere espressioni regolari come etichette, invece che solo elementi dell'alfabeto o ϵ. Il GNFA legge blocchi di simboli dall'input, non necessariamente solo un simbolo alla volta come in un comune NFA. Il GNFA si muove lungo un arco di transizione che collega due stati leggendo un blocco di simboli dall'input che formano una stringa descritta dall'espressione regolare su quell'arco. Un GNFA è non deterministico e quindi può avere diversi modi di elaborare la stessa stringa di input. Esso accetta il suo input se la sua elaborazione può far sì che il GNFA sia in uno stato accettante alla fine dell'input. La figura seguente presenta un esempio di un GNFA.

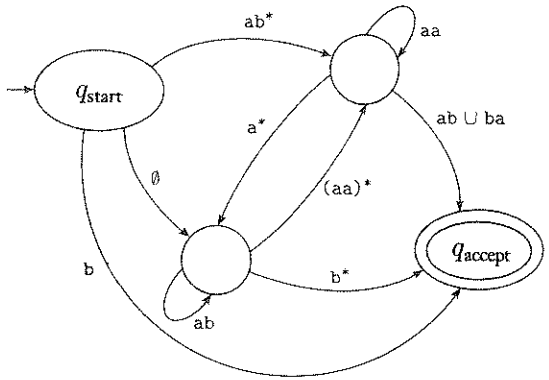


FIGURA 1.61
Un automa finito non deterministico generalizzato

Per comodità, richiediamo che i GNFA abbiano sempre una forma speciale che soddisfi le seguenti condizioni.

- Lo stato iniziale ha archi di transizione uscenti verso un qualsiasi altro stato ma nessun arco entrante proveniente da un qualsiasi altro stato.
- Esiste un solo stato accettante, ed esso ha archi entranti provenienti da un qualsiasi altro stato ma nessun arco uscente verso un qualsiasi altro stato. Inoltre, lo stato accettante non è uguale allo stato iniziale.
- Eccetto che per lo stato iniziale e lo stato accettante, un arco va da ogni stato ad ogni altro stato e anche da ogni stato in sé stesso.

Possiamo facilmente trasformare un DFA in un GNFA nella forma speciale. Aggiungiamo semplicemente un nuovo stato iniziale con un ϵ-arco che entra

nel vecchio stato iniziale e un nuovo stato accettante con ε-archi entranti, provenienti dai vecchi stati accettanti. Se alcuni archi hanno più etichette (o se ci sono più archi che collegano gli stessi due stati nella stessa direzione), sostituiamo ognuno di essi con un solo arco la cui etichetta è l'unione delle precedenti etichette. Infine, aggiungiamo archi con etichetta $\emptyset$ tra stati che non hanno archi. Questo ultimo passo non cambierebbe il linguaggio riconosciuto perché una transizione etichettata con $\emptyset$ non può mai essere usata. Da qui in poi assumiamo che tutti i GNFA sono nella forma speciale.

Ora mostriamo come trasformare un GNFA in un'espressione regolare. Supponiamo che il GNFA abbia k stati. Allora, poiché un GNFA deve avere uno stato iniziale e uno stato accettante ed essi devono essere diversi tra loro, sappiamo che $k \geq 2$. Se $k > 2$, costruiamo un GNFA equivalente con $k - 1$ stati. Questo passo può essere ripetuto sul nuovo GNFA fino a quando esso è ridotto a due stati. Se $k = 2$, il GNFA ha un solo arco che va dallo stato iniziale allo stato accettante. L'etichetta di questo arco è l'espressione regolare equivalente. Per esempio, le fasi per trasformare un DFA con tre stati in un'espressione regolare equivalente sono mostrati nella figura seguente.

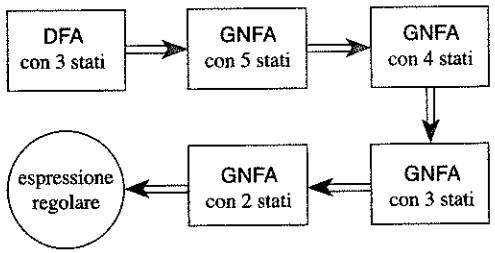


FIGURA 1.62
Fasi tipiche nella trasformazione di un DFA in un'espressione regolare

Il passo cruciale è costruire un GNFA equivalente con uno stato in meno quando $k > 2$. Lo facciamo scegliendo uno stato, estraendolo dalla macchina, e modificando il resto in modo che sia ancora riconosciuto lo stesso linguaggio. Ogni stato può essere scelto, purché non sia lo stato iniziale o lo stato accettante. Siamo certi che un tale stato esiste perché $k > 2$. Chiamiamo q_{rip} lo stato rimosso.

Dopo aver rimosso q_{rip} modifichiamo la macchina cambiando le espressioni regolari che etichettano ciascuno dei restanti archi. Le nuove etichette controbilanciano l'assenza di q_{rip} reintegrando le computazioni perse. La nuova etichetta che va da uno stato q_i a uno stato q_j è un'espressione regolare che descrive tutte le stringhe che porterebbero la macchina da q_i a q_j o direttamente o tramite q_{rip}. Illustriamo questa strategia nella Figura 1.63.

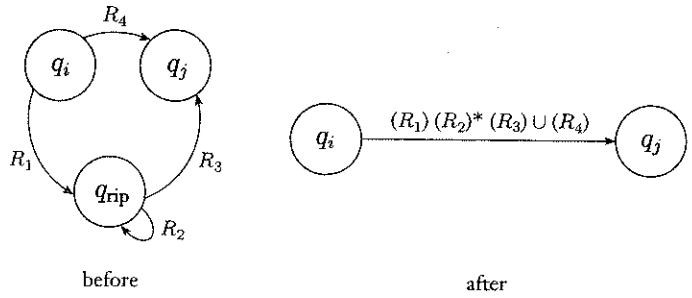


FIGURA 1.63
Costruzione di un GNFA equivalente con uno stato in meno

Nella vecchia macchina, se

1. q_i va a q_{rip} con un arco etichettato R_1,
2. q_{rip} va in sé stesso con un arco etichettato R_2,
3. q_{rip} va a q_j con un arco etichettato R_3,
4. q_i va a q_j con un arco etichettato R_4,

allora nella nuova macchina, l'arco da q_i a q_j riceve l'etichetta

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

Facciamo questa modifica per ogni arco che va da un qualsiasi stato q_i a un qualsiasi stato q_j, includendo il caso in cui $q_i = q_j$. La nuova macchina riconosce il linguaggio di partenza.

DIMOSTRAZIONE Formalizziamo questa idea. In primo luogo, per rendere più facile la prova, definiamo formalmente il nuovo tipo di automa introdotto. Un GNFA è simile a un automa finito non deterministico tranne che per la funzione di transizione, che ha la forma

$$\delta\colon \big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big) \longrightarrow \mathcal{R}.$$

Il simbolo $\mathcal{R}$ è la collezione di tutte le espressioni regolari sull'alfabeto Σ, e q_{start} e q_{accept} sono gli stati iniziale e accettante. Se $\delta(q_i, q_j) = R$, l'arco dallo stato q_i allo stato q_j ha come etichetta l'espressione regolare R. Il dominio della funzione di transizione è $\big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big)$ perché un arco collega ogni stato a un qualsiasi altro stato, tranne che nessun arco è uscente da q_{accept} o è entrante in q_{start}.

DEFINIZIONE **1.64**

Un ***automa finito non deterministico generalizzato*** è una quintupla, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, dove

1. Q è l'insieme finito degli stati,
2. Σ è l'alfabeto di input,
3. δ: $(Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \longrightarrow \mathcal{R}$ è la funzione di transizione,
4. q_{start} è lo stato iniziale e
5. q_{accept} è lo stato accettante.

Un GNFA accetta una stringa w in Σ^* se $w = w_1 w_2 \cdots w_k$, dove ogni w_i è in Σ^* ed esiste una sequenza di stati $q_0, q_1, \ldots, q_k$ tale che

1. $q_0 = q_{\text{start}}$ è lo stato iniziale,
2. $q_k = q_{\text{accept}}$ è lo stato accettante e
3. per ogni i, risulta $w_i \in L(R_i)$, dove $R_i = \delta(q_{i-1}, q_i)$; in altre parole, R_i è l'espressione sull'arco da q_{i-1} a q_i.

Tornando alla prova del Lemma 1.60, sia M il DFA per il linguaggio A. Allora trasformiamo M in un GNFA G aggiungendo un nuovo stato iniziale e un nuovo stato accettante e i necessari archi di transizione supplementari. Usiamo la procedura CONVERT(G), che prende un GNFA e restituisce un'espressione regolare equivalente. Questa procedura usa la ***ricorsione***, che significa che essa chiama sé stessa. Evitiamo un ciclo infinito perché la procedura chiama sé stessa solo per elaborare un GNFA che ha uno stato in meno. Il caso in cui il GNFA ha due stati è trattato senza ricorsione.

CONVERT(G):

1. Sia k il numero di stati di G.
2. Se $k = 2$, allora G deve consistere di uno stato iniziale, uno stato accettante, e un singolo arco che li collega ed è etichettato con un'espressione regolare R.
   Restituisce l'espressione R.
3. Se $k > 2$, scegliamo un qualsiasi stato $q_{\text{rip}} \in Q$ diverso da q_{start} e q_{accept} e sia G' il GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, dove
   $$Q' = Q - \{q_{\text{rip}}\},$$
   e per ogni $q_i \in Q' - \{q_{\text{accept}}\}$ e ogni $q_j \in Q' - \{q_{\text{start}}\}$, poniamo
   $$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$
   dove $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$ ed $R_4 = \delta(q_i, q_j)$.
4. Calcola CONVERT(G') e restituisce questo valore.

Di seguito proviamo che CONVERT restituisce un valore corretto.

FATTO 1.65

Per ogni GNFA G, CONVERT(G) è equivalente a G.

Proviamo quest'affermazione per induzione su k, il numero di stati del GNFA.

Base: Proviamo che l'affermazione è vera per $k = 2$ stati. Se G ha solo due stati, esso può avere solo un singolo arco, che va dallo stato iniziale allo stato accettante. L'espressione regolare che è l'etichetta di quest'arco descrive tutte le stringhe che permettono a G di raggiungere lo stato accettante. Quindi quest'espressione è equivalente a G.

Passo induttivo: Assumiamo che l'affermazione sia vera per $k - 1$ stati e usiamo questa ipotesi per provare che l'affermazione è vera per k stati. In primo luogo mostriamo che G e G' riconoscono lo stesso linguaggio. Supponiamo che G accetti un input w. Allora in un ramo accettante della computazione, G entra in una sequenza di stati:

$$q_{\text{start}}, q_1, q_2, q_3, \ldots, q_{\text{accept}}.$$

Se nessuno di essi è lo stato rimosso q_{rip}, evidentemente anche G' accetta w. La ragione è che ciascuna delle nuove espressioni regolari che etichettano gli archi di G' contiene le vecchie espressioni regolari come parte di un'unione.

Se q_{rip} è presente, eliminando ogni sequenza di stati q_{rip} consecutivi creiamo una computazione accettante per G'. Gli stati q_i e q_j che raggruppano una tale sequenza hanno una nuova espressione regolare sull'arco tra essi, la quale descrive tutte le stringhe che portano da q_i a q_j attraverso q_{rip} in G. Quindi G' accetta w.

Viceversa, supponiamo che G' accetti un input w. Poiché ciascun arco tra due qualsiasi stati q_i e q_j in G' descrive la collezione delle stringhe che portano da q_i a q_j in G, o direttamente o attraverso q_{rip}, anche G deve accettare w. Quindi G e G' sono equivalenti.

L'ipotesi induttiva afferma che quando l'algoritmo chiama sé stesso ricorsivamente sull'input G', il risultato è un'espressione regolare che è equivalente a G' perché G' ha $k - 1$ stati. Quindi anche questa espressione regolare è equivalente a G, e abbiamo provato che l'algoritmo è corretto.

Questo conclude la prova del Claim 1.65, del Lemma 1.60, e del Teorema 1.54.

ESEMPIO 1.66

In questo esempio, usiamo il precedente algoritmo per trasformare un DFA in un'espressione regolare. Iniziamo con il DFA con due stati nella Figura 1.67(a).

Nella Figura 1.67(b), creiamo un GNFA con quattro stati aggiungendo un nuovo stato iniziale e un nuovo stato accettante, chiamati s e a invece di q_{start} e q_{accept} in modo da poterli disegnare in modo conveniente. Per evitare di ingombrare la figura, non disegniamo gli archi etichettati $\emptyset$, sebbene essi vi siano. Nota che sostituiamo le etichette a, b sul ciclo nello stato 2 del DFA con l'etichetta a ∪ b nel corrispondente punto del GNFA. Facciamo in questo modo perché le etichette del DFA rappresentano due transizioni, una per a e l'altra per b, mentre il GNFA può avere solo una singola transizione che va da 2 a sé stesso.

Nella Figura 1.67(c), eliminiamo lo stato 2 e aggiorniamo le etichette degli archi restanti. In questo caso, la sola etichetta che cambia è quella da 1 ad a. Nella parte (b) era $\emptyset$, ma nella parte (c) essa è $\mathtt{b}(\mathtt{a}\cup\mathtt{b})^*$. Otteniamo questo risultato seguendo il passo 3 della procedura CONVERT. Lo stato q_i è lo stato 1, lo stato q_j è a, e q_{rip} è 2, quindi $R_1 = \mathtt{b}$, $R_2 = \mathtt{a}\cup\mathtt{b}$, $R_3 = \varepsilon$ ed $R_4 = \emptyset$. Pertanto, la nuova etichetta sull'arco da 1 ad a è $(\mathtt{b})(\mathtt{a}\cup\mathtt{b})^*(\varepsilon)\cup\emptyset$. Semplifichiamo quest'espressione regolare in $\mathtt{b}(\mathtt{a}\cup\mathtt{b})^*$.

Nella Figura 1.67(d), eliminiamo lo stato 1 dalla parte (c) e seguiamo la stessa procedura. Poiché restano solo lo stato iniziale e lo stato accettante, l'etichetta sull'arco che li collega è l'espressione regolare equivalente al DFA iniziale.

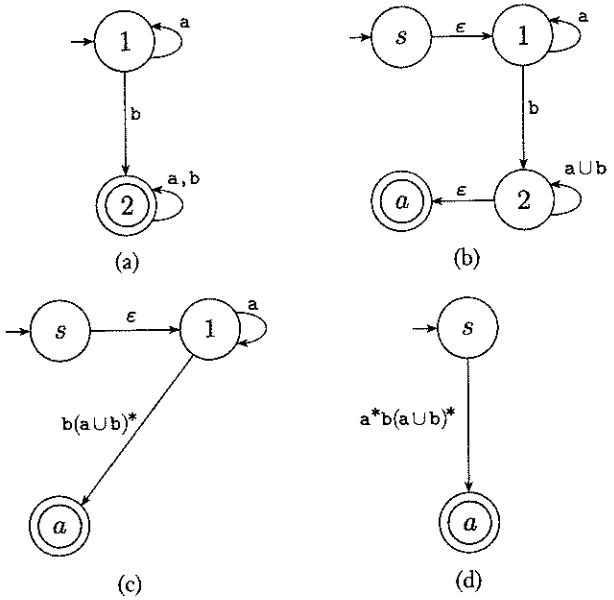


FIGURA 1.67
Conversione di un DFA con due stati in un'espressione regolare equivalente

ESEMPIO 1.68

In questo esempio, iniziamo con un DFA con tre stati. I passi nella trasformazione sono mostrati nella figura seguente.

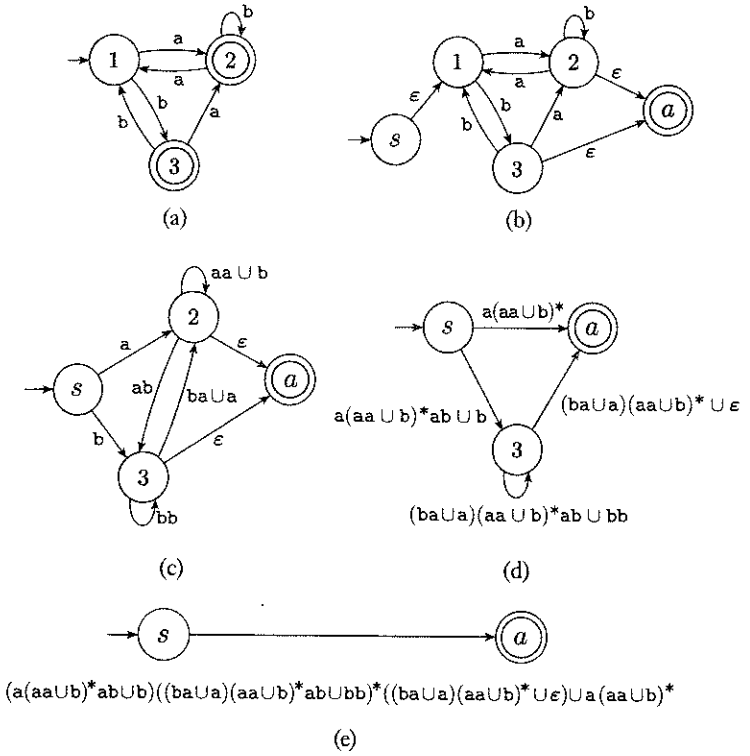


FIGURA 1.69
Conversione di un DFA con tre stati in un'espressione regolare equivalente

1.4
LINGUAGGI NON REGOLARI

Per comprendere il potere computazionale degli automi finiti dobbiamo anche comprenderne i limiti. In questa sezione, mostriamo come provare che alcuni linguaggi non possono essere riconosciuti da alcun automa finito.

Consideriamo il linguaggio $B = \{0^n1^n \mid n \geq 0\}$. Se cerchiamo di trovare un DFA che riconosca B, scopriamo che la macchina sembra aver bisogno di ricordare quanti simboli uguali a 0 ha visto fin quando legge l'input.

Poiché il numero di simboli uguali a 0 non è limitato, la macchina dovrà tenere traccia di un numero infinito di possibilità. Ma non può farlo con un numero finito di stati.

Di seguito presentiamo un metodo per provare che linguaggi come B non sono regolari. L'argomento prima presentato non prova la non regolarità poiché il numero di simboli uguali a 0 non è limitato? No, non lo ha fatto. Solo perché il linguaggio sembra richiedere memoria non limitata non significa che sia necessariamente così. É vero per il linguaggio B; ma altri linguaggi sembrano richiedere un numero non limitato di possibilità, e tuttavia sorprendentemente essi sono regolari. Per esempio, consideriamo i due linguaggi sull'alfabeto $\Sigma = \{0,1\}$:

$C = \{w | w$ ha lo stesso numero di simboli uguali a 0 e simboli uguali a $1\}$, e

$D = \{w |$ w ha un numero uguale di occorrenze di 01 e 10 come sottostringhe$\}$.

A prima vista, una macchina che riconosce sembra aver bisogno di contare in ciascun caso, e quindi nessuno dei due linguaggi sembra essere regolare. Come previsto, C non è regolare, ma sorprendentemente D è regolare![6] Quindi la nostra intuizione può a volte condurci nella direzione sbagliata, e questo è il motivo per cui abbiamo bisogno di dimostrazioni matematiche per la certezza. In questa sezione, mostriamo come provare che alcuni linguaggi non sono regolari.

Il pumping lemma per i linguaggi regolari

La nostra tecnica per provare la non regolarità deriva da un teorema sui linguaggi regolari, tradizionalmente chiamato il ***pumping lemma***. Questo teorema afferma che tutti i linguaggi regolari hanno una proprietà speciale. Se noi possiamo mostrare che un linguaggio non ha questa proprietà, siamo sicuri che esso non è regolare. La proprietà afferma che tutte le stringhe nel linguaggio possono essere "replicate" se la loro lunghezza raggiunge almeno uno specifico valore speciale, chiamato la ***lunghezza del pumping***. Questo significa che ogni tale stringa contiene una parte che può essere ripetuta un numero qualsiasi di volte ottenendo una stringa che appartiene ancora al linguaggio.

TEOREMA 1.70

Pumping lemma Se A è un linguaggio regolare, allora esiste un numero p (la lunghezza del pumping) tale che se s è una qualsiasi stringa in A di lunghezza almeno p, allora s può essere divisa in tre parti, $s = xyz$, soddisfacenti le seguenti condizioni:

1. per ogni $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, e
3. $|xy| \leq p$.

Ricordiamo che $|s|$ rappresenta la lunghezza della stringa s, y^i indica i copie di y concatenate insieme, e y^0 è uguale a ϵ.

Quando s è divisa in xyz, x o z potrebbe essere ϵ, ma la condizione 2 dice che $y \neq \epsilon$. Osserva che senza la condizione 2 il teorema sarebbe banalmente vero. La conditione 3 afferma che le parti x e y insieme hanno lunghezza al più p. È una condizione tecnica supplementare che ogni tanto troviamo utile quando dimostriamo che alcuni linguaggi non sono regolari. Vedi l'Esempio 1.74 per un'applicazione della condizione 3.

IDEA. Sia $M = (Q, \Sigma, \delta, q_1, F)$ un DFA che riconosce A. Assegniamo alla lunghezza del pumping p il numero degli stati di M. Mostriamo che ogni stringa s in A di lunghezza almeno p può essere divisa nelle tre parti xyz, soddisfacenti le nostre tre condizioni. Cosa accade se nessuna stringa in A è di lunghezza almeno p? Allora il nostro compito è perfino più facile perché il teorema diventa *banalmente* vero: ovviamente le tre condizioni valgono per tutte le stringhe di lunghezza almeno p se non vi è alcuna di tali stringhe.

Se s in A ha lunghezza almeno p, consideriamo la sequenza di stati che M attraversa nella computazione con input s. Inizia con lo stato iniziale q_1, poi va per esempio in q_3, poi per esempio in q_{20}, poi in q_9, e così via, finché raggiunge la fine di s nello stato q_{13}. Se s è in A, sappiamo che M accetta s, quindi q_{13} è uno stato accettante.

Se supponiamo che n sia la lunghezza di s, la sequenza di stati $q_1, q_3, q_{20}, q_9, \ldots, q_{13}$ ha lunghezza $n+1$. Poiché n è almeno p, sappiamo che $n+1$ è più grande di p, il numero di stati di M. Quindi, la sequenza deve contenere uno stato che si ripete. Questo risultato è un esempio del ***principio della piccionaia*** (*pigeonhole principle*), un nome di fantasia per il fatto piuttosto ovvio che se p piccioni sono messi in una piccionaia con meno di p nicchie, qualche nicchia deve avere più di un piccione in essa.

La figura seguente mostra la stringa s e la sequenza di stati che M attraversa quando elabora s. Lo stato q_9 è quello che si ripete.

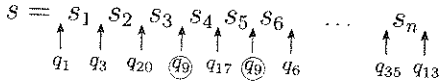


FIGURA 1.71
Esempio che mostra lo stato q_9 che si ripete quando M legge s

[6] Vedi il Problema 1.53.

Ora dividiamo s nelle tre componenti x, y e z. La componente x è la parte di s che compare prima di q_9, la componente y è la parte tra le due occorrenze di q_9 e la componente z è la parte restante di s, che viene dopo la seconda occorrenza di q_9. Quindi x porta M dallo stato q_1 a q_9, y riporta M da q_9 a q_9 e z porta M da q_9 allo stato accettante q_{13}, come mostrato nella figura seguente.

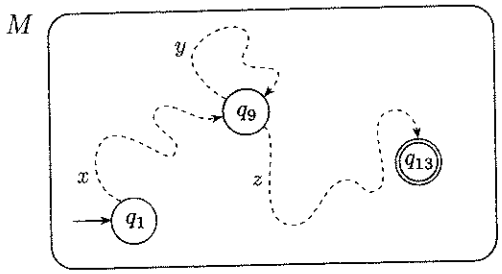


FIGURA 1.72
Esempio che mostra l'effetto delle stringhe x, y e z su M

Vediamo perché questa divisione di s soddisfa le tre condizioni. Supponiamo di eseguire M sull'input $xyyz$. Sappiamo che x porta M da q_1 a q_9, e poi il primo y lo riporta da q_9 a q_9, come fa il secondo y e poi z lo porta in q_{13}. Essendo q_{13} uno stato accettante, M accetta l'input $xyyz$. Analogamente, esso accetterà xy^iz per ogni $i > 0$. Nel caso $i = 0$, $xy^iz = xz$, che è accettata per ragioni analoghe. Questo prova la condizione 1.

Per verificare la condizione 2, vediamo che $|y| > 0$, poiché era la parte di s tra due diverse occorrenze dello stato q_9.

Per ottenere la condizione 3, ci assicuriamo che q_9 sia la prima ripetizione nella sequenza. Per il principio della piccionaia, i primi $p + 1$ stati nella sequenza devono contenere una ripetizione. Quindi, $|xy| \leq p$.

DIMOSTRAZIONE Sia $M = (Q, \Sigma, \delta, q_1, F)$ un DFA che riconosce A e sia p il numero di stati di M.

Sia $s = s_1 s_2 \cdots s_n$ una stringa in A di lunghezza n, dove $n \geq p$. Sia $r_1, \ldots, r_{n+1}$ la sequenza di stati attraversati da M mentre elabora s, quindi $r_{i+1} = \delta(r_i, s_i)$ per $1 \leq i \leq n$. Questa sequenza ha lunghezza $n + 1$, che è almeno $p + 1$. Due tra i primi $p + 1$ elementi nella sequenza devono essere lo stesso stato, per il principio della piccionaia. Chiamiamo il primo di questi r_j e il secondo r_l. Poiché r_l si presenta tra le prime $p + 1$ posizioni in una sequenza che inizia in r_1, abbiamo $l \leq p+1$. Ora sia $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$ e $z = s_l \cdots s_n$.

Poiché x porta M da r_1 a r_j, y porta M da r_j a r_j e z porta M da r_j a r_{n+1}, che è uno stato accettante, M deve accettare xy^iz per $i \geq 0$. Sappiamo che $j \neq l$, perciò $|y| > 0$; e $l \leq p+1$, perciò $|xy| \leq p$. Quindi tutte le condizioni del pumping lemma sono rispettate.

Per usare il pumping lemma per provare che un linguaggio B non è regolare, in primo luogo si assuma che B sia regolare per ottenere una contraddizione. Poi si usi il pumping lemma per assicurare l'esistenza di una lunghezza del pumping p tale che tutte le stringhe di lunghezza maggiore o uguale a p in B possano essere iterate. In seguito, si trovi una stringa s in B che ha lunghezza maggiore o uguale a p, ma che non può essere iterata. Infine, si dimostri che s non può essere iterata considerando tutti i modi di dividere s in x, y e z (prendendo in considerazione la condizione 3 del pumping lemma se è utile) e, per ogni tale divisione, trovando un valore i tale che $xy^iz \notin B$. Questo passo finale spesso comporta il dover raggruppare i vari modi di dividere s in diversi casi e l'analizzarli individualmente. L'esistenza di s contraddirebbe il pumping lemma se B fosse regolare. Quindi B non può essere regolare.

Trovare s a volte richiede un po' di ragionamento creativo. Potresti dover cercare tra diversi candidati per s prima di scoprirne uno che funzioni. Prova con elementi di B che sembrano esibire l'"essenza" della non regolarità di B. Discutiamo ulteriormente il compito di trovare s in alcuni degli esempi seguenti.

ESEMPIO 1.73

Sia B il linguaggio $\{0^n1^n \mid n \geq 0\}$. Usiamo il pumping lemma per provare che B non è regolare. La prova è per assurdo.

Assumiamo al contrario che B sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Scegliamo s uguale alla stringa 0^p1^p. Poiché s è un elemento di B ed s ha lunghezza maggiore di p, il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, tali che per ogni $i \geq 0$ la stringa xy^iz è in B. Consideriamo tre casi per mostrare che questo risultato è impossibile.

1. La stringa y consiste solo di simboli uguali a 0. In questo caso, la stringa $xyyz$ ha più simboli uguali a 0 che simboli uguali a 1 e quindi non è un elemento di B, violando la condizione 1 del pumping lemma. Questo caso porta a una contraddizione.
2. La stringa y consiste solo di simboli uguali a 1. Anche questo caso porta a una contraddizione.
3. La stringa y consiste sia di simboli uguali a 0 che di simboli uguali a 1. In questo caso, la stringa $xyyz$ può avere lo stesso numero di simboli

uguali a 0 e simboli uguali a 1, ma essi non saranno nell'ordine corretto, con qualche 1 prima di qualche 0. Quindi non è un elemento di B, il che è una contraddizione.

Pertanto una contraddizione è inevitabile se assumiamo che B sia regolare, quindi B non è regolare. Nota che possiamo semplificare questo ragionamento applicando la condizione 3 del pumping lemma per eliminare i casi 2 e 3.

In questo esempio, trovare la stringa s era facile perché una qualsiasi stringa in B di lunghezza p o maggiore avrebbe funzionato. Nei successivi due esempi, alcune scelte per s non funzionano quindi è richiesta un'attenzione supplementare.

ESEMPIO 1.74

Sia $C = \{w \mid w$ ha lo stesso numero di simboli uguali a 0 e simboli uguali a $1\}$. Usiamo il pumping lemma per provare che C non è regolare. La prova è per contraddizione.

Assumiamo al contrario che C sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Come nell'Esempio 1.73, sia s la stringa 0^p1^p. Poiché s è un elemento di C che ha lunghezza maggiore di p, il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, tali che per ogni $i \geq 0$ la stringa xy^iz è in C. Ci piacerebbe mostrare che questo risultato è impossibile. Ma aspetta, esso *è* possibile! Se prendiamo x e z uguali alla stringa vuota e y uguale alla stringa 0^p1^p, allora xy^iz ha sempre lo stesso numero di simboli uguali a 0 e simboli uguali a 1 e quindi è in C. Quindi *sembra* che s possa essere iterata.

Qui la condizione 3 nel pumping lemma è utile. Essa stabilisce che quando iteriamo s, essa deve essere divisa in modo che $|xy| \leq p$. Questa restrizione sul modo in cui s può essere divisa rende più facile mostrare che la stringa $s = 0^p1^p$ che abbiamo scelto non può essere iterata. Se $|xy| \leq p$, allora y deve consistere solo di simboli uguali a 0, perciò $xyyz \notin C$. Quindi, s non può essere iterata. Questo ci fornisce la contraddizione desiderata.

Scegliere la stringa s in questo esempio richiede più attenzione che nell'Esempio 1.73. Se invece avessimo scelto $s = (01)^p$, avremmo avuto dei problemi perché abbiamo bisogno di una stringa che *non può* essere iterata e quella stringa *può* essere iterata, perfino prendendo in considerazione la condizione 3. Riesci a vedere come iterarla? Un modo per farlo è porre $x = \varepsilon$, $y = 01$ e $z = (01)^{p-1}$. Allora $xy^iz \in C$ per ogni valore di i. Se non riesci a trovare una stringa che non può essere iterata al primo tentativo, non disperare. Provane un'altra!

Un metodo alternativo per provare che C non è regolare segue dal fatto che sappiamo che B non è regolare. Se C fosse regolare, anche $C \cap 0^*1^*$ sarebbe regolare. Questo perché il linguaggio 0^*1^* è regolare e la classe dei linguaggi regolari è chiusa rispetto all'intersezione, come provammo nella nota a fondo pagina 3 (pagina 49). Ma $C \cap 0^*1^*$ è uguale a B, e noi sappiamo che B non è regolare dall'Esempio 1.73.

ESEMPIO 1.75

Sia $F = \{ww \mid w \in \{0,1\}^*\}$. Mostriamo che F non è regolare, usando il pumping lemma.

Assumiamo al contrario che F sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Sia s la stringa 0^p10^p1. Poiché s è un elemento di F ed s ha lunghezza maggiore di p, il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, che verificano le tre condizioni del lemma. Mostriamo che questo risultato è impossibile.

La condizione 3 è ancora una volta cruciale perché senza di essa potremmo iterare s se poniamo x e z uguali alla stringa vuota. Con la condizione 3 la prova segue poiché y deve consistere solo di simboli uguali a 0, quindi $xyyz \notin F$.

Osserva che abbiamo scelto $s = 0^p10^p1$ come stringa che esibisce l'"essenza" della non regolarità di F, invece per esempio della stringa 0^p0^p. Sebbene 0^p0^p sia un elemento di F, essa non riesce a dar luogo a una contraddizione poiché può essere iterata.

ESEMPIO 1.76

Mostriamo un linguaggio non regolare unario. Sia $D = \{1^{n^2} \mid n \geq 0\}$. In altre parole, D contiene tutte le stringhe di simboli uguali a 1 la cui lunghezza è un quadrato perfetto. Usiamo il pumping lemma per provare che D non è regolare. La prova è per contraddizione.

Assumiamo al contrario che D sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Sia s la stringa 1^{p^2}. Poiché s è un elemento di D ed s ha lunghezza almeno p, il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, tali che per ogni $i \geq 0$ la stringa xy^iz è in D. Come negli esempi precedenti, mostriamo che questo risultato è impossibile. Farlo in questo caso richiede una piccola riflessione sulla successione dei quadrati perfetti:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

Nota il divario crescente tra gli elementi successivi di questa sequenza. Elementi grandi di questa sequenza non possono essere vicini l'un l'altro.

Ora consideriamo le due stringhe xyz e xy^2z. Queste stringhe differiscono l'una dall'altra per una sola ripetizione di y, e conseguentemente le loro lunghezze differiscono di una quantità uguale alla lunghezza di y. Per la condizione 3 del pumping lemma, $|xy| \leq p$ e quindi $|y| \leq p$. Abbiamo $|xyz| = p^2$ e allora $|xy^2z| \leq p^2+p$. Ma $p^2+p < p^2+2p+1 = (p+1)^2$. Inoltre, la condizione 2 implica che y non è la stringa vuota e perciò $|xy^2z| > p^2$.

Quindi, la lunghezza di xy^2z è compresa strettamente tra i quadrati perfetti consecutivi p^2 e $(p+1)^2$. Perciò questa lunghezza non può essere essa stessa un quadrato perfetto. Dunque arriviamo alla contraddizione che $xy^2z \notin D$ e concludiamo che D non è regolare.

ESEMPIO 1.77

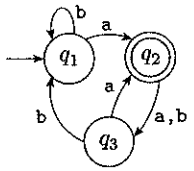
Talvolta "cancellare" ("pumping down") è utile quando applichiamo il pumping lemma. Usiamo il pumping lemma per mostrare che $E = \{0^i1^j | i > j\}$ non è regolare. La prova è per contraddizione.

Assumiamo che E sia regolare. Sia p la lunghezza del pumping per E data dal pumping lemma. Sia $s = 0^{p+1}1^p$. Allora s può essere divisa in xyz, dove le tre parti soddisfano le condizioni del pumping lemma. Per la condizione 3, y consiste solo di simboli uguali a 0. Esaminiamo la stringa $xyyz$ per vedere se essa può essere in E. Aggiungere una copia supplementare di y aumenta il numero di simboli uguali a 0. Ma E contiene tutte le stringhe in 0^*1^* che hanno più simboli uguali a 0 che simboli uguali a 1, quindi aumentando il numero di simboli uguali a 0 otterremo ancora una parola in E. Non vi è contraddizione. Dobbiamo provare qualcos'altro.

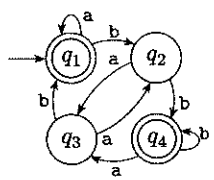
Il pumping lemma afferma che $xy^iz \in E$ anche quando $i = 0$, quindi consideriamo la stringa $xy^0z = xz$. Eliminare la stringa y fa diminuire il numero di simboli uguali a 0 in s. Ricorda che s ha solo uno 0 in più dei simboli uguali a 1. Pertanto, xz non può avere più simboli uguali a 0 di quelli uguali a 1, di conseguenza non può essere un elemento di E. Quindi otteniamo una contraddizione.

ESERCIZI

A1.1 Di seguito sono riportati i diagrammi di stato di due DFA, M_1 ed M_2. Rispondere alle domande seguenti per ognuna di queste macchine.



M_1



M_2

a. Qual è lo stato iniziale?
b. Qual è l'insieme degli stati accettanti?
c. Qual è la sequenza di stati che la macchina attraversa sull'input aabb?
d. La macchina accetta la stringa aabb?
e. La macchina accetta la stringa ε?

A1.2 Dare la descrizione formale delle macchine M_1 ed M_2 disegnate nell'Esercizio 1.1.

1.3 La descrizione formale di un DFA M è $(\{q_1, q_2, q_3, q_4, q_5\}, \{u, d\}, \delta, q_3, \{q_3\})$, dove δ è data dalla tabella seguente. Dare il diagramma di stato di questa macchina.

	u	d
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_2	q_4
q_4	q_3	q_5
q_5	q_4	q_5

1.4 Ciascuno dei seguenti linguaggi è l'intersezione di due linguaggi più semplici. Per ognuno di essi, costruire i DFA per i linguaggi più semplici, poi comporli usando la costruzione discussa nella nota a fondo pagina 3 (pagina 49) per dare il diagramma di stato di un DFA per il linguaggio dato. Per ognuno di essi, $\Sigma = \{a, b\}$.

a. $\{w | w$ ha almeno tre a e almeno due b$\}$
Ab. $\{w | w$ ha esattamente due a e almeno due b$\}$
c. $\{w | w$ ha un numero pari di a e uno o due b$\}$
Ad. $\{w | w$ ha un numero pari di a e ogni a è seguita da almeno una b$\}$
e. $\{w | w$ inizia con una a e ha al più una b$\}$
f. $\{w | w$ ha un numero dispari di a e termina con una b$\}$
g. $\{w | w$ ha lunghezza pari e un numero dispari di a$\}$

1.5 Ciascuno dei seguenti linguaggi è il complemento di un linguaggio più semplice. Per ognuno di essi, costruire un DFA per il linguaggio più semplice, poi usarlo per dare il diagramma di stato di un DFA per il linguaggio dato. Per ognuno di essi, $\Sigma = \{a, b\}$.

Aa. $\{w | w$ non contiene la sottostringa ab$\}$
Ab. $\{w | w$ non contiene la sottostringa baba$\}$
c. $\{w | w$ non contiene la sottostringa ab né la sottostringa ba$\}$
d. $\{w | w$ è una stringa non in $a^*b^*\}$
e. $\{w | w$ è una stringa non in $(ab^+)^*\}$
f. $\{w | w$ è una stringa non in $a^* \cup b^*\}$
g. $\{w | w$ è una stringa che non contiene esattamente due a$\}$
h. $\{w | w$ è una stringa diversa da a e b$\}$

1.6 Dare i diagrammi di stato di DFA che riconoscono i seguenti linguaggi. Per ciascuno di essi, l'alfabeto è {0,1}.

a. $\{w | w$ inizia con un 1 e termina con uno 0$\}$
b. $\{w | w$ contiene almeno tre simboli uguali a 1$\}$
c. $\{w | w$ contiene la sottostringa 0101 (cioè, $w = x0101y$ per qualche x e y)$\}$

d. $\{w \mid w$ ha lunghezza almeno 3 e il suo terzo simbolo è uno $0\}$

e. $\{w \mid w$ inizia con uno 0 e ha lunghezza dispari, oppure inizia con un 1 e ha lunghezza pari$\}$

f. $\{w \mid w$ non contiene la sottostringa 110$\}$

g. $\{w \mid$ la lunghezza di w è al più $5\}$

h. $\{w \mid w$ è una stringa diversa da 11 e 111$\}$

i. $\{w \mid$ in ogni posizione dispari di w c'è il simbolo $1\}$

j. $\{w \mid w$ contiene almeno due simboli uguali a 0 e al più un $1\}$

k. $\{\varepsilon, 0\}$

l. $\{w \mid w$ contiene un numero pari di simboli uguali a 0, oppure contiene esattamente due simboli uguali a $1\}$

m. L'insieme vuoto

n. Tutte le stringhe eccetto la stringa vuota

1.7 Dare i diagrammi di stato di NFA con il numero indicato di stati, che riconoscono i seguenti linguaggi. Per ognuno di essi, l'alfabeto è $\{0,1\}$.

[A]a. Il linguaggio $\{w \mid w$ termina con $00\}$ con tre stati

b. Il linguaggio dell'Esercizio 1.6c con cinque stati

c. Il linguaggio dell'Esercizio 1.6l con sei stati

d. Il linguaggio $\{0\}$ con due stati

e. Il linguaggio $0^*1^*0^+$ con tre stati

[A]f. Il linguaggio $1^*(001^+)^*$ con tre stati

g. Il linguaggio $\{\varepsilon\}$ con uno stato

h. Il linguaggio 0^* con uno stato

1.8 Usare la costruzione nella prova del Teorema 1.45 per dare i diagrammi di stato degli NFA che riconoscono l'unione dei linguaggi descritti negli

a. Esercizi 1.6a e 1.6b.

b. Esercizi 1.6c e 1.6f.

1.9 Usare la costruzione nella prova del Teorema 1.47 per dare i diagrammi di stato degli NFA che riconoscono la concatenazione dei linguaggi descritti negli

a. Esercizi 1.6g e 1.6i.

b. Esercizi 1.6b e 1.6m.

1.10 Usare la costruzione nella prova del Teorema 1.49 per dare i diagrammi di stato degli NFA che riconoscono lo star dei linguaggi descritti in

a. Esercizio 1.6b.

b. Esercizio 1.6j.

c. Esercizio 1.6m.

[A]1.11 Provare che ogni NFA può essere trasformato in uno equivalente che ha un solo stato accettante.

1.12 Sia $D = \{w \mid w$ contiene un numero pari di a e un numero dispari di b e non contiene la sottostringa ab$\}$. Fornire un DFA con cinque stati che riconosce D e un'espressione regolare che genera D. (Suggerimento: Descrivere D più semplicemente.)

1.13 Sia F il linguaggio di tutte le stringhe su $\{0,1\}$ che non contengono una coppia di simboli uguali a 1 separati da un numero dispari di simboli. Fornire il diagramma di stato di un DFA con cinque stati che riconosce F. (Potresti trovare utile trovare prima un NFA con 4 stati per il complemento di F.)

1.14

a. Mostrare che se M è un DFA che riconosce il linguaggio B, scambiare gli stati accettanti con gli stati non accettanti in M definisce un nuovo DFA che riconosce il complemento di B. Concludere che la classe dei linguaggi regolari è chiusa rispetto al complemento.

b. Mostrare con un esempio che se M è un NFA che riconosce il linguaggio C, scambiare gli stati accettanti con gli stati non accettanti in M non necessariamente definisce un nuovo NFA che riconosce il complemento di C. La classe dei linguaggi riconosciuti da NFA è chiusa rispetto al complemento? Giustifica la tua risposta.

1.15 Dare un controesempio per mostrare che la costruzione seguente non dimostra il Teorema 1.49, la chiusura della classe dei linguaggi regolari rispetto all'operazione star.[7] Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ un automa che riconosce A_1. Costruire $N = (Q_1, \Sigma, \delta, q_1, F)$ come segue. N dovrebbe riconoscere A_1^*.

a. Gli stati di N sono gli stati di N_1.

b. Lo stato iniziale di N è lo stato iniziale di N_1.

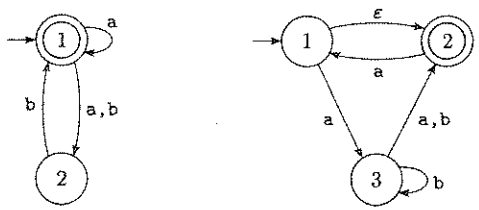
c. $F = \{q_1\} \cup F_1$.
Gli stati accettanti F sono i vecchi stati accettanti più il suo stato iniziale.

d. Definire δ in modo che per ogni $q \in Q_1$ e ogni $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \notin F_1 \text{ o } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ e } a = \varepsilon. \end{cases}$$

(Suggerimento: Mostrare questa costruzione graficamente, come nella Figura 1.50.)

1.16 Usare la costruzione data nel Teorema 1.39 per trasformare i due automi finiti non deterministici seguenti in automi finiti deterministici equivalenti.



(a) (b)

1.17

a. Dare un NFA che riconosce il linguaggio $(01 \cup 001 \cup 010)^*$.

b. Trasformare questo NFA in un DFA equivalente. Dare solo la parte del DFA che è raggiungibile dallo stato iniziale.

1.18 Fornire espressioni regolari che generino i linguaggi dell'Esercizio 1.6.

---

[7] In altre parole, devi esibire un automa finito, N_1, per il quale l'automa N ottenuto mediante la costruzione non riconosce lo star del linguaggio di N_1.

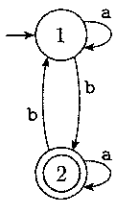
1.19 Usare la procedura descritta nel Lemma 1.55 per trasformare le espressioni regolari seguenti in automi finiti non deterministici.

a. $(0 \cup 1)^*000(0 \cup 1)^*$
b. $(((00)^*(11)) \cup 01)^*$
c. $\emptyset^*$

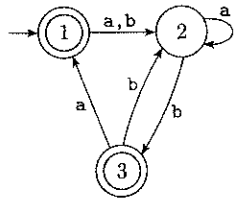
1.20 Per ciascuno dei seguenti linguaggi, fornire due stringhe che ne sono elementi e due stringhe che *non* lo sono – un totale di quattro stringhe per ogni linguaggio. Assumere che l'alfabeto sia $\Sigma = \{a,b\}$ per ognuno di essi.

a. a^*b^*
b. $a(ba)^*b$
c. $a^* \cup b^*$
d. $(aaa)^*$
e. $\Sigma^*a\Sigma^*b\Sigma^*a\Sigma^*$
f. $aba \cup bab$
g. $(\varepsilon \cup a)b$
h. $(a \cup ba \cup bb)\Sigma^*$

1.21 Usare la procedura descritta nel Lemma 1.60 per trasformare gli automi finiti seguenti in espressioni regolari.



(a)



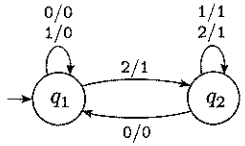
(b)

1.22 In alcuni linguaggi di programmazione, i commenti appaiono tra delimitatori come `/#` e `#/`. Sia C il il linguaggio di tutte le stringhe di commento correttamente delimitate. Un elemento di C deve iniziare con `/#` e terminare con `#/` ma non deve avere alcun `#/` intermedio. Per semplicità, assumere che l'alfabeto per C sia $\Sigma = \{a, b, /, \#\}$.

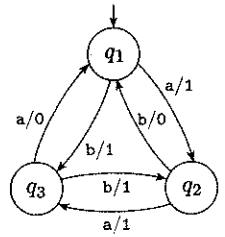
a. Dare un DFA che riconosce C.
b. Dare un'espressione regolare che genera C.

A1.23 Sia B un linguaggio sull'alfabeto Σ. Provare che $B = B^+$ se e solo se $BB \subseteq B$.

1.24 Un ***trasduttore a stati finiti*** (FST) è uno tipo di automa finito deterministico il cui output è una stringa e non solo *accetta* o *rifiuta*. Di seguito sono riportati i diagrammi di stato dei trasduttori a stati finiti T_1 e T_2.



T_1



T_2

Ogni transizione di un FST è etichettata con due simboli, di cui uno denota il simbolo di input per quella transizione e l'altro denota il simbolo in uscita. I due simboli sono separati da una barra, /. In T_1, la transizione da q_1 a q_2 ha come simbolo di input 2 e come simbolo di uscita 1. Alcune transizioni possono avere più coppie input–output, come la transizione in T_1 da q_1 a sé stesso. Quando un FST esegue una computazione su una stringa in input w, esso prende i simboli input $w_1 \cdots w_n$ uno a uno e, partendo dallo stato iniziale, segue le transizioni facendo corrispondere le etichette di input con la sequenza di simboli $w_1 \cdots w_n = w$. Ogni volta che effettua una transizione, esso fornisce in uscita il corrispondente simbolo di output. Per esempio, sull'input `2212011`, la macchina T_1 passa attraverso la sequenza di stati $q_1, q_2, q_2, q_2, q_2, q_1, q_1, q_1$ e produce in output `1111000`. Sull'input `abbb`, T_2 fornisce in output `1011`. Dare la sequenza degli stati attraversati e l'output prodotto in ciascuno dei casi seguenti.

a. T_1 sull'input `011`
b. T_1 sull'input `211`
c. T_1 sull'input `121`
d. T_1 sull'input `0202`
e. T_2 sull'input `b`
f. T_2 sull'input `bbab`
g. T_2 sull'input `bbbbbb`
h. T_2 sull'input ε

1.25 Leggere la definizione informale del trasduttore a stati finiti data nell'Esercizio 1.24. Dare una definizione formale di questo modello, seguendo lo schema nella Definizione 1.5 (pagina 38). Assumere che un FST ha un alfabeto di input Σ e un alfabeto di output Γ ma non ha un insieme di stati accettanti. Includere una definizione formale della computazione di un FST. (Suggerimento: Un FST è una quintupla. La sua funzione di transizione è della forma $\delta\colon Q \times \Sigma \longrightarrow Q \times \Gamma$.)

1.26 Usando la soluzione data all'Esercizio 1.25, dare una descrizione formale delle macchine T_1 e T_2 disegnate nell'Esercizio 1.24.

1.27 Leggere la definizione informale di trasduttore a stati finiti data nell'Esercizio 1.24. Dare il diagramma di stato di un FST con il seguente comportamento. I suoi alfabeti di input e output sono $\{0,1\}$. La stringa in output è identica alla stringa di input nelle posizioni pari ma complementata nelle posizioni dispari. Per esempio, sull'input `0000111` dovrebbe dare in uscita `1010010`.

1.28 Trasformare le espressioni regolari seguenti in NFA usando la procedura data nel Teorema 1.54. In tutti i casi, $\Sigma = \{a, b\}$.

a. $a(abb)^* \cup b$
b. $a^+ \cup (ab)^+$
c. $(a \cup b^+)a^+b^+$

1.29 Usare il pumping lemma per mostrare che i linguaggi seguenti non sono regolari.

[A]**a.** $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$

b. $A_2 = \{www \mid w \in \{a,b\}^*\}$

[A]**c.** $A_3 = \{a^{2^n} \mid n \geq 0\}$ (Qui, a^{2^n} indica una stringa di 2^n simboli uguali a.)

1.30 Descrivere l'errore nella seguente "prova" che 0^*1^* non è un linguaggio regolare. (Un errore deve esserci perché 0^*1^* è regolare.) La prova è per contraddizione. Assumere che 0^*1^* sia regolare. Sia p la lunghezza del pumping per 0^*1^* data dal pumping lemma. Scegliere s uguale alla stringa 0^p1^p. Sai che s è un elemento di 0^*1^*, ma l'Esempio 1.73 mostra che s non può essere iterata. Allora hai una contraddizione. Quindi 0^*1^* non è regolare.

PROBLEMI

1.31 Dati i linguaggi A e B, lo ***shuffle perfetto*** di A e B è il linguaggio

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ dove } a_1 \cdots a_k \in A \text{ e } b_1 \cdots b_k \in B, \text{ ogni } a_i, b_i \in \Sigma\}.$$

Mostrare che la classe dei linguaggi regolari è chiusa rispetto allo shuffle perfetto.

1.32 Dati i linguaggi A e B, lo ***shuffle*** di A e B è il linguaggio

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ dove } a_1 \cdots a_k \in A \text{ e } b_1 \cdots b_k \in B, \text{ ogni } a_i, b_i \in \Sigma^*\}.$$

Mostrare che la classe dei linguaggi regolari è chiusa rispetto allo shuffle.

1.33 Sia A un linguaggio. Si definisca $DROP\text{-}OUT(A)$ come il linguaggio contenente tutte le stringhe che possono essere ottenute togliendo un simbolo da una stringa in A. Quindi, $DROP\text{-}OUT(A) = \{xz \mid xyz \in A \text{ dove } x, z \in \Sigma^*, y \in \Sigma\}$. Mostrare che la classe dei linguaggi regolari è chiusa rispetto all'operazione $DROP\text{-}OUT$. Dare sia una prova mediante un disegno, sia una prova più formale mediante una costruzione come nel Teorema 1.47.

[A]**1.34** Siano B e C linguaggi su $\Sigma = \{0,1\}$. Si definisca

$$B \stackrel{1}{\leftarrow} C = \{w \in B \mid \text{per qualche } y \in C, \text{ le stringhe } w \text{ e } y \text{ hanno un ugual numero di } 1\}.$$

Mostrare che la classe dei linguaggi regolari è chiusa rispetto all'operazione $\stackrel{1}{\leftarrow}$.

[*]**1.35** Sia $A/B = \{w \mid wx \in A \text{ per qualche } x \in B\}$. Mostrare che se A è regolare e B è un linguaggio qualsiasi, allora A/B è regolare.

1.36 Per una stringa $w = w_1 w_2 \cdots w_n$, l'***inversa*** di w, denotata $w^{\mathcal{R}}$, è la stringa w in ordine inverso, $w_n \cdots w_2 w_1$. Per ogni linguaggio A, sia $A^{\mathcal{R}} = \{w^{\mathcal{R}} \mid w \in A\}$. Mostrare che se A è regolare, lo è anche $A^{\mathcal{R}}$.

1.37 Sia

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Σ_3 contiene tutte le colonne di simboli 0 e 1 di dimensione 3. Una stringa di simboli in Σ_3 dà luogo a tre righe di simboli 0 e 1. Considerare ogni riga come un numero binario e sia

$$B = \{w \in \Sigma_3^* \mid \text{la riga inferiore di } w \text{ è la somma delle due righe superiori}\}.$$

Per esempio,

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \in B, \quad \text{ma} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \notin B.$$

Mostrare che B è regolare. (Suggerimento: Lavorare con $B^{\mathcal{R}}$ è più facile. Puoi assumere vero il risultato affermato nel Problema 1.36.)

1.38 Sia

$$\Sigma_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

In questo caso, Σ_2 contiene tutte le colonne di simboli 0 e 1 di altezza due. Una stringa di simboli in Σ_2 dà luogo a due righe di simboli 0 e 1. Considerare ciascuna riga come un numero binario e sia

$$C = \{w \in \Sigma_2^* \mid \text{la riga inferiore di } w \text{ è tre volte la riga superiore}\}.$$

Per esempio, $\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} \in C$, ma $\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} \notin C$. Mostrare che C è regolare. (Puoi assumere vero il risultato affermato nel Problema 1.36.)

1.39 Sia Σ_2 lo stesso del Problema 1.38. Considerare ciascuna riga come un numero binario e sia

$$D = \{w \in \Sigma_2^* \mid \text{la riga superiore di } w \text{ è un numero più grande della riga inferiore}\}.$$

Per esempio, $\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} \in D$. ma $\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} \notin D$. Mostrare che D è regolare.

1.40 Sia Σ_2 lo stesso del Problema 1.38. Considerare le righe superiore e inferiore come stringhe di simboli 0 e 1, e sia

$$E = \{w \in \Sigma_2^* \mid \text{la riga inferiore di } w \text{ è l'inversa della riga superiore di } w\}.$$

Mostrare che E non è regolare.

1.41 Sia $B_n = \{a^k \mid k \text{ è un multiplo di } n\}$. Mostrare che per ogni $n \geq 1$, il linguaggio B_n è regolare.

1.42 Sia $C_n = \{x \mid x \text{ è un numero binario multiplo di } n\}$. Mostrare che per ogni $n \geq 1$, il linguaggio C_n è regolare.

1.43 Un ***all*-NFA** M è una quintupla $(Q, \Sigma, \delta, q_0, F)$ che accetta $x \in \Sigma^*$ se *ogni* possibile stato in cui M potrebbe essere dopo aver letto l'input x è uno stato in F. Nota, invece, che un NFA qualsiasi accetta una stringa se *qualche* stato tra questi possibili stati è uno stato accettante. Provare che gli all-NFA riconoscono la classe dei linguaggi regolari.

1.44 La costruzione nel Teorema 1.54 mostra che ogni GNFA è equivalente a un GNFA con solo due stati. Possiamo mostrare che si verifica un fenomeno opposto per i DFA. Provare che per ogni $k > 1$, esiste un linguaggio $A_k \subseteq \{0,1\}^*$ che è riconosciuto da un DFA con k stati ma non da uno con solo $k-1$ stati.

1.45 Ricorda che una stringa x è un ***prefisso*** di una stringa y se esiste una stringa z tale che $xz = y$, e che x è un ***prefisso proprio*** di y se inoltre $x \neq y$. In ognuno dei punti seguenti, definiamo un'operazione su un linguaggio A. Mostrare che la classe dei linguaggi regolari è chiusa rispetto a questa operazione.

[A]**a.** $NOPREFIX(A) = \{w \in A \mid \text{nessun prefisso proprio di } w \text{ è un elemento di } A\}$.

b. $NOEXTEND(A) = \{w \in A \mid w \text{ non è prefisso proprio di alcuna stringa in } A\}$.

[A]**1.46** Leggere la definizione informale di trasduttore a stati finiti data nell'Esercizio 1.24. Provare che nessun FST può dare in uscita $w^{\mathcal{R}}$ per ogni input w se gli alfabeti di input e output sono {0,1}.

1.47 Siano x e y stringhe e sia L un linguaggio. Diciamo che x e y sono ***distinguibili da L*** se esiste una stringa z tale che esattamente una delle stringhe xz e yz è un elemento di L; altrimenti, per ogni stringa z, abbiamo $xz \in L$ ogni volta che $yz \in L$ e diciamo che x e y sono ***indistinguibili da L***. Se x e y sono indistinguibili da L, scriviamo $x \equiv_L y$. Mostrare che $\equiv_L$ è una relazione di equivalenza.

A* **1.48** **Myhill–Nerode theorem.** Riferirsi al Problema 1.47. Sia L un linguaggio e sia X un insieme di stringhe. Diciamo che X è ***distinguibile a coppie da L*** se tutte le coppie di stringhe distinte in X sono distinguibili da L. Definire l'***indice di L*** come il numero massimo di elementi in un insieme che è distinguibile a coppie da L. L'indice di L può essere finito o infinito.

- **a.** Mostrare che se L è riconosciuto da un DFA con k stati, L ha al più indice k.
- **b.** Mostrare che se l'indice di L è un numero finito k, esso è riconosciuto da un DFA con k stati.
- **c.** Concludere che L è regolare se e solo se ha indice finito. Inoltre, il suo indice è la dimensione del più piccolo DFA che lo riconosce.

1.49 Considerare il linguaggio $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i,j,k \geq 0 \text{ e se } i = 1 \text{ allora } j = k\}$.

- **a.** Mostrare che F non è regolare.
- **b.** Mostrare che F si comporta come un linguaggio regolare rispetto al pumping lemma. In altre parole, dare una lunghezza del pumping p e dimostrare che F soddisfa le tre condizioni del pumping lemma per questo valore di p.
- **c.** Spiegare perché i punti (a) e (b) non contraddicono il pumping lemma.

1.50 Il pumping lemma afferma che ogni linguaggio regolare ha una lunghezza del pumping p, tale che ogni stringa del linguaggio può essere iterata se ha lunghezza maggiore o uguale a p. Se p è una lunghezza del pumping per il linguaggio A, tale è ogni lunghezza $p' \geq p$. La ***lunghezza minima del pumping*** per A è il più piccolo p che è una lunghezza del pumping per A. Per esempio, se $A = \mathtt{01}^*$, la lunghezza minima del pumping è 2. Questo perché la stringa $s = \mathtt{0}$ è in A, ha lunghezza 1 ed s non può essere iterata; ma ogni stringa in A di lunghezza maggiore o uguale a 2 contiene un 1 e quindi può essere iterata dividendola in modo che $x = \mathtt{0}$, $y = \mathtt{1}$ e z sia la restante parte della stringa. Per ognuno dei linguaggi seguenti, dare la lunghezza minima del pumping e giustificare la risposta.

- A**a.** $\mathtt{0001}^*$
- A**b.** $\mathtt{0}^*\mathtt{1}^*$
- **c.** $\mathtt{001} \cup \mathtt{0}^*\mathtt{1}^*$
- A**d.** $\mathtt{0}^*\mathtt{1}^+\mathtt{0}^+\mathtt{1}^* \cup \mathtt{10}^*\mathtt{1}$
- **e.** $(\mathtt{01})^*$
- **f.** ϵ
- **g.** $\mathtt{1}^*\mathtt{01}^*\mathtt{01}^*$
- **h.** $\mathtt{10}(\mathtt{11}^*\mathtt{0})^*\mathtt{0}$
- **i.** $\mathtt{1011}$
- **j.** Σ^*

1.51 Provare che i linguaggi seguenti non sono regolari. Puoi usare il pumping lemma e la chiusura della classe dei linguaggi regolari rispetto all'unione, all'intersezione e al complemento.

- **a.** $\{\mathtt{0}^n\mathtt{1}^m\mathtt{0}^n \mid m,n \geq 0\}$
- A**b.** $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$
- **c.** $\{w \mid w \in \{\mathtt{0},\mathtt{1}\}^* \text{ non è palindroma}\}$[8]
- ***d.** $\{wtw \mid w,t \in \{\mathtt{0},\mathtt{1}\}^+\}$

[8]Una stringa è ***palindroma*** se rimane uguale letta da sinistra a destra e da destra a sinistra.

1.52 Sia $\Sigma = \{\mathtt{1},\mathtt{\#}\}$ e sia

$$Y = \{w \mid w = x_1\mathtt{\#}x_2\mathtt{\#}\cdots\mathtt{\#}x_k \text{ con } k \geq 0, \text{ ciascun } x_i \in \mathtt{1}^* \text{ e } x_i \neq x_j \text{ per } i \neq j\}.$$

Provare che Y non è regolare.

1.53 Sia $\Sigma = \{\mathtt{0},\mathtt{1}\}$ e sia

$$D = \{w \mid w \text{ contiene un ugual numero di occorrenze delle sottostringhe } \mathtt{01} \text{ e } \mathtt{10}\}.$$

Quindi $\mathtt{101} \in D$ perché 101 contiene una sola occorrenza di 01 e una sola occorrenza di 10, ma $\mathtt{1010} \notin D$ perché 1010 contiene due occorrenze di 10 e una di 01. Mostrare che D è un linguaggio regolare.

1.54 Sia $\Sigma = \{\mathtt{a},\mathtt{b}\}$. Per ogni $k \geq 1$, sia C_k il linguaggio che consiste di tutte le stringhe che contengono una a esattamente nella k-esima posizione dalla destra. Quindi $C_k = \Sigma^*\mathtt{a}\Sigma^{k-1}$. Descrivere un NFA con $k+1$ stati che riconosce C_k sia attraverso un diagramma di stato sia mediante una descrizione formale.

1.55 Considerare i linguaggi C_k definiti nel Problema 1.54. Provare che per ogni k, nessun DFA può riconoscere C_k con meno di 2^k stati.

1.56 Sia $\Sigma = \{\mathtt{a},\mathtt{b}\}$. Per ogni $k \geq 1$, sia D_k il linguaggio che consiste di tutte le stringhe che hanno almeno una a tra gli ultimi k simboli. Quindi $D_k = \Sigma^*\mathtt{a}(\Sigma \cup \epsilon)^{k-1}$. Descrivere un DFA con al più $k+1$ stati che riconosce D_k sia attraverso un diagramma di stato sia mediante una descrizione formale.

***1.57**
- **a.** Sia A un linguaggio regolare infinito. Provare che A può essere diviso in due sottoinsiemi disgiunti infiniti regolari.
- **b.** Siano B e D due linguaggi. Scriviamo $B \Subset D$ se $B \subseteq D$ e D contiene infinite stringhe che non sono in B. Mostrare che se B e D sono due linguaggi regolari tali che $B \Subset D$, allora possiamo trovare un linguaggio regolare C tale che $B \Subset C \Subset D$.

1.58 Sia N un NFA con k stati che riconosce un linguaggio A.

- **a.** Mostrare che se A non è vuoto, A contiene qualche stringa di lunghezza al più k.
- **b.** Mostrare, dando un esempio, che il punto (a) non è necessariamente vero se sostituisci entrambi gli A con $\overline{A}$.
- **c.** Mostrare che se $\overline{A}$ non è vuoto, $\overline{A}$ contiene qualche stringa di lunghezza al più 2^k.
- **d.** Mostrare che il limite dato nel punto (c) è quasi stretto; cioè, per ogni k, esibire un NFA che riconosce un linguaggio A_k tale che $\overline{A_k}$ non è vuoto e dove le stringhe più corte di $\overline{A_k}$ sono di lunghezza esponenziale in k. Avvicinati al limite in (c) più che puoi.

***1.59** Provare che per ogni $n > 0$, esiste un linguaggio B_n tale che

- **a.** B_n è riconosciuto da un NFA che ha n stati e
- **b.** se $B_n = A_1 \cup \cdots \cup A_k$, per linguaggi regolari A_i, allora almeno uno degli A_i richiede un DFA con un numero di stati esponenziale.

1.60 Un ***omomorfismo*** è una funzione $f\colon \Sigma \longrightarrow \Gamma^*$ da un alfabeto all'insieme delle stringhe su un altro alfabeto. Possiamo estendere f per operare sulle stringhe definendo $f(w) = f(w_1)f(w_2)\cdots f(w_n)$, dove $w = w_1w_2\cdots w_n$ e ciascun $w_i \in \Sigma$. Possiamo ulteriormente estendere f per operare sui linguaggi definendo $f(A) = \{f(w) \mid w \in A\}$, per ogni linguaggio A.

a. Mostrare, dando una costruzione formale, che la classe dei linguaggi regolari è chiusa rispetto agli omomorfismi. In altre parole, dato un DFA M che riconosce B e un omomorfismo f, costruire un automa finito M' che riconosce $f(B)$. Considera la macchina M' che hai costruito. È un DFA in ogni caso?

b. Mostrare, dando un esempio, che la classe dei linguaggi non regolari non è chiusa rispetto agli omomorfismi.

1.61** Sia $RC(A) = \{yx | xy \in A\}$ la ***chiusura rotazionale del linguaggio A.

a. Mostrare che per ogni linguaggio A, risulta $RC(A) = RC(RC(A))$.

b. Mostrare che la classe dei linguaggi regolari è chiusa rispetto alla chiusura rotazionale.

1.62 Sia $\Sigma = \{0, 1, +, =\}$ e

$$ADD = \{x=y+z | \, x, y, z \text{ sono interi binari e } x \text{ è la somma di } y \text{ e } z\}.$$

Mostrare che ADD non è regolare.

***1.63** Se A è un insieme di numeri naturali e k è un numero naturale più grande di 1, sia

$$B_k(A) = \{w | \, w \text{ è la rappresentazione in base } k \text{ di un numero in } A\}.$$

In questo caso non permettiamo 0 iniziali nella rappresentazione di un numero. Per esempio, $B_2(\{3, 5\}) = \{11, 101\}$ e $B_3(\{3, 5\}) = \{10, 12\}$. Dare un esempio di un insieme A per tale che $B_2(A)$ sia regolare ma $B_3(A)$ non sia regolare. Provare che l'esempio funziona.

***1.64** Per un linguaggio A, sia $A_{\frac{1}{2}-}$ l'insieme delle prime metà delle stringhe in A, ossia

$$A_{\frac{1}{2}-} = \{x | \text{ per qualche } y, \, |x| = |y| \text{ e } xy \in A\}.$$

Mostrare che se A è regolare, allora lo è anche $A_{\frac{1}{2}-}$.

***1.65** Per un linguaggio A, sia $A_{\frac{1}{3}-\frac{1}{3}}$ l'insieme di tutte le stringhe in A in cui sia stata eliminato il loro terzo medio (cioè la parte tra la prima e l'ultima della stringa divisa in tre parti uguali), ossia

$$A_{\frac{1}{3}-\frac{1}{3}} = \{xz | \text{ per qualche } y, \, |x| = |y| = |z| \text{ e } xyz \in A\}.$$

Mostrare che se A è regolare, allora $A_{\frac{1}{3}-\frac{1}{3}}$ non è necessariamente regolare.

1.66** Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA e sia h uno stato di M. Una ***sequenza sincronizzante per M e h è una stringa $s \in \Sigma^*$ tale che $\delta(q, s) = h$ per ogni $q \in Q$. (Qui abbiamo esteso δ alle stringhe, per cui $\delta(q, s)$ è lo stato in cui M si trova alla fine quando M inizia nello stato q e legge l'input s.) Diciamo che M è ***sincronizzabile*** se ha una sequenza sincronizzante per qualche stato h. Provare che se M è un DFA sincronizzabile con k stati, allora ha una sequenza sincronizzante di lunghezza al più k^3. Riesci a migliorare questo limite?

1.67 Definiamo l'operazione ***evita*** per linguaggi A e B nel modo seguente

$$A \textit{ evita } B = \{w | \, w \in A \text{ e } w \text{ non ha alcuna sottostringa che appartiene a } B\}.$$

Provare che la classe dei linguaggi regolari è chiusa rispetto all'operazione *evita*.

1.68 Sia $\Sigma = \{0,1\}$.

a. Sia $A = \{0^k u 0^k | \, k \geq 1 \text{ e } u \in \Sigma^*\}$. Mostrare che A è regolare.

b. Sia $B = \{0^k 1 u 0^k | \, k \geq 1 \text{ e } u \in \Sigma^*\}$. Mostrare che B non è regolare.

1.69 Siano M_1 ed M_2 due DFA che hanno k_1 e k_2 stati, rispettivamente, e inoltre sia $U = L(M_1) \cup L(M_2)$.

a. Mostrare che se $U \neq \emptyset$, allora U contiene qualche stringa s, tale che $|s| < \max(k_1, k_2)$.

b. Mostrare che se $U \neq \Sigma^*$, allora U esclude qualche stringa s, tale che $|s| < k_1 k_2$.

1.70 Sia $\Sigma = \{0,1,\#\}$. Sia $C = \{x\#x^{\mathcal{R}}\#x | \, x \in \{0,1\}^*\}$. Mostrare che $\overline{C}$ è un CFL.

1.71 **a.** Sia $B = \{1^k y | \, y \in \{0,1\}^* \text{ e } y \text{ contiene almeno } k \text{ simboli uguali a } 1, \text{ per } k \geq 1\}$.
Mostrare che B è un linguaggio regolare.

b. Sia $C = \{1^k y | \, y \in \{0,1\}^* \text{ e } y \text{ contiene al più } k \text{ simboli uguali a } 1, \text{ per } k \geq 1\}$.
Mostrare che C non è un linguaggio regolare.

***1.72** Nel metodo tradizionale per tagliare un mazzo di carte da gioco, il mazzo è arbitrariamente diviso in due parti, che sono scambiate prima di ricostituire il mazzo. In un taglio più complesso, chiamato taglio di Scarne, il mazzo è diviso in tre parti e la parte centrale è posta per prima nel ricomporlo. Ci ispireremo al taglio di Scarne per definire un'operazione sui linguaggi. Per un linguaggio A, sia $CUT(A) = \{yxz | \, xyz \in A\}$.

a. Esibire un linguaggio B per il quale $CUT(B) \neq CUT(CUT(B))$.

b. Mostrare che la classe dei linguaggi regolari è chiusa rispetto a CUT.

1.73 Sia $\Sigma = \{0,1\}$. Sia $WW_k = \{ww | \, w \in \Sigma^* \text{ e } w \text{ ha lunghezza } k\}$.

a. Mostrare che per ogni k, nessun DFA può riconoscere WW_k con meno di 2^k stati.

b. Descrivere un NFA molto più piccolo per $\overline{WW_k}$, il complemento di WW_k.

SOLUZIONI SELEZIONATE

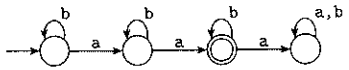
1.1 Per M_1: **(a)** q_1; **(b)** $\{q_2\}$; **(c)** q_1, q_2, q_3, q_1, q_1; **(d)** No; **(e)** No
Per M_2: **(a)** q_1; **(b)** $\{q_1, q_4\}$; **(c)** q_1, q_1, q_1, q_2, q_4; **(d)** Sì; **(e)** Sì

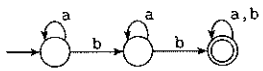
1.2 $M_1 = (\{q_1, q_2, q_3\}, \{\mathtt{a}, \mathtt{b}\}, \delta_1, q_1, \{q_2\})$.
$M_2 = (\{q_1, q_2, q_3, q_4\}, \{\mathtt{a}, \mathtt{b}\}, \delta_2, q_1, \{q_1, q_4\})$.
Le funzioni di transizione sono

δ_1	a	b
q_1	q_2	q_1
q_2	q_3	q_3
q_3	q_2	q_1

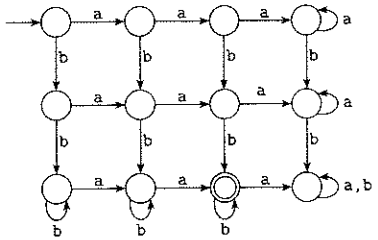
δ_2	a	b
q_1	q_1	q_2
q_2	q_3	q_4
q_3	q_2	q_1
q_4	q_3	q_4.

1.4 **(b)** Di seguito sono riportati i DFA per i due linguaggi $\{w | \, w$ ha esattamente due $\mathtt{a}\}$ e $\{w | \, w$ ha almeno due $\mathtt{b}\}$.

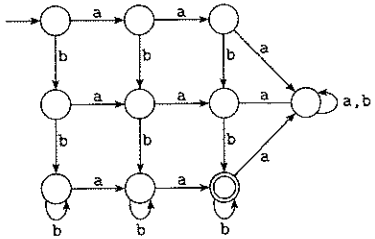




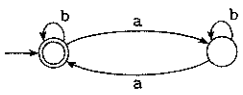
Comporli usando la costruzione per l'intersezione dà il seguente DFA.

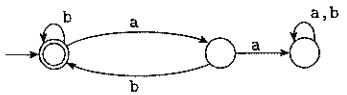


Sebbene il problema non richieda che tu semplifichi il DFA, alcuni stati possono essere fusi dando luogo al seguente DFA.

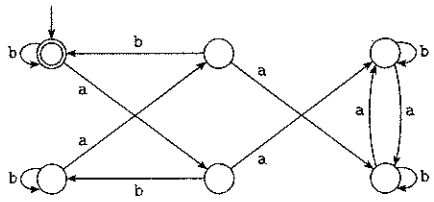


(d) Questi sono i DFA per i due linguaggi $\{w|\, w$ ha un numero pari di a$\}$ e $\{w|\,$ogni a in w è seguita da almeno una b$\}$.

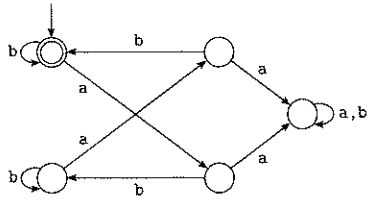




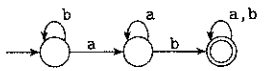
Comporli usando la costruzione per l'intersezione dà il seguente DFA.

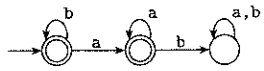


Sebbene il problema non richieda che tu semplifichi il DFA, alcuni stati possono essere fusi dando luogo al seguente DFA.

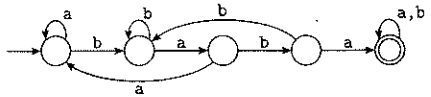


1.5 **(a)** Il DFA sul lato sinistro riconosce $\{w|\, w$ contiene ab$\}$. Il DFA sul lato destro riconosce il suo complemento, $\{w|\, w$ non contiene ab$\}$.

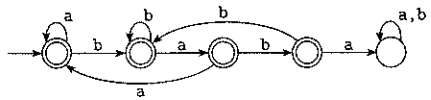




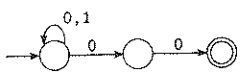
(b) Questo DFA riconosce $\{w|\, w$ contiene baba$\}$.



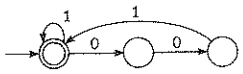
Questo DFA riconosce $\{w|\, w$ non contiene baba$\}$.



1.7 (a)



(f)



1.11 Sia $N = (Q, \Sigma, \delta, q_0, F)$ un NFA. Costruire un NFA N' con un solo stato accettante che riconosce lo stesso linguaggio di N. Informalmente, N' è esattamente come N tranne che ha ε-transizioni dagli stati corrispondenti agli stati accettanti di N, a un nuovo stato accettante, q_{accept}. Lo stato q_{accept} non ha transizioni uscenti. Più formalmente, $N' = (Q \cup \{q_{\text{accept}}\}, \Sigma, \delta', q_0, \{q_{\text{accept}}\})$, dove per ogni $q \in Q$ e $a \in \Sigma_\varepsilon$

$$\delta'(q,a) = \begin{cases} \delta(q,a) & \text{se } a \neq \varepsilon \text{ o } q \notin F \\ \delta(q,a) \cup \{q_{\text{accept}}\} & \text{se } a = \varepsilon \text{ e } q \in F \end{cases}$$

e $\delta'(q_{\text{accept}}, a) = \emptyset$ per ogni $a \in \Sigma_\varepsilon$.

1.23 Proviamo entrambe le direzioni del "se e solo se."
($\rightarrow$) Assumiamo che $B = B^+$ e mostriamo che $BB \subseteq B$.
Per ogni linguaggio risulta $BB \subseteq B^+$, quindi se $B = B^+$, allora $BB \subseteq B$.
($\leftarrow$) Assumiamo che $BB \subseteq B$ e mostriamo che $B = B^+$.
Per ogni linguaggio $B \subseteq B^+$, noi dobbiamo quindi solo mostrare $B^+ \subseteq B$. Se $w \in B^+$, allora $w = x_1 x_2 \cdots x_k$ dove ciascun $x_i \in B$ e $k \geq 1$. Poiché $x_1, x_2 \in B$ e $BB \subseteq B$, abbiamo $x_1 x_2 \in B$. Analogamente, poiché $x_1 x_2$ è in B e x_3 è in B, abbiamo $x_1 x_2 x_3 \in B$. Continuando in questo modo, $x_1 \cdots x_k \in B$. Quindi $w \in B$, e così possiamo concludere che $B^+ \subseteq B$.

L'ultimo ragionamento può essere formalizzato con la seguente prova per induzione.
Assumiamo che $BB \subseteq B$.
Enunciato: Per ogni $k \geq 1$, se $x_1, \ldots, x_k \in B$, allora $x_1 \cdots x_k \in B$.
Base: Provarlo per $k = 1$. In questo caso l'enunciato è ovviamente vero.
Passo induttivo: Per ogni $k \geq 1$, si assuma che l'enunciato sia vero per k e si mostri che è vero per $k+1$.
Se $x_1, \ldots, x_k, x_{k+1} \in B$, allora per l'ipotesi induttiva, $x_1 \cdots x_k \in B$. Quindi, $x_1 \cdots x_k x_{k+1} \in BB$, ma $BB \subseteq B$, dunque $x_1 \cdots x_{k+1} \in B$. Questo prova il passo induttivo e l'enunciato. Questo enunciato implica che se $BB \subseteq B$, allora $B^+ \subseteq B$.

1.29 (a) Assumiamo che $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$ sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Scegliamo s uguale alla stringa $0^p 1^p 2^p$. Poiché s è un elemento di A_1 ed s è più lunga di p, il pumping lemma garantisce che s può essere divisa in tre parti, $s = xyz$, tali che per ogni $i \geq 0$ la stringa $xy^i z$ è in A_1. Consideriamo le due eventualità:

1. La stringa y consiste solo di simboli uguali 0, solo di simboli uguali a 1, o solo di simboli uguali a 2. In ognuno di questi casi, la stringa $xyyz$ non avrà lo stesso numero di simboli uguali a 0, uguali a 1 e uguali a 2. Quindi $xyyz$ non è un elemento di A_1, una contraddizione.
2. La stringa y non consiste di un solo tipo di simbolo. In questo caso, $xyyz$ avrà i simboli uguali a 0, a 1, o a 2 non nel corretto ordine. Quindi $xyyz$ non è un elemento di A_1, una contraddizione.

In ogni caso giungiamo a una contraddizione. Quindi, A_1 non è regolare.

(c) Assumiamo che $A_3 = \{\mathtt{a}^{2^n} \mid n \geq 0\}$ sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Sia s la stringa $\mathtt{a}^{2^p}$. Poiché s è un elemento di A_3 ed s è più lunga di p, il pumping lemma garantisce che s può essere divisa in tre parti, $s = xyz$, che verificano le tre condizioni del pumping lemma.

La terza condizione ci dice che $|xy| \leq p$. Inoltre, $p < 2^p$ e quindi $|y| < 2^p$. Quindi, $|xyyz| = |xyz| + |y| < 2^p + 2^p = 2^{p+1}$. La seconda condizione impone $|y| > 0$ quindi $2^p < |xyyz| < 2^{p+1}$. La lunghezza di $xyyz$ non può essere una potenza di 2. Quindi $xyyz$ non è un elemento di A_3, una contraddizione. Quindi, A_3 non è regolare.

1.34 Siano $M_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$ ed $M_C = (Q_C, \Sigma, \delta_C, q_C, F_C)$ due DFA che riconoscono B e C, rispettivamente. Costruiamo un NFA $M = (Q, \Sigma, \delta, q_0, F)$ che riconosce $B \overset{1}{\leftarrow} C$ come segue. Per decidere se il suo input w è in $B \overset{1}{\leftarrow} C$, la macchina M verifica che $w \in B$, e in parallelo non deterministicamente ipotizza una stringa y che contiene lo stesso numero di simboli uguali a 1 di w e verifica che $y \in C$.

1. $Q = Q_B \times Q_C$.
2. Per $(q,r) \in Q$ e $a \in \Sigma_\varepsilon$, definiamo

$$\delta((q,r),a) = \begin{cases} \{(\delta_B(q,0), r)\} & \text{se } a = 0 \\ \{(\delta_B(q,1), \delta_C(r,1))\} & \text{se } a = 1 \\ \{(q, \delta_C(r,0))\} & \text{se } a = \varepsilon. \end{cases}$$

3. $q_0 = (q_B, q_C)$.
4. $F = F_B \times F_C$.

1.45 (a) Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA che riconosce A, dove A è un linguaggio regolare. Costruiamo $M' = (Q', \Sigma, \delta', q_0', F')$ che riconosce $NOPREFIX(A)$ come segue:

1. $Q' = Q$.
2. Per $r \in Q'$ e $a \in \Sigma$, definiamo $\delta'(r,a) = \begin{cases} \{\delta(r,a)\} & \text{se } r \notin F \\ \emptyset & \text{se } r \in F. \end{cases}$
3. $q_0' = q_0$.
4. $F' = F$.

1.46 Assumiamo al contrario che qualche FST T dia in uscita $w^{\mathcal{R}}$ sull'input w. Consideriamo le stringhe input 00 e 01. Sull'input 00, T deve dare in uscita 00, e sull'input 01, T deve dare in uscita 10. In entrambi i casi, il primo bit di input è uno 0 ma i primi bit di uscita sono diversi. Operare in questo modo è impossibile per un FST poiché esso produce il suo primo bit di uscita prima che abbia letto il suo secondo input. Quindi un tale FST non può esistere.

1.48 (a) Proviamo questa asserzione con una prova per assurdo. Sia M un DFA a k stati che riconosce L. Supponiamo per assurdo che L abbia un indice maggiore di k. Questo significa che qualche insieme X con più di k elementi è distinguibile a coppie da L. Poiché M ha k stati, il principio della piccionaia implica che X contiene due diverse stringhe x e y, tali che $\delta(q_0, x) = \delta(q_0, y)$. Qui $\delta(q_0, x)$ è lo stato in cui si trova M dopo che, partendo dallo stato q_0, ha letto la stringa di input x. Allora, per ogni stringa $z \in \Sigma^*$, $\delta(q_0, xz) = \delta(q_0, yz)$. Quindi, o entrambe xz e yz sono in L o nessuna delle due è in L. Ma allora x e y non sarebbero distinguibili da L, contraddicendo la nostra ipotesi che X sia distinguibile a coppie da L.

(b) Sia $X = \{s_1, \ldots, s_k\}$ distinguibile a coppie da L. Costruiamo un DFA $M = (Q, \Sigma, \delta, q_0, F)$ con k stati che riconosce L. Sia $Q = \{q_1, \ldots, q_k\}$, e definiamo $\delta(q_i, a)$ uguale allo stato q_j tale che $s_j \equiv_L s_i a$ (la relazione $\equiv_L$ è definita nel Problema 1.47). Si osservi che $s_j \equiv_L s_i a$ per qualche $s_j \in X$; altrimenti, $X \cup s_i a$ avrebbe $k+1$ elementi e sarebbe distinguibile a coppie da L, il che contraddirebbe l'assunzione che L abbia indice k. Sia $F = \{q_i \mid s_i \in L\}$. Sia lo stato iniziale q_0 uguale a q_i tale che $s_i \equiv_L \varepsilon$. M è costruito in modo che per ogni stato q_i,

$\{s|\ \delta(q_0, s) = q_i\} = \{s|\ s \equiv_L s_i\}$. Quindi M riconosce L.

(c) Supponiamo che L sia regolare e sia k il numero di stati in un DFA che riconosce L. Allora dal punto (a), L ha indice al più k. Viceversa, se L ha indice k, allora per il punto (b) è riconosciuto da un DFA con k stati e quindi è regolare. Per mostrare che l'indice di L è la dimensione del più piccolo DFA che lo accetta, supponiamo che l'indice di L sia *esattamente* k. Allora, per il punto (b), esiste un DFA con k stati che accetta L. Questo è il più piccolo di tali DFA perché se ce ne fosse uno più piccolo, allora potremmo mostrare mediante il punto (a) che l'indice di L è minore di k.

1.50 **(a)** La lunghezza minima del pumping è 4. La stringa 000 è nel linguaggio ma non può essere iterata, quindi 3 non è una lunghezza del pumping per questo linguaggio. Se s ha lunghezza maggiore o uguale a 4, contiene dei simboli uguali a 1. Dividendo s in xyz, dove x è 000 e y è il primo 1 e z è tutto quello che segue, le tre condizioni del pumping lemma sono soddisfatte.

(b) La lunghezza minima del pumping è 1. La lunghezza del pumping non può essere 0 poiché la stringa ε è nel linguaggio e non può essere iterata. Ogni stringa non vuota nel linguaggio può essere divisa in xyz, dove x, y, e z sono ε, il primo simbolo, e il resto, rispettivamente. Questa divisione soddisfa le tre condizioni.

(d) La lunghezza minima del pumping è 3. La lunghezza del pumping non può essere 2 poiché la stringa 11 è nel linguaggio e non può essere iterata. Sia s una stringa nel linguaggio di lunghezza almeno 3. Se s è generata da $0^*1^+0^+1^*$ ed s inizia o con 0 o con 11, scriviamo $s = xyz$ dove $x = \varepsilon$, y è il primo simbolo, e z è il resto di s. Se s è generata da $0^*1^+0^+1^*$ ed s inizia con 10, scriviamo $s = xyz$ dove $x = 10$, y è il simbolo seguente, e z è il resto di s. Queste divisioni di s mostrano che essa può essere iterata. Se s è generata da 10^*1, possiamo scriverla come xyz dove $x = 1$, $y = 0$, e z è il resto di s. Questa divisione fornisce un modo per iterare s.

1.51 **(b)** Sia $B = \{0^m1^n|\ m \neq n\}$. Osserviamo che $\overline{B} \cap 0^*1^* = \{0^k1^k|\ k \geq 0\}$. Se B fosse regolare, allora $\overline{B}$ sarebbe regolare e ugualmente lo sarebbe $\overline{B} \cap 0^*1^*$. Ma già sappiamo che $\{0^k1^k|\ k \geq 0\}$ non è regolare, quindi B non può essere regolare.

Alternativamente, possiamo provare che B non è regolare usando il pumping lemma direttamente, sebbene far così sia più difficile. Assumiamo che $B = \{0^m1^n|\ m \neq n\}$ sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Si osservi che $p!$ è divisibile per tutti gli interi da 1 a p, dove $p! = p(p-1)(p-2)\cdots 1$. La stringa $s = 0^p1^{p+p!} \in B$, ed $|s| \geq p$. Allora il pumping lemma implica che s può essere divisa in xyz con $x = 0^a$, $y = 0^b$, e $z = 0^c1^{p+p!}$, dove $b \geq 1$ e $a + b + c = p$. Sia s' la stringa $xy^{i+1}z$, dove $i = p!/b$. Allora $y^i = 0^{p!}$ quindi $y^{i+1} = 0^{b+p!}$, e quindi $s' = 0^{a+b+c+p!}1^{p+p!}$. Questo dà $s' = 0^{p+p!}1^{p+p!} \notin B$, una contraddizione.

2

LINGUAGGI CONTEXT-FREE

Nel Capitolo 1 abbiamo introdotto due diversi, ma equivalenti, metodi per descrivere linguaggi: gli *automi finiti* e le *espressioni regolari*. Abbiamo mostrato che molti linguaggi possono essere descritti in questo modo ma che questo non è possibile per alcuni semplici linguaggi, come $\{0^n1^n|n \geq 0\}$.

In questo capitolo presentiamo le ***grammatiche context-free***, un metodo più potente per descrivere linguaggi. Queste grammatiche possono descrivere alcuni aspetti che hanno una struttura ricorsiva. Ciò le rende utili in una varietà di applicazioni.

Inizialmente le grammatiche context-free furono usate nello studio dei linguaggi naturali. Un modo per capire le relazioni tra termini quali *nome*, *verbo*, *preposizione* e i loro rispettivi sintagmi, conduce a una ricorsione naturale poiché sintagmi nominali possono apparire all'interno di sintagmi verbali e viceversa. Le grammatiche context-free aiutano a organizzare e capire queste relazioni.

Un'importante applicazione delle grammatiche context-free si presenta nella specifica e nella compilazione dei linguaggi di programmazione. Spesso una grammatica per un linguaggio di programmazione costituisce una referenza per coloro che provano ad apprendere la sintassi del linguaggio. Progettisti di compilatori e interpreti per linguaggi di programmazione spesso iniziano con l'ottenere una grammatica per il linguaggio. La maggior parte dei compilatori e degli interpreti contiene una componente chiamata ***parser*** che estrae il significato di un programma prima di generare il codice compilato o di eseguire il programma interpretato. Diversi metodi

facilitano la costruzione di un parser quando è disponibile una grammatica context-free. Alcuni strumenti generano perfino automaticamente il parser dalla grammatica.

I linguaggi associati alle grammatiche context-free sono chiamati ***linguaggi context-free***. La classe dei linguaggi context-free include tutti i linguaggi regolari e molti ulteriori linguaggi. In questo capitolo, diamo una definizione delle grammatiche context-free e studiamo le proprietà dei linguaggi context-free. Introduciamo anche gli ***automi a pila*** (pushdown automata), una classe di macchine che riconoscono i linguaggi context-free. Gli automi a pila sono utili perché permettono di acquisire una maggiore comprensione del potere delle grammatiche context-free.

2.1
GRAMMATICHE CONTEXT-FREE

Un esempio di grammatica context-free, che chiamiamo G_1, è il seguente

$$\begin{aligned} A &\to 0A1 \\ A &\to B \\ B &\to \# \end{aligned}$$

Una grammatica consiste di un insieme di ***regole di sostituzione***, anche chiamate ***produzioni***. Ogni regola appare come una linea nella grammatica, costituita da un simbolo e una stringa separati da una freccia. Il simbolo è chiamato una ***variabile***. La stringa consiste di variabili e altri simboli chiamati ***terminali***. Le variabili sono spesso rappresentate da lettere maiuscole. I terminali sono analoghi ai simboli dell'alfabeto di input e sono spesso rappresentati da lettere minuscole, numeri o simboli speciali. Una delle variabili è chiamata la ***variabile iniziale***. Essa generalmente si trova sul lato sinistro della regola più in alto. Per esempio, la grammatica G_1 ha tre regole. Le variabili di G_1 sono A e B e A è la variabile iniziale. I suoi terminali sono `0`, `1` e `#`.

Una grammatica può essere usata per descrivere un linguaggio generando ogni stringa del linguaggio nel seguente modo.

1. Scrivi la variabile iniziale. È la variabile sul lato sinistro della regola più in alto, a meno che non sia diversamente specificato.
2. Trova una variabile che è stata scritta e una regola che inizia con quella variabile. Sostituisci la variabile scritta con il lato destro di quella regola.
3. Ripeti il passo 2 fino a quando non vi sono più variabili.

Per esempio, la grammatica G_1 genera la stringa `000#111`. Una sequenza di sostituzioni per ottenere una stringa è chiamata una ***derivazione***. Una derivazione della stringa `000#111` nella grammatica G_1 è

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111.$$

La stessa informazione può essere anche rappresentata mediante un disegno con un ***albero sintattico*** (*parse tree*). Un esempio di albero sintattico è mostrato in Figura 2.1.

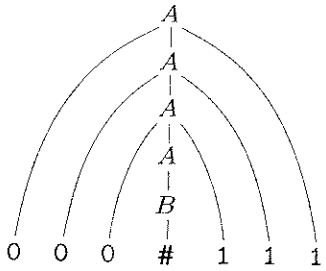


FIGURA 2.1
Albero sintattico per `000#111` nella grammatica G_1

Tutte le stringhe generate in questo modo costituiscono il ***linguaggio della grammatica***. Denoteremo con $L(G_1)$ il linguaggio della grammatica G_1. Alcune prove con la grammatica G_1 mostrano che $L(G_1)$ è $\{0^n\#1^n \mid n \geq 0\}$. Ogni linguaggio che può essere generato da una grammatica context-free è chiamato un ***linguaggio context-free*** (CFL). Nel presentare una grammatica context-free, è conveniente abbreviare le diverse regole con la stessa variabile a sinistra, come $A \to 0A1$ e $A \to B$, in una singola linea $A \to 0A1 \mid B$, usando il simbolo " | " come un "oppure".

Quello che segue è un secondo esempio di grammatica context-free, chiamata G_2, che descrive un frammento della lingua inglese.

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → `a` | `the`
⟨NOUN⟩ → `boy` | `girl` | `flower`
⟨VERB⟩ → `touches` | `likes` | `sees`
⟨PREP⟩ → `with`

La grammatica G_2 ha 10 variabili (i termini grammaticali scritti in lettere maiuscole e tra parentesi); 27 terminali (i simboli dell'alfabeto inglese standard più un carattere spazio); e 18 regole. Tra le stringhe di $L(G_2)$ vi sono:

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

Ognuna di queste stringhe ha una derivazione nella grammatica G_2. Una derivazione della prima stringa su questa lista è fornita di seguito.

$$\begin{aligned}
\langle \text{SENTENCE} \rangle &\Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a}\ \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{CMPLX-VERB} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{VERB} \rangle \\
&\Rightarrow \texttt{a boy sees}
\end{aligned}$$

Definizione formale di grammatica context-free

Formalizziamo la nostra nozione di grammatica context-free (CFG).

DEFINIZIONE 2.2

Una ***grammatica context-free*** è una quadrupla (V, Σ, R, S), dove

1. V è un insieme finito i cui elementi sono chiamati ***variabili***,
2. Σ è un insieme finito, disgiunto da V, i cui elementi sono chiamati ***terminali***,
3. R è un insieme finito di ***regole***, dove ciascuna regola è una variabile e una stringa di variabili e terminali, ed
4. $S \in V$ è la variabile iniziale.

Se u, v e w sono stringhe di variabili e terminali e $A \to w$ è una regola della grammatica, diciamo che uAv ***produce*** uwv, e lo denotiamo con $uAv \Rightarrow uwv$. Diciamo che u ***deriva*** v, e lo denotiamo con $u \overset{*}{\Rightarrow} v$, se $u = v$ o se esiste una sequenza $u_1, u_2, \ldots, u_k$, con $k \geq 0$ e

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

Il ***linguaggio della grammatica*** è $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

Nella grammatica G_1, $V = \{A, B\}$, $\Sigma = \{0, 1, \#\}$, $S = A$ ed R è l'insieme delle tre regole che compaiono a pagina 104. Nella grammatica G_2,

$$\begin{aligned}
V = \{&\langle \text{SENTENCE} \rangle, \langle \text{NOUN-PHRASE} \rangle, \langle \text{VERB-PHRASE} \rangle, \\
&\langle \text{PREP-PHRASE} \rangle, \langle \text{CMPLX-NOUN} \rangle, \langle \text{CMPLX-VERB} \rangle, \\
&\langle \text{ARTICLE} \rangle, \langle \text{NOUN} \rangle, \langle \text{VERB} \rangle, \langle \text{PREP} \rangle\},
\end{aligned}$$

e $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}, \text{“ ”}\}$. Il simbolo “ ” è il simbolo vuoto (blank), posto, senza essere visto, dopo ciascuna parola (a, boy, ecc.), in modo che le parole non siano unite.

Spesso specifichiamo una grammatica scrivendo solo le sue regole. Possiamo identificare le variabili nei simboli che compaiono sul lato sinistro delle regole e i terminali nei rimanenti simboli. Per convenzione, la variabile iniziale è la variabile sul lato sinistro della prima regola.

Esempi di grammatiche context-free

ESEMPIO 2.3

Consideriamo la grammatica $G_3 = (\{S\}, \{\texttt{a}, \texttt{b}\}, R, S)$. L'insieme delle regole, R, è

$$S \to \texttt{a}S\texttt{b} \mid SS \mid \varepsilon.$$

Questa grammatica genera stringhe come abab, aaabbb, e aababb. Si può più facilmente vedere qual è questo linguaggio pensando ad a come una parentesi aperta “(” e a b come una parentesi chiusa “)”. Visto in questo modo, $L(G_3)$ è il linguaggio di tutte le stringhe di parentesi correttamente annidate. Si osservi che il lato destro di una regola può essere la parola vuota ε.

ESEMPIO 2.4

Si consideri la grammatica $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V è $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ e Σ è $\{\texttt{a}, \texttt{+}, \texttt{x}, \texttt{(}, \texttt{)}\}$. Le regole sono

$$\begin{aligned}
\langle \text{EXPR} \rangle &\to \langle \text{EXPR} \rangle \texttt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\
\langle \text{TERM} \rangle &\to \langle \text{TERM} \rangle \texttt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\
\langle \text{FACTOR} \rangle &\to \texttt{(}\ \langle \text{EXPR} \rangle\ \texttt{)} \mid \texttt{a}
\end{aligned}$$

Le due stringhe a+axa e (a+a)xa possono essere generate dalla grammatica G_4. Gli alberi sintattici sono mostrati nella figura seguente.

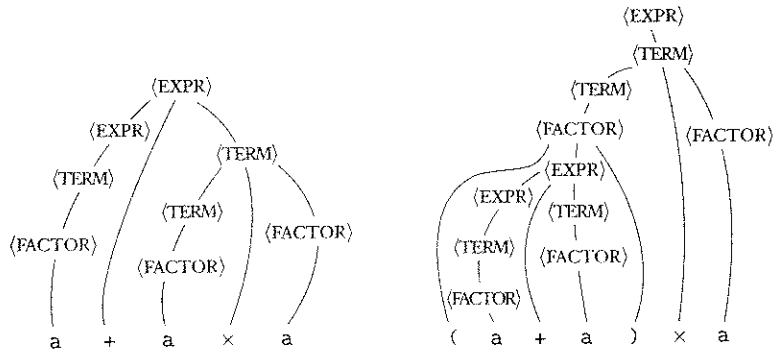


FIGURA 2.5
Alberi sintattici per le stringhe a+axa e (a+a)xa

Un compilatore traduce codice scritto in un linguaggio di programmazione in un'altra forma, solitamente una più adatta per l'esecuzione. Per farlo, il compilatore estrae il significato del codice da compilare in un processo chiamato ***parsing***. Una rappresentazione di questo significato è l'albero sintattico per il codice, nella grammatica context-free per il linguaggio di programmazione. In seguito discuteremo un algoritmo che fa il parsing dei linguaggi context-free nel Teorema 7.16 e nel Problema 7.22.

La grammatica G_4 descrive un frammento di un linguaggio di programmazione che riguarda le espressioni aritmetiche. Si osservi come gli alberi sintattici nella Figura 2.5 "raggruppino" le operazioni. L'albero per a+axa raggruppa l'operatore x e i suoi operandi (le ultime due a) in un operando dell'operatore +. Nell'albero per (a+a)xa, il raggruppamento è invertito. Questi raggruppamenti corrispondono alla precedenza standard della moltiplicazione rispetto all'addizione e all'uso delle parentesi per scavalcare questa precedenza. La grammatica G_4 è progettata per tenere conto di queste regole di precedenza.

Progettare grammatiche context-free

Come con la progettazione di automi finiti, discussa nella Sezione 1.1 (pagina 43), la progettazione di grammatiche context-free richiede creatività. Anzi, le grammatiche context-free sono perfino più complesse da costruire degli automi finiti poiché noi siamo più abituati a programmare una macchina per compiti specifici che a descrivere linguaggi con grammatiche. Le tecniche seguenti sono utili, singolarmente o combinate, quando affrontiamo il problema di costruire una CFG.

Innanzitutto, molti CFL sono l'unione di CFL più semplici. Se devi costruire una CFG per un CFL che puoi dividere in componenti più semplici, fallo e poi costruisci grammatiche separate per ciascuna componente. Queste singole grammatiche possono facilmente essere fuse in una grammatica per il linguaggio iniziale unendo le loro regole e poi aggiungendo la nuova regola $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, dove le variabili S_i sono le variabili iniziali per le grammatiche individuali. Risolvere diversi problemi più semplici è spesso più facile che risolvere un problema complicato.

Per esempio, per ottenere una grammatica per il linguaggio $\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$, costruiamo prima la grammatica

$$S_1 \to 0S_11 \mid \varepsilon$$

per il linguaggio $\{0^n1^n \mid n \geq 0\}$ e la grammatica

$$S_2 \to 1S_20 \mid \varepsilon$$

per il linguaggio $\{1^n0^n \mid n \geq 0\}$ e poi aggiungiamo la regola $S \to S_1 \mid S_2$ ottenendo la grammatica

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to 0S_11 \mid \varepsilon \\ S_2 &\to 1S_20 \mid \varepsilon. \end{aligned}$$

In secondo luogo, costruire una CFG per un linguaggio che sia regolare è facile se si può prima costruire un DFA per quel linguaggio. Si può trasformare un DFA in una CFG equivalente nel modo seguente. Si introduca una variabile R_i per ogni stato q_i del DFA. Si aggiunga la regola $R_i \to aR_j$ alla CFG se $\delta(q_i, a) = q_j$ è una transizione nel DFA. Si aggiunga la regola $R_i \to \varepsilon$ se q_i è uno stato accettante del DFA. Si assuma R_0 come variabile iniziale della grammatica, dove q_0 è lo stato iniziale della macchina. Si può verificare che la CFG risultante genera lo stesso linguaggio di quello riconosciuto dal DFA.

Inoltre, alcuni linguaggi context-free contengono stringhe con due sottostringhe che sono "collegate", nel senso che una macchina per un tale linguaggio avrebbe bisogno di ricordare una quantità non limitata di informazione su una delle sottostringhe per verificare che essa corrisponde correttamente all'altra sottostringa. Questa situazione si verifica nel linguaggio $\{0^n1^n \mid n \geq 0\}$ poiché una macchina avrebbe bisogno di ricordare il numero di simboli uguali a 0 per verificare che esso è uguale al numero di simboli uguali a 1. Si può costruire una CFG per gestire questa situazione usando una regola della forma $R \to uRv$, che genera stringhe nelle quali la porzione contenente le u corrisponde alla porzione contenente le v.

Infine, in linguaggi più complessi, le stringhe possono contenere alcune strutture che compaiono ricorsivamente come parte di altre strutture (o

della stessa). Questa situazione si verifica nella grammatica che genera le espressioni aritmetiche nell'Esempio 2.4. Ogni volta che compare il simbolo a, al suo posto potrebbe invece comparire ricorsivamente un'intera espressione parentesizzata. Per ottenere questo effetto, si ponga nelle regole il simbolo di variabile che genera la struttura in questione, laddove la struttura può apparire ricorsivamente.

Ambiguità

Qualche volta una grammatica può generare la stessa stringa in più modi diversi. Una tale stringa avrà diversi alberi sintattici e quindi diversi significati. Questo risultato può essere indesiderabile per alcune applicazioni, come i linguaggi di programmazione, dove un programma dovrebbe avere un'unica interpretazione.

Se una grammatica genera la stessa stringa in più modi diversi, diciamo che la stringa è derivata *ambiguamente* in quella grammatica. Se una grammatica genera alcune stringhe ambiguamente, diciamo che la grammatica è *ambigua*.

Per esempio, consideriamo la grammatica G_5:

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle \texttt{+} \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \texttt{x} \langle \text{EXPR} \rangle \mid \texttt{(} \langle \text{EXPR} \rangle \texttt{)} \mid \texttt{a}$$

Questa grammatica genera la stringa `a+axa` ambiguamente. La figura seguente mostra i due diversi alberi sintattici.

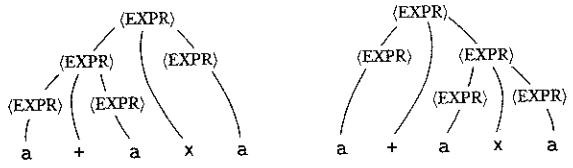


FIGURA 2.6
I due alberi sintattici per la stringa a+axa nella grammatica G_5

Questa grammatica non tiene conto delle usuali regole di precedenza e quindi può raggruppare + prima di × o viceversa. Al contrario, la grammatica G_4 genera esattamente lo stesso linguaggio, ma ogni stringa generata ha un solo albero sintattico. Quindi G_4 non è ambigua, mentre G_5 è ambigua.

La grammatica G_2 (pagina 106) è un altro esempio di grammatica ambigua. La frase `the girl touches the boy with the flower` ha due differenti derivazioni. Nell'Esercizio 2.8 ti viene richiesto di fornire i due alberi sintattici e osservare la loro corrispondenza con i due diversi modi di leggere quella frase.

Ora formalizziamo la nozione di ambiguità. Quando diciamo che una grammatica genera ambiguamente una stringa, intendiamo che la stringa

ha due diversi alberi sintattici, non due differenti derivazioni. Due derivazioni possono differire solo nell'ordine in cui esse sostituiscono le variabili ma non nella loro struttura complessiva. Per concentrarci sulla struttura, definiamo un tipo di derivazione che sostituisce le variabili con un ordine stabilito. Una derivazione di una stringa w in una grammatica G è una ***derivazione a sinistra*** (*leftmost derivation*) se a ogni passo la variabile sostituita è quella che si trova più a sinistra. La derivazione che precede la Definizione 2.2 (pagina 106) è una derivazione a sinistra.

DEFINIZIONE **2.7**

Una stringa w è derivata ***ambiguamente*** in una grammatica context-free G se essa ha due o più diverse derivazioni a sinistra. Una grammatica G è ***ambigua*** se essa genera qualche stringa ambiguamente.

Qualche volta quando abbiamo una grammatica ambigua, possiamo trovare una grammatica non ambigua che genera lo stesso linguaggio. Tuttavia, alcuni linguaggi context-free possono essere generati solo da grammatiche ambigue. Tali linguaggi sono chiamati ***inerentemente ambigui***. Il Problema 2.41 chiede di dimostrare che il linguaggio $\{\texttt{a}^i\texttt{b}^j\texttt{c}^k \mid i = j \text{ oppure } j = k\}$ è inerentemente ambiguo.

Forma normale di Chomsky

Quando si lavora con le grammatiche context-free, è spesso conveniente averle in forma semplificata. Una delle forme più semplici e utili è chiamata la forma normale di Chomsky. La forma normale di Chomsky è utile nel dare algoritmi che lavorano con grammatiche context-free, come facciamo nei Capitoli 4 e 7.

DEFINIZIONE **2.8**

Una grammatica context-free è in ***forma normale di Chomsky*** se ogni regola è della forma

$$A \rightarrow BC$$
$$A \rightarrow a$$

dove a è un terminale e A, B e C sono variabili qualsiasi – tranne che B e C non possono essere la variabile iniziale. Inoltre, permettiamo la regola $S \rightarrow \varepsilon$, dove S è la variabile iniziale.

TEOREMA 2.9

Ogni linguaggio context-free è generato da una grammatica context-free in forma normale di Chomsky.

IDEA. Possiamo trasformare ogni grammatica G in forma normale di Chomsky. La trasformazione ha diversi passi nei quali le regole che violano le condizioni sono rimpiazzate con regole equivalenti che sono adeguate. Innanzitutto, aggiungiamo una nuova variabile iniziale. Poi, eliminiamo tutte le ***ε-regole*** della forma $A \to \varepsilon$. Eliminiamo anche tutte le ***regole unitarie*** della forma $A \to B$. In entrambi i casi, modifichiamo la grammatica in modo da essere sicuri che generi ancora lo stesso linguaggio. Infine, trasformiamo le restanti regole nella forma adeguata.

DIMOSTRAZIONE In primo luogo, aggiungiamo una nuova variabile iniziale S_0 e la regola $S_0 \to S$, dove S era la variabile iniziale di partenza. Questo cambiamento garantisce che la variabile iniziale non compare sul lato destro di una regola.

In secondo luogo, ci occupiamo di tutte le ε-regole. Eliminiamo una ε-regola $A \to \varepsilon$, dove A non è la variabile iniziale. Poi, per ogni occorrenza di A sul lato destro di una regola, aggiungiamo una nuova regola con quell'occorrenza cancellata. In altre parole, se $R \to uAv$ è una regola in cui u e v sono stringhe di variabili e terminali, aggiungiamo la regola $R \to uv$. Facciamo questo per ogni *occorrenza* di A, in modo che la regola $R \to uAvAw$ faccia sì che vengano aggiunte $R \to uvAw$, $R \to uAvw$ ed $R \to uvw$. Se abbiamo la regola $R \to A$, aggiungiamo $R \to \varepsilon$ a meno che non avevamo precedentemente rimosso la regola $R \to \varepsilon$. Ripetiamo questi passi fino a eliminare tutte le ε-regole che non coinvolgono la variabile iniziale.

Inoltre, ci occupiamo delle regole unitarie. Eliminiamo una regola unitaria $A \to B$. Poi, per ogni regola $B \to u$, aggiungiamo la regola $A \to u$ a meno che questa non sia una regola unitaria precedentemente cancellata. Come prima, u è una stringa di variabili e terminali. Ripetiamo questi passi fino a eliminare tutte le regole unitarie.

Infine, trasformiamo tutte le restanti regole nella forma appropriata. Rimpiazziamo ogni regola $A \to u_1u_2\cdots u_k$, dove $k \geq 3$ e ciascun u_i è una variabile o un simbolo terminale, con le regole $A \to u_1A_1$, $A_1 \to u_2A_2$, $A_2 \to u_3A_3$, ... and $A_{k-2} \to u_{k-1}u_k$. Le A_i sono nuove variabili. Rimpiazziamo ogni terminale u_i nelle precedenti regole (o regola) con la nuova variabile U_i e aggiungiamo la regola $U_i \to u_i$.

ESEMPIO 2.10

Consideriamo la seguente CFG G_6 e trasformiamola in forma normale di Chomsky usando la procedura di conversione appena fornita. La sequenza di grammatiche presentate illustra i passi nella conversione. Le regole mostrate in grassetto sono quelle appena aggiunte. Le regole mostrate in grigio sono quelle appena rimosse.

1. La CFG iniziale G_6 è mostrata sulla sinistra. Il risultato dell'applicazione del primo passo, che crea una nuova variabile iniziale, è mostrato sulla destra.

$S \to ASA \mid \mathtt{a}B$
$A \to B \mid S$
$B \to \mathtt{b} \mid \varepsilon$

$\boldsymbol{S_0 \to S}$
$S \to ASA \mid \mathtt{a}B$
$A \to B \mid S$
$B \to \mathtt{b} \mid \varepsilon$

2. Eliminiamo le ε-regole $B \to \varepsilon$, come mostrato sulla sinistra, e $A \to \varepsilon$, come mostrato sulla destra.

$S_0 \to S$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a}$
$A \to B \mid S \mid \boldsymbol{\varepsilon}$
$B \to \mathtt{b} \mid \varepsilon$

$S_0 \to S$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S}$
$A \to B \mid S \mid \boldsymbol{\varepsilon}$
$B \to \mathtt{b}$

3a. Eliminiamo le regole unitarie $S \to S$, come mostrato sulla sinistra, e $S_0 \to S$, come mostrato sulla destra.

$S_0 \to S$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid S$
$A \to B \mid S$
$B \to \mathtt{b}$

$S_0 \to S \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$A \to B \mid S$
$B \to \mathtt{b}$

3b. Eliminiamo le regole unitarie $A \to B$ e $A \to S$.

$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$A \to B \mid S \mid \mathtt{b}$
$B \to \mathtt{b}$

$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$A \to S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$
$B \to \mathtt{b}$

4. Trasformiamo le rimanenti regole nella forma corretta, aggiungendo variabili e regole supplementari. La grammatica finale in forma normale di Chomsky è equivalente a G_6. (In realtà la procedura fornita nel Teorema 2.9

crea diverse variabili U_i e diverse regole $U_i \rightarrow$ a. Abbiamo semplificato la grammatica risultante usando una sola variabile U e una sola regola $U \rightarrow$ a.)

$$
\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow \mathtt{a} \\
B &\rightarrow \mathtt{b}
\end{aligned}
$$

2.2
AUTOMI A PILA

In questa sezione introduciamo un nuovo tipo di modello computazionale chiamato ***automa a pila*** (*pushdown automata*). Questi automi sono come gli automi finiti non deterministici ma hanno una componente in più chiamata ***pila*** (*stack*). La pila fornisce memoria aggiuntiva oltre alla quantità finita di essa disponibile nel controllo. La pila consente a tali automi di riconoscere alcuni linguaggi non regolari.

Gli automi a pila sono computazionalmente equivalenti alle grammatiche context-free. Questa equivalenza è utile poiché essa ci dà due alternative per provare che un linguaggio è context free. Possiamo dare una grammatica context-free che lo genera oppure un automa a pila che lo riconosce. Alcuni linguaggi sono più facilmente descritti in termini di generatori, mentre altri sono più facilmente descritti da riconoscitori.

La figura seguente è una rappresentazione schematica di un automa finito. Il controllo rappresenta gli stati e la funzione di transizione, il nastro contiene la stringa di input, e la freccia rappresenta la testina sull'input, che indica il successivo simbolo di input da leggere.

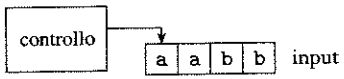


FIGURA 2.11
Schema di un automa finito

Aggiungendo l'elemento pila, otteniamo una rappresentazione schematica di un automa a pila, come mostrato nella figura seguente.

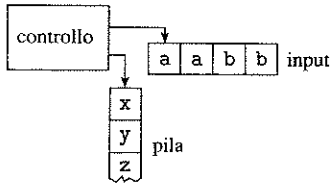


FIGURA 2.12
Schema di un automa a pila

Un automa a pila (PDA) può scrivere simboli nella pila e rileggerli in seguito. Scrivere un simbolo "spinge giù" tutti gli altri simboli nella pila. In un qualunque momento il simbolo sulla cima (*top*) della pila può essere letto e rimosso. I rimanenti simboli allora ritornano più in alto. L'operazione di scrivere un simbolo sulla pila è spesso chiamata ***push***, quella di eliminare un simbolo dalla pila è spesso chiamata ***pop***. Nota che ogni accesso alla pila, sia per leggere che per scrivere, può essere fatto solo sulla sommità. In altre parole, una pila è un dispositivo di memoria "last in, first out" (*ultimo entrato, primo uscito*). Se alcuni dati sono scritti nella pila e dati aggiuntivi sono scritti in seguito, i dati precedenti diventano inaccessibili fino a quando quelli successivi vengono rimossi.

I piatti sul bancone di servizio di un bar forniscono un esempio di pila. La pila di piatti si trova su una molla in modo che, quando un nuovo piatto è posto sulla sommità della pila, i piatti che sono sotto di esso, si spostano verso il basso. La pila di un automa è come una pila di piatti, dove ciascun piatto ha un simbolo scritto su di esso.

Una pila è molto importante perché può mantenere una quantità non limitata di informazioni. Ricordiamo che un automa finito non può riconoscere il linguaggio $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ poiché non può memorizzare numeri molto grandi nella sua memoria finita. Un PDA è in grado di riconoscere questo linguggio poiché può usare la sua pila per memorizzare il numero di simboli uguali a 0 che ha visto. Quindi, la natura non limitata di una pila permette al PDA di memorizzare numeri di dimensioni non limitate. La seguente descrizione informale mostra come opera l'automa per questo linguaggio.

> Legge i simboli in input. Scrive ciascuno 0 letto sulla pila. Non appena vede simboli uguali a 1, cancella uno 0 dalla pila per ogni 1 letto. Se la lettura dell'input termina esattamente quando la pila diventa priva

di simboli uguali a 0, accetta l'input. Se la pila si svuota ma restano simboli uguali a 1 o se i simboli uguali a 1 sono finiti ma la pila contiene ancora simboli uguali a 0 o se qualche 0 appare nell'input dopo i simboli uguali a 1, rifiuta l'input.

Come menzionato prima, gli automi a pila possono essere non deterministici. Automi a pila deterministici e non deterministici *non* sono computazionalmente equivalenti. Gli automi a pila non deterministici riconoscono alcuni linguaggi che nessun automa a pila deterministico può riconoscere, come vedremo nella Sezione 2.4. Esibiamo linguaggi che richiedono il non determinismo negli Esempi 2.16 e 2.18. Ricordiamo che gli automi finiti deterministici e non deterministici riconoscono la stessa classe di linguaggi, quindi la situatione per gli automi a pila è diversa. Ci concentreremo sugli automi a pila non deterministici perché questi automi sono computazionalmente equivalenti alle grammatiche context-free.

Definizione formale di automa a pila

La definizione di automa a pila è simile a quella di un automa finito, tranne che per la pila. La pila è un dispositivo che contiene simboli presi da un qualche alfabeto. La macchina può usare differenti alfabeti per il suo input e la sua pila, quindi ora specifichiamo sia un alfabeto di simboli di input Σ sia un alfabeto di pila Γ.

Nel cuore di ogni definizione di automa c'è la funzione di transizione, che descrive il suo comportamento. Ricordiamo che $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. Il dominio della funzione di transizione è $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Quindi lo stato corrente, il prossimo simbolo di input letto, e il simbolo sulla cima della pila determinano la mossa seguente di un automa a pila. L'uno o l'altro simbolo può essere ε, il che determina che la macchina si muova senza leggere un simbolo di input o senza leggere un simbolo dalla pila.

Per quanto riguarda il codominio della funzione di transizione, dobbiamo considerare cosa può fare l'automa quando è in una specifica situazione. Esso può trovarsi in un nuovo stato ed eventualmente scrivere un simbolo sulla cima della pila. La funzione δ può indicare quest'azione restituendo un elemento di Q insieme a un elemento di Γ_ε, cioè un elemento di $Q \times \Gamma_\varepsilon$. Poiché permettiamo il non determinismo in questo modello, una situazione può avere diverse mosse successive lecite. La funzione di transizione ingloba il non determinismo nel modo usuale, restituendo un insieme di elementi di $Q \times \Gamma_\varepsilon$, cioè un elemento di $\mathcal{P}(Q \times \Gamma_\varepsilon)$. Mettendo tutto questo insieme, la nostra funzione di transizione δ assume la forma $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.

DEFINIZIONE **2.13**

Un ***automa a pila*** è una sestupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, dove Q, Σ, Γ ed F sono tutti insiemi finiti, e

1. Q è l'insieme degli stati,
2. Σ è l'alfabeto dell'input,
3. Γ è l'alfabeto della pila,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ è la funzione di transizione,
5. $q_0 \in Q$ è lo stato iniziale, ed
6. $F \subseteq Q$ è l'insieme degli stati accettanti.

Un automa a pila $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computa come segue. Accetta un input w se w può essere scritto come $w = w_1 w_2 \cdots w_m$, dove ciascun $w_i \in \Sigma_\varepsilon$ ed esistono sequenze di stati $r_0, r_1, \ldots, r_m \in Q$ e di stringhe $s_0, s_1, \ldots, s_m \in \Gamma^*$ che soddisfano le tre condizioni seguenti. Le stringhe s_i rappresentano la sequenza del contenuto della pila che M ha su un ramo accettante della computazione.

1. $r_0 = q_0$ and $s_0 = \varepsilon$. Questa condizione significa che M inizia correttamente, nello stato iniziale e con una pila vuota.
2. Per $i = 0, \ldots, m-1$, abbiamo $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, dove $s_i = at$ ed $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$. Questa condizione afferma che M si muove correttamente in base allo stato, al simbolo di pila, e al prossimo simbolo in input.
3. $r_m \in F$. Questa condizione afferma che alla fine dell'input M si trova in uno stato accettante.

Esempi di automi a pila

ESEMPIO **2.14**

Quello che segue è la descrizione formale del PDA (pagina 116) che riconosce il linguaggio $\{0^n 1^n \mid n \geq 0\}$. Sia M_1 la sestupla $(Q, \Sigma, \Gamma, \delta, q_1, F)$, dove

$Q = \{q_1, q_2, q_3, q_4\}$,

$\Sigma = \{0, 1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$ e

δ è data dalla seguente tabella, nella quale le voci vuote indicano $\emptyset$.

Input:	0			1			ε		
Pila:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$			$\{(q_4, \varepsilon)\}$		
q_4									

Possiamo anche usare un diagramma di stato per descrivere un PDA, come nelle Figure 2.15, 2.17 e 2.19. Tali diagrammi sono simili ai diagrammi di stato usati per descrivere gli automi finiti, modificati per mostrare come il PDA usa la sua pila quando passa da uno stato a un altro. Scriviamo "$a,b \to c$" per indicare che, quando la macchina sta leggendo una a dall'input, essa può sostituire il simbolo b sulla sommità della pila con una c. Ognuno dei simboli a, b e c può essere ε. Se a è ε, la macchina può effettuare questa transizione senza leggere alcun simbolo dall'input. Se b è ε, la macchina può realizzare questa transizione senza leggere ed eliminare alcun simbolo dalla pila. Se c è ε, la macchina non scrive alcun simbolo sulla pila durante questa transizione.

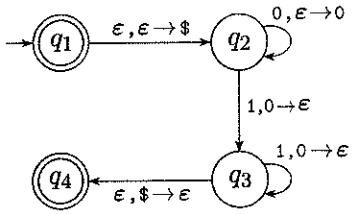


FIGURA 2.15
Diagramma di stato per il PDA M_1 che riconosce $\{0^n1^n | n \geq 0\}$

La definizione di un PDA non contiene alcun meccanismo esplicito per permettere al PDA di controllare se la pila è vuota. Questo PDA è in grado di ottenere lo stesso effetto inserendo inizialmente un simbolo speciale $ nella pila. Allora, nel caso in cui veda il $ di nuovo, sa che la pila è di fatto vuota. In seguito, quando in una descrizione informale di un PDA facciamo riferimento al controllo se la pila è vuota, implementiamo questa stessa procedura.

Analogamente, i PDA non possono controllare esplicitamente se hanno raggiunto la fine della stringa di input . Questo PDA è in grado di ottenere lo stesso risultato poiché lo stato accettante potrebbe essere eventualmente raggiunto solo quando la macchina è alla fine dell'input. Quindi, d'ora in poi, assumiamo che i PDA possono controllare la fine dell'input e sappiamo che possiamo implementarlo nello stesso modo.

ESEMPIO 2.16

Questo esempio illustra un automa a pila che riconosce il linguaggio

$$\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | \, i, j, k \geq 0 \text{ e } i = j \text{ oppure } i = k\}.$$

Informalmente, il PDA per questo linguaggio opera dapprima leggendo e inserendo le a nella pila. Quando le a sono terminate, la macchina le ha tutte nella pila, quindi può abbinarle con le b o con le c. Quest'operazione è un pó complicata poiché la macchina non sa in anticipo se far corrispondere le a con le b o con le c. In questo caso il non determinismo risulta utile.

Usando il suo non determinismo, il PDA può ipotizzare di far corrispondere le a con le b o con le c come mostrato in Figura 2.17. Si pensi alla macchina come se avesse due rami del suo non determinismo, uno per ogni possibile ipotesi. Se il numero di una delle due lettere b o c corrisponde al numero di a), quel ramo accetta e l'intera macchina accetta. Il Problema 2.27 chiede di mostrare che il non determinismo è necessario per riconoscere questo linguaggio con un PDA.

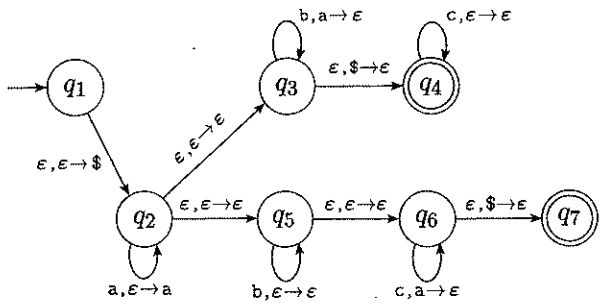


FIGURA 2.17
Diagramma di stato per il PDA M_2 che riconosce $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | \, i, j, k \geq 0 \text{ e } i = j \text{ oppure } i = k\}$

ESEMPIO 2.18

In questo esempio diamo un PDA M_3 che riconosce il linguaggio $\{ww^{\mathcal{R}} | \, w \in \{0,1\}^*\}$. Ricordiamo che $w^{\mathcal{R}}$ significa w scritta al contrario. Nel seguito diamo la descrizione informale e il diagramma di stato del PDA.

Inizia inserendo nella pila i simboli letti. In qualunque momento ipotizza non deterministicamente di aver raggiunto la prima metà della stringa e

quindi cambia azione, eliminando dalla pila un simbolo per ciascun simbolo letto, se essi coincidono. Se essi coincidono sempre e la pila si svuota nello stesso momento in cui l'input è terminato, accetta; altrimenti rifiuta.

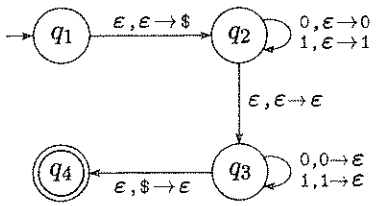


FIGURA 2.19
Diagramma di stato per il PDA M_3 che riconosce $\{ww^R \mid w \in \{0,1\}^*\}$

Il Problema 2.28 mostra che questo linguaggio richiede un PDA non deterministico.

Equivalenza con le grammatiche context-free

In questa sezione mostriamo che grammatiche context-free e automi a pila sono computazionalmente equivalenti. Entrambi sono in grado di descrivere la classe dei linguaggi context-free. Mostreremo come trasformare ogni grammatica context-free in un automa a pila che riconosce lo stesso linguaggio e viceversa. Ricordiamo che abbiamo definito un linguaggio context-free come un linguaggio che può essere descritto con una grammatica context-free. Quindi, il nostro obiettivo è il teorema seguente.

TEOREMA 2.20

Un linguaggio è context free se e solo se esiste un automa a pila che lo riconosce.

Come al solito, per i teoremi "se e solo se", abbiamo due enunciati da provare, nelle due direzioni. In questo teorema, entrambi gli enunciati sono interessanti. Innanzitutto trattiamo la direzione più facile, la diretta.

LEMMA 2.21

Se un linguaggio è context-free, allora esiste un automa a pila che lo riconosce.

IDEA. Sia A un CFL. Dalla definizione sappiamo che esiste una CFG, G, che genera A. Mostriamo come trasformare G in un PDA equivalente, che chiamiamo P.

Il PDA P che ora descriviamo, opererà accettando il suo input w, se G genera tale input, determinando se esiste una derivazione per w. Ricordiamo che una derivazione è semplicemente la sequenza di sostituzioni fatte nel processo di generazione di una stringa mediante una grammatica. Ogni passo di una derivazione produce una ***stringa intermedia*** di variabili e terminali. Noi progettiamo P in modo che possa stabilire se una serie di sostituzioni che usano le regole di G possa condurre dalla variabile iniziale a w.

Una delle difficoltà nell'esaminare se esiste una derivazione per w consiste nel capire quali sostituzioni fare. Il non determinismo di un PDA gli consente di ipotizzare la sequenza di sostituzioni giuste. In ogni passo della derivazione, una delle regole per una particolare variabile è scelta non deterministicamente e usata per sostituire quella variabile.

Il PDA P inizia scrivendo la variabile iniziale sulla sua pila. Esso passa attraverso una serie di stringhe intermedie, facendo una sostituzione dietro l'altra. Infine può giungere a una stringa che contiene solo simboli terminali, il che significa che ha usato la grammatica per derivare una stringa. Allora P accetta se questa stringa è identica alla stringa che ha ricevuto in input.

Implementare questa strategia su un PDA richiede un'idea supplementare. Dobbiamo capire come il PDA immagazzina le stringhe intermedie quando passa dall'una all'altra. Usare semplicemente la pila per immagazzinare ciascuna stringa intermedia è allettante. Tuttavia non funziona del tutto poiché il PDA ha bisogno di trovare le variabili nelle stringhe intermedie e fare sostituzioni. Il PDA può accedere solo al simbolo sulla cima della pila e questo potrebbe essere un simbolo terminale e non una variabile. Il modo per aggirare questo problema è mantenere solo *parte* della stringa intermedia sulla pila: i simboli che iniziano con la prima variabile nella stringa intermedia. Tutti i simboli terminali che compaiono prima della prima variabile sono subito abbinati con i simboli nella stringa di input. La figura seguente mostra il PDA P.

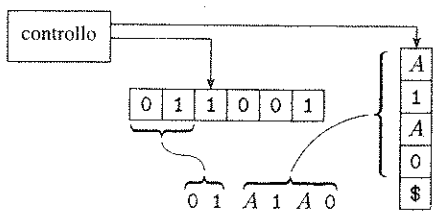


FIGURA 2.22
Come P rappresenta la stringa intermedia $01A1A0$

Quello che segue è una descrizione informale di P.

1. Inserisce il simbolo marcatore \$ e la variabile iniziale sulla pila.
2. Ripete i seguenti passi.
   a. Se sulla cima della pila c'è il simbolo di una variabile A, sceglie non deterministicamente una delle regole per A e sostituisce A con la stringa sul lato destro della regola.
   b. Se sulla cima della pila c'è un simbolo terminale a, legge il simbolo seguente dall'input e lo confronta con a. Se essi sono uguali, ripete. Se essi non sono uguali, rifiuta su questo ramo del non determinismo.
   c. Se sulla cima della pila c'è il simbolo \$, entra nello stato accettante. In questo modo accetta l'input se esso è stato completamente letto.

DIMOSTRAZIONE Ora diamo i dettagli formali della costruzione dell'automa a pila $P = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, F)$. Per rendere più chiara la costruzione, usiamo una notazione abbreviata per la funzione di transizione. Questa notazione fornisce un modo per scrivere un'intera stringa sulla pila in un passo della macchina. Possiamo simulare quest'azione introducendo stati aggiuntivi per scrivere la stringa un simbolo alla volta, come realizzato nella seguente costruzione formale.

Siano q ed r stati del PDA e siano a in Σ_ε ed s in Γ_ε. Supponiamo di volere che il PDA vada da q a r quando legge a ed elimina s. Inoltre, vogliamo che esso inserisca l'intera stringa $u = u_1 \cdots u_l$ sulla pila simultaneamente. Possiamo eseguire questa azione introducendo nuovi stati $q_1, \ldots, q_{l-1}$ e definendo la funzione di transizione come segue:

$$\begin{aligned}
&\delta(q, a, s) \text{ contiene } (q_1, u_l),\\
&\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{l-1})\},\\
&\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{l-2})\},\\
&\qquad\vdots\\
&\delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}.
\end{aligned}$$

Useremo la notazione $(r, u) \in \delta(q, a, s)$ per denotare che quando q è lo stato in cui si trova l'automa, a è il prossimo simbolo di input ed s è il simbolo sulla cima della pila, il PDA può leggere a ed eliminare s, poi inserire la stringa u nella pila e passare nello stato r. La figura seguente ne mostra la realizzazione.

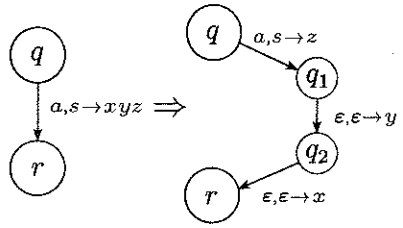


FIGURA 2.23
Implementazione dell'abbreviazione $(r, xyz) \in \delta(q, a, s)$

Gli stati di P sono $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, dove E è l'insieme degli stati necessari per realizzare l'abbreviazione appena descritta. Lo stato iniziale è q_{start}. L'unico stato accettante è q_{accept}.

La funzione di transizione è definita come segue. Cominciamo inizializzando la pila inserendo i simboli \$ and S, realizzando così il passo 1 nella descrizione informale: $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$. Poi aggiungiamo le transizioni per il ciclo principale del passo 2.

In primo luogo, trattiamo il caso (a) in cui la cima della pila contiene una variabile. Poniamo $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) | \text{ dove } A \to w \text{ è una regola in } R\}$.

In secondo luogo, trattiamo il caso (b) in cui la cima della pila contiene un terminale. Poniamo $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

Infine, trattiamo il caso (c) in cui il marcatore scelto per indicare la pila vuota \$ è sulla cima della pila. Poniamo $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

Il diagramma di stato è mostrato nella Figura 2.24.

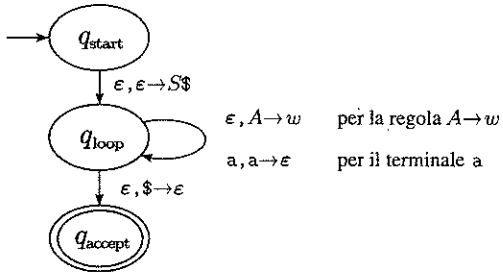


FIGURA 2.24
Diagramma di stato di P

Questo completa la prova del Lemma 2.21.

ESEMPIO 2.25

Usiamo la procedura sviluppata nel Lemma 2.21 per costruire un PDA P_1 dalla seguente CFG G.

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

La funzione di transizione è mostrata nel diagramma seguente.

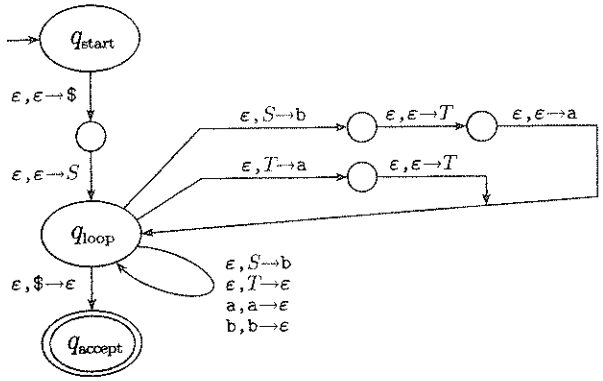


FIGURA 2.26
Diagramma di stato di P_1

Ora dimostriamo la direzione inversa del Teorema 2.20. Per la direzione diretta, abbiamo dato una procedura per trasformare una CFG in un PDA. L'idea principale era progettare l'automa che simula la grammatica. Ora vogliamo dare una procedura per andare nella direzione opposta: trasformare un PDA in una CFG. Progettiamo la grammatica per simulare l'automa. Questo compito è impegnativo perché "programmare" un automa è più facile che "programmare" una grammatica.

LEMMA 2.27

Se un linguaggio è riconosciuto da un automa a pila, allora esso è context-free.

IDEA. Abbiamo un PDA P e vogliamo costruire una CFG G che genera tutte le stringhe che P accetta. In altre parole, G dovrebbe generare una stringa se quella stringa fa andare il PDA dal suo stato iniziale a uno stato accettante.

Per raggiungere questo risultato, progettiamo una grammatica che fa un pò in più. Per ciascuna coppia di stati p e q in P, la grammatica avrà una

variabile A_{pq}. Questa variabile genera tutte le stringhe che possano portare P da p con pila vuota a q con pila vuota. Si osservi che tali stringhe possono anche condurre P da p a q, indipendentemente dal contenuto della pila in p, lasciando la pila in q nella stessa condizione in cui era in p.

Innanzitutto, semplifichiamo il nostro compito modificando leggermente P per munirlo delle seguenti tre caratteristiche.

1. Ha un unico stato accettante, q_{accept}.
2. Svuota la sua pila prima di accettare.
3. Ciascuna transizione inserisce un simbolo sulla pila (effettua un *push*) o ne elimina uno dalla pila (effettua un *pop*), ma non fa entrambe le azioni contemporaneamente.

Dare a P le caratteristiche 1 e 2 è facile. Per munirlo della caratteristica 3, sostituiamo ciascuna transizione che contemporaneamente elimina ed inserisce simboli con una sequenza di due transizioni che attraversa un nuovo stato, e sostituiamo ogni transizione che non elimina né inserisce simboli con una sequenza di due transizioni che inserisce e poi elimina un simbolo arbitrario della pila.

Per progettare G in modo che A_{pq} generi tutte le stringhe che portano P da p da q, iniziando e terminando con una pila vuota, dobbiamo capire come P agisce su queste stringhe. Per ognuna di tali stringhe x, la prima mossa di P su x deve essere un push, poiché ogni mossa è un push o un pop e P non può eliminare da una pila vuota. Analogamente, l'ultima mossa su x deve essere un pop perché alla fine la pila deve essere vuota.

Durante la computazione di P su x si presentano due eventualità. O il simbolo eliminato alla fine è il simbolo che era stato inserito all'inizio, oppure no. Nel primo caso, la pila potrebbe essere vuota solo all'inizio e alla fine della computazione di P su x. Altrimenti, il simbolo inizialmente inserito deve essere stato eliminato in qualche punto prima della fine di x e quindi la pila si svuota in questo punto. Simuliamo la prima possibilità con la regola $A_{pq} \to aA_{rs}b$, dove a è l'input letto nella prima mossa, b è l'input letto nell'ultima mossa, r è lo stato che segue p ed s è lo stato che precede q. Simuliamo la seconda possibilità con la regola $A_{pq} \to A_{pr}A_{rq}$, dove r è lo stato in cui la pila diventa vuota.

DIMOSTRAZIONE. Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ e costruiamo G. L'insieme delle variabili di G è $\{A_{pq} \mid p, q \in Q\}$. La variabile iniziale è $A_{q_0, q_{\text{accept}}}$. Ora descriviamo le regole di G nei seguenti tre punti.

1. Per ogni $p, q, r, s \in Q$, $u \in \Gamma$ e $a, b \in \Sigma_\varepsilon$, se $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε), poni la regola $A_{pq} \to aA_{rs}b$ in G.
2. Per ogni $p, q, r \in Q$, poni la regola $A_{pq} \to A_{pr}A_{rq}$ in G.
3. Infine, per ogni $p \in Q$, poni la regola $A_{pp} \to \varepsilon$ in G.

Si può intuire questa costruzione dalle figure seguenti.

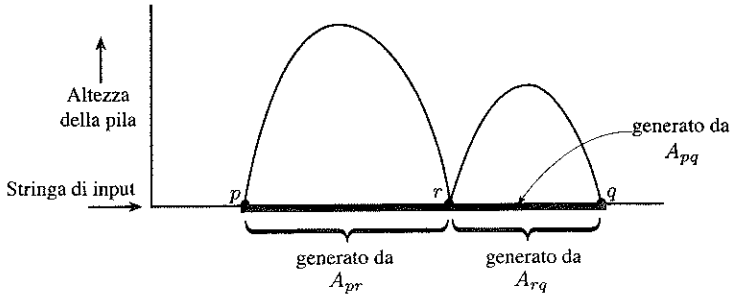


FIGURA 2.28
La computazione del PDA corrispondente alla regola $A_{pq} \to A_{pr}A_{rq}$

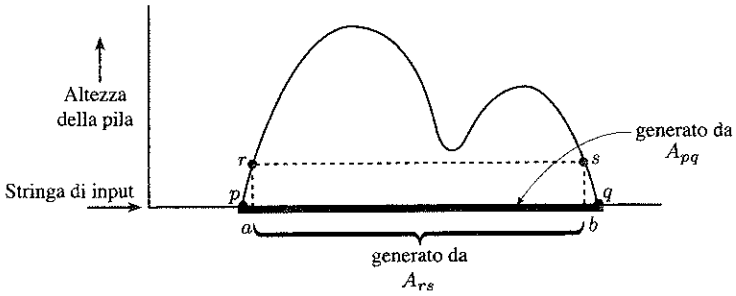


FIGURA 2.29
La computazione del PDA corrispondente alla regola $A_{pq} \to aA_{rs}b$

Ora proviamo che questa costruzione funziona dimostrando che A_{pq} genera x se e solo se x porta P da p con la pila vuota a q con la pila vuota. Consideriamo ogni direzione del "se e solo se" come un enunciato separato.

FATTO 2.30

Se A_{pq} genera x, allora x può portare P da p con la pila vuota a q con la pila vuota.

Dimostriamo questo enunciato per induzione sul numero dei passi nella derivazione di x da A_{pq}.

Base: La derivazione è in un solo passo.
Una derivazione in un solo passo deve usare una regola il cui lato destro non contenga variabili. Le uniche regole in G con nessuna variabile sul lato destro sono $A_{pp} \to \varepsilon$. Ovviamente, l'input ε porta P da p con la pila vuota a p con la pila vuota quindi la base è dimostrata.

Passo induttivo: Assumiamo che l'enunciato sia vero per le derivazioni di lunghezza al più k, dove $k \geq 1$, e proviamo che esso è vero per le derivazioni di lunghezza $k+1$.
Supponiamo che $A_{pq} \stackrel{*}{\Rightarrow} x$ in $k+1$ passi. Il primo passo in questa derivazione è $A_{pq} \Rightarrow aA_{rs}b$ oppure $A_{pq} \Rightarrow A_{pr}A_{rq}$. Trattiamo questi due casi separatamente.

Nel primo caso, consideriamo la parte y di x che A_{rs} genera, quindi $x = ayb$. Poiché $A_{rs} \stackrel{*}{\Rightarrow} y$ in k passi, l'ipotesi induttiva ci dice che P può andare da r con la pila vuota a s con la pila vuota. Poiché $A_{pq} \to aA_{rs}b$ è una regola di G, $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε), per qualche simbolo di pila u. Quindi, se P inizia in p con la pila vuota, dopo aver letto a può andare nello stato r e inserire u in cima alla pila. Poi, la lettura della stringa y può portarlo in s e lasciare u sulla pila. In seguito dopo aver letto b può andare nello stato q ed eliminare u dalla pila. Pertanto, x può portare P da p con la pila vuota a q con la pila vuota.

Nel secondo caso, consideriamo le parti y e z di x che A_{pr} e A_{rq} rispettivamente generano, quindi $x = yz$. Poiché $A_{pr} \stackrel{*}{\Rightarrow} y$ in al più k passi e $A_{rq} \stackrel{*}{\Rightarrow} z$ in al più k passi, l'ipotesi induttiva ci dice che y può portare P da p a r e z può portare P da r a q, con la pila vuota all'inizio e alla fine della computazione. Quindi x può portare P da p con la pila vuota a q con la pila vuota. Questo completa la prova del passo induttivo.

FATTO 2.31

Se x può portare P da p con la pila vuota a q con la pila vuota, A_{pq} genera x.

Dimostriamo questo enunciato per induzione sul numero dei passi nella computazione di P da p a q con pile vuote sull'input x.

Base: La computazione è in 0 passi.
Se una computazione è in 0 passi, essa inizia e termina nello stesso stato – diciamo, p. Quindi dobbiamo mostrare che $A_{pp} \stackrel{*}{\Rightarrow} x$. P non può leggere alcun carattere in 0 passi, quindi $x = \varepsilon$. Per costruzione, G ha la regola $A_{pp} \to \varepsilon$, perciò la base è dimostrata.

Passo induttivo: Assumiamo l'enunciato vero per le computazioni di lunghezza al più k, dove $k \geq 0$, e dimostriamo che è vero per le computazioni di lunghezza $k+1$.
Supponiamo che P abbia una computazione dove x porta da p a q con pile vuote in $k+1$ passi. O la pila è vuota solo all'inizio e alla fine di questa computazione oppure essa si svuota anche altrove.

Nel primo caso, il simbolo che è stato inserito nella prima mossa deve essere lo stesso simbolo che è stato rimosso nell'ultima mossa. Chiamiamo u questo simbolo. Sia a il simbolo di input letto nella prima mossa, b il simbolo di input letto nell'ultima mossa, r lo stato dopo la prima mossa ed s lo stato prima dell'ultima mossa. Allora $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε) e quindi la regola $A_{pq} \to aA_{rs}b$ è in G.

Sia y la parte di x senza a e b, quindi $x = ayb$. L'input y può portare P da r a s senza toccare il simbolo u che è sulla cima della pila e quindi P può andare da r con una pila vuota a s con una pila vuota sull'input y. Abbiamo eliminato il primo e l'ultimo passo dei $k+1$ passi nella computazione iniziale su x, perciò la computazione su y ha $(k+1) - 2 = k - 1$ passi. Quindi l'ipotesi induttiva ci dice che $A_{rs} \stackrel{*}{\Rightarrow} y$. Allora $A_{pq} \stackrel{*}{\Rightarrow} x$.

Nel secondo caso, sia r uno stato in cui la pila si svuota oltre che all'inizio o alla fine della computazione su x. Allora le parti della computazione da p a r e da r a q contengono al più k passi. Sia y l'input letto nella prima parte e sia z l'input letto nella seconda parte. L'ipotesi induttiva ci dice che $A_{pr} \stackrel{*}{\Rightarrow} y$ e $A_{rq} \stackrel{*}{\Rightarrow} z$. Poiché la regola $A_{pq} \to A_{pr}A_{rq}$ è in G, $A_{pq} \stackrel{*}{\Rightarrow} x$ e la dimostrazione è completa.

Questo completa la prova del Lemma 2.27 e del Teorema 2.20.

Abbiamo appena dimostrato che gli automi a pila riconoscono la classe dei linguaggi context-free. Questa dimostrazione ci consente di stabilire una relazione tra i linguaggi regolari e i linguaggi context-free. Poiché ogni linguaggio regolare è riconosciuto da un automa finito e ogni automa finito è automaticamente un automa a pila che semplicemente ignora la sua pila, ora sappiamo che ogni linguaggio regolare è anche un linguaggio context-free.

COROLLARIO 2.32

Ogni linguaggio regolare è context-free.

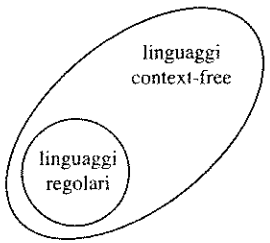


FIGURA 2.33
Relazione tra i linguaggi regolari e i linguaggi context-free

2.3
LINGUAGGI NON CONTEXT-FREE

In questa sezione presentiamo una tecnica per dimostrare che alcuni linguaggi non sono context-free. Ricordiamo che nella Sezione 1.4 introducemmo il pumping lemma per dimostrare che alcuni linguaggi non sono regolari. A questo punto presentiamo un pumping lemma simile per i linguaggi context-free. Esso stabilisce che per ogni linguaggio context-free esiste un particolare valore chiamato la ***lunghezza del pumping*** tale che tutte le stringhe più lunghe nel linguaggio possono essere "iterate." Questa volta il significato di *iterazione* è un pò più complesso. Significa che la stringa può essere divisa in cinque parti in modo tale che la seconda e la quarta parte possono insieme essere replicate un numero qualsiasi di volte ottenendo una stringa che appartiene ancora al linguaggio.

Il pumping lemma per i linguaggi context-free

TEOREMA 2.34

Il pumping lemma per i linguaggi context-free Se A è un linguaggio context-free, allora esiste un numero p (la lunghezza del pumping) tale che, se s è una qualsiasi stringa in A di lunghezza almeno p, allora s può essere divisa in cinque parti $s = uvxyz$ che soddisfano le condizioni

1. per ogni $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$ e
3. $|vxy| \leq p$.

Quando s è divisa in $uvxyz$, la condizione 2 afferma che v oppure y non è la stringa vuota. Altrimenti il teorema sarebbe banalmente vero. La condizione 3 afferma che le parti v, x e y insieme hanno lunghezza al più p. Questa condizione tecnica a volte è utile per dimostrare che alcuni linguaggi non sono context-free.

IDEA. Sia A un CFL e sia G una CFG che lo genera. Dobbiamo mostrare che ogni stringa sufficientemente lunga s in A può essere iterata e restare in A. L'idea dietro questa strategia è semplice.

Sia s una stringa molto lunga in A. (Chiariremo in seguito cosa intendiamo con "molto lunga.") Poiché s è in A, essa è derivabile da G e quindi ha un albero sintattico. L'albero sintattico per s deve essere molto alto perché s è molto lunga. Cioè l'albero sintattico deve contenere un cammino lungo dalla variabile alla radice dell'albero a uno dei simboli terminali su una

foglia. Per il principio della piccionaia, qualche simbolo di variabile R si deve ripetere in questo cammino lungo. Come mostra la figura seguente, questa ripetizione ci permette di sostituire il sottoalbero sotto la seconda occorrenza di R con il sottoalbero sotto la prima occorrenza di R e ottenere ancora un albero sintattico consentito. Pertanto, possiamo dividere s in cinque parti $uvxyz$, come indicato nella figura, e possiamo replicare il secondo e quarto pezzo e ottenere una stringa ancora nel linguaggio. In altre parole, uv^ixy^iz è in A per ogni $i \geq 0$.

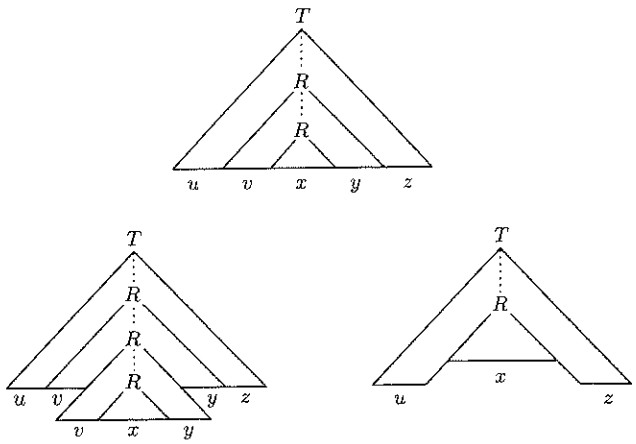


FIGURA 2.35
Operazioni sugli alberi sintattici

Veniamo ora ai dettagli per ottenere tutte e tre le condizioni del pumping lemma. Mostreremo anche come calcolare la lunghezza del pumping p.

DIMOSTRAZIONE Sia G una CFG per il CFL A. Sia b il massimo numero di simboli nel lato destro di una regola (assumiamo che sia almeno 2). Sappiamo che, in ogni albero sintattico costruito usando questa grammatica, un nodo non può avere più di b figli. In altre parole, ci sono al più b foglie in un passo dalla variabile iniziale; ci sono al più b^2 foglie in 2 passi dalla variabile iniziale; e ci sono al più b^h foglie in h passi dalla variabile iniziale. Quindi, se l'altezza dell'albero sintattico è al più h, la lunghezza della stringa generata è al più b^h. Viceversa, se una stringa generata ha lunghezza maggiore o uguale a $b^h + 1$, ciascuno dei suoi alberi sintattici deve avere un'altezza maggiore o uguale a $h + 1$.

Sia $|V|$ il numero delle variabili in G. Poniamo p, la lunghezza del pumping, uguale a $b^{|V|+1}$. Ora se s è una stringa in A e la sua lunghezza è maggiore o uguale a p, il suo albero sintattico deve avere altezza maggiore o uguale a $|V| + 1$, poiché $b^{|V|+1} \geq b^{|V|} + 1$.

Per vedere come iterare una tale stringa s, sia τ uno dei suoi alberi sintattici. Se s ha diversi alberi sintattici, scegliamo un albero sintattico τ che abbia il più piccolo numero di nodi. Sappiamo che τ deve avere altezza maggiore o uguale a $|V| + 1$, quindi il suo cammino più lungo dalla radice a una foglia ha lunghezza almeno $|V| + 1$. Questo cammino ha almeno $|V| + 2$ nodi; uno etichettato da un terminale, gli altri etichettati da variabili. Quindi questo cammino ha almeno $|V| + 1$ variabili. Poiché G ha solo $|V|$ variabili, qualche variabile R è presente più di una volta su questo cammino. Per un utilizzo successivo, scegliamo R in modo che sia una variabile che si ripete tra le $|V| + 1$ variabili più in basso su questo cammino.

Dividiamo s in $uvxyz$ come nella Figura 2.35. Ogni occorrenza di R ha un sottoalbero sotto essa che genera una parte della stringa s. L'occorrenza più in alto di R ha un sottoalbero più grande e genera vxy, mentre l'occorrenza più in basso genera solo x con un sottoalbero più piccolo. Entrambi questi sottoalberi sono generati dalla stessa variabile, quindi possiamo sostituire l'uno con l'altro e ottenere ancora un albero sintattico corretto. Sostituire ripetutamente il più piccolo con il più grande fornisce gli alberi sintattici per le stringhe uv^ixy^iz per ogni $i > 1$. Sostituire il più grande con il più piccolo genera la stringa uxz. Questo dimostra la condizione 1 del lemma. Ora veniamo alle condizioni 2 e 3.

Per ottenere la condizione 2, dobbiamo essere sicuri che v e y non sono entrambe ε. Se lo fossero, l'albero sintattico ottenuto sostituendo il più piccolo sottoalbero al più grande avrebbe meno nodi di τ e genererebbe ancora s. Questo non è possibile perché abbiamo scelto τ in modo che sia un albero sintattico per s con il più piccolo numero di nodi. Questa è la ragione per aver selezionato così τ.

Per ottenere la condizione 3, dobbiamo essere sicuri che vxy ha lunghezza al più p. Nell'albero sintattico per s l'occorrenza più in alto di R genera vxy. Abbiamo scelto R in modo che entrambe le occorrenze di essa cadano nelle $|V|+1$ variabili più in basso del cammino e abbiamo scelto il più lungo cammino nell'albero sintattico, in modo che il sottoalbero in cui R genera vxy sia alto al più $|V| + 1$. Un albero con questa altezza può generare una stringa di lunghezza al più $b^{|V|+1} = p$.

Per alcuni suggerimenti sull'uso del pumping lemma per dimostrare che alcuni linguaggi non sono context-free, rivedi il testo che precede l'Esempio 1.73 (pagina 83) dove trattiamo il problema correlato di provare la non regolarità con il pumping lemma per i linguaggi regolari.

ESEMPIO 2.36

Usiamo il pumping lemma per mostrare che il linguaggio $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \geq 0\}$ non è context-free.

Assumiamo che B sia un CFL e giungiamo a una contraddizione. Sia p la lunghezza del pumping per B la cui esistenza è garantita dal pumping lemma. Scegliamo la stringa $s = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. Ovviamente s è un elemento di B e di lunghezza almeno p. Il pumping lemma afferma che s può essere iterata, ma noi mostriamo che non può esserlo. In altre parole, mostriamo che, non importa come dividiamo s in $uvxyz$, una delle tre condizioni del lemma è violata.

In primo luogo, la condizione 2 stabilisce che v o y non è vuota. Allora consideriamo due casi, a seconda che le sottostringhe v e y contengano più di un tipo di simbolo dell'alfabeto o no.

1. Quando entrambe v e y contengono solo un tipo di simbolo dell'alfabeto, v non contiene entrambi i simboli a e b o entrambi i simboli b e c e lo stesso vale per y. In questo caso, la stringa uv^2xy^2z non può contenere lo stesso numero di a, b e c. Quindi, essa non può essere un elemento di B. Questo viola la condizione 1 del lemma e allora abbiamo una contraddizione.
2. Quando v o y contengono più di un tipo di simbolo, uv^2xy^2z può contenere un ugual numero dei tre simboli dell'alfabeto ma non nell'ordine corretto. Perciò essa non può essere un elemento di B e si verifica un assurdo.

Uno di questi casi deve verificarsi. Poiché entrambi i casi conducono a un assurdo, la contraddizione è inevitabile. Quindi l'assunzione che B sia un CFL deve essere falsa. Pertanto abbiamo provato che B non è un CFL.

ESEMPIO 2.37

Sia $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | 0 \leq i \leq j \leq k\}$. Usiamo il pumping lemma per mostrare che C non è un CFL. Questo linguaggio è simile al linguaggio B nell'Esempio 2.36, ma provare che esso non è context-free è un pò più complicato.

Assumiamo che C sia un CFL e otteniamo una contraddizione. Sia p la lunghezza del pumping fornita dal pumping lemma. Usiamo la stringa $s = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$ che abbiamo usato prima, ma questa volta dobbiamo eliminare ("pump down") oltre a iterare ("pump up"). Sia $s = uvxyz$ e consideriamo nuovamente i due casi presenti nell'Esempio 2.36.

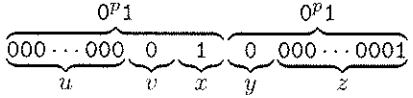
1. Quando v e y contengono solo un tipo di simbolo dell'alfabeto, v non contiene entrambi i simboli a e b o entrambi i simboli b e c e lo stesso vale per y. Nota che il ragionamento usato in precedenza nel caso 1 non è più utilizzabile. Il motivo è che C contiene stringhe con numeri diversi di a, b, e c se la sequenza di tali numeri è non decrescente. Dobbiamo analizzare la situazione con più attenzione per mostrare che s non può essere iterata. Osserviamo che, poiché v e y contengono solo un tipo di simbolo dell'alfabeto, uno dei simboli a, b o c non è presente in v o y. Suddividiamo ulteriormente questo caso in tre sottocasi, a seconda di quale simbolo non sia presente.
   a. *Il simbolo a non è presente.* Allora proviamo a eliminare ("pumping down") ottenendo la stringa $uv^0xy^0z = uxz$. Questa stringa contiene lo stesso numero di a che ha s, ma contiene meno b o meno c. Perciò essa non è un elemento di C e abbiamo una contraddizione.
   b. *Il simbolo b non è presente.* Allora dei simboli uguali ad a o a c devono apparire in v o y poiché esse non possono essere entrambe la stringa vuota. Se sono presenti delle a, la stringa uv^2xy^2z contiene più a che b, quindi essa non è in C. Se sono presenti delle c, la stringa uv^0xy^0z contiene più b che c, quindi essa non è in C. In ogni caso abbiamo una contraddizione.
   c. *Il simbolo c non è presente.* Allora la stringa uv^2xy^2z contiene più a o più b che c, quindi essa non è in C, e abbiamo un assurdo.
2. Quando v o y contengono più di un tipo di simbolo, uv^2xy^2z non conterrà i simboli nell'ordine corretto. Perciò essa non può essere un elemento di C e giungiamo a una contraddizione.

Pertanto abbiamo mostrato che s non può essere iterata in contrasto con il pumping lemma e che C non è context-free.

ESEMPIO 2.38

Sia $D = \{ww | w \in \{0,1\}^*\}$. Usiamo il pumping lemma per mostrare che D non è un CFL. Assumiamo che D sia un CFL e otteniamo una contraddizione. Sia p la lunghezza del pumping fornita dal pumping lemma.

Questa volta scegliere la stringa s è meno ovvio. Una possibilità è la stringa 0^p10^p1. È un elemento di D e ha lunghezza maggiore di p, quindi sembra essere un buon candidato. Ma questa stringa si *può* iterare, dividendola come segue, perciò non è adeguata per il nostro scopo.



Proviamo con un altro candidato per s. Intuitivamente, la stringa $0^p1^p0^p1^p$ sembra catturare maggiormente l'"essenza" del linguaggio D di quanto facesse il precedente candidato. In effetti, possiamo mostrare che questa stringa funziona nel modo seguente.

Mostriamo che la stringa $s = 0^p1^p0^p1^p$ non può essere iterata. Questa volta usiamo la condizione 3 del pumping lemma per limitare il modo in

cui s può essere divisa. Essa afferma che noi possiamo iterare s dividendo $s = uvxyz$, dove $|vxy| \leq p$.

Innanzitutto, mostriamo che la sottostringa vxy deve stare a cavallo del punto centrale di s. Altrimenti, se la sottostringa è presente solo nella prima metà di s, la stringa uv^2xy^2z sposta un 1 nella prima posizione della seconda metà e quindi essa non può essere della forma ww. Analogamente, se vxy è presente nella seconda metà di s, la stringa uv^2xy^2z sposta uno 0 nell'ultima posizione della prima metà e quindi essa non può essere della forma ww.

Ma se la sottostringa vxy è a cavallo del punto centrale di s, la stringa uxz ha la forma $0^p1^i0^j1^p$, dove i e j non possono essere entrambi p. Questa stringa non è della forma ww. Quindi s non può essere iterata e D non è un CFL.

2.4
LINGUAGGI CONTEXT-FREE DETERMINISTICI

Come ricorderai, gli automi finiti deterministici e gli automi finiti non deterministici sono equivalenti dal punto di vista del potere riconoscitivo di linguaggi. Invece gli automi a pila non deterministici sono più potenti della loro controparte deterministica. Mostreremo che alcuni linguaggi context-free non possono essere riconosciuti da PDA deterministici – questi linguaggi richiedono PDA non deterministici. I linguaggi che sono riconosciuti da automi a pila deterministici (DPDA) sono chiamati linguaggi context-free deterministici (DCFL). Questa sottoclasse dei linguaggi context-free è importante per le applicazioni, come il progetto di parser nei compilatori per i linguaggi di programmazione, perché il problema del parsing è generalmente più facile per i DCFL che per i CFL. In questa sezione diamo una breve panoramica di tale importante e bella tematica.

Nel definire un DPDA, ci atteniamo al principio di base del determinismo: in ogni passo della sua computazione, il DPDA ha al più un modo di procedere in base alla sua funzione di transizione. Definire i DPDA è più complicato che definire i DFA perché i DPDA possono leggere un simbolo di input senza eliminare un simbolo dalla pila e viceversa. Di conseguenza, permettiamo ε-mosse nella funzione di transizione di un DPDA sebbene le ε-mosse siano proibite nei DFA. Queste ε-mosse assumono due forme: ***mosse ε-input*** che corrispondono a $\delta(q, \varepsilon, x)$ e ***mosse ε-pila*** che corrispondono a $\delta(q, a, \varepsilon)$. Una mossa può assumere entrambe le forme, in corrispondenza con $\delta(q, \varepsilon, \varepsilon)$. Se un DPDA può fare una ε-mossa in una certa situazione, gli è vietato fare una mossa in quella stessa situazione che comporta elaborare un simbolo invece di ε. Altrimenti potrebbero verificarsi rami di computazioni lecite parallele, conducendo a un comportamento non deterministico.

La definizione formale segue.

DEFINIZIONE **2.39**

Un ***automa a pila deterministico*** è una sestupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, dove Q, Σ, Γ ed F sono tutti insiemi finiti e

1. Q è l'insieme degli stati,
2. Σ è l'alfabeto dell'input,
3. Γ è l'alfabeto della pila,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ è la funzione di transizione,
5. $q_0 \in Q$ è lo stato iniziale ed
6. $F \subseteq Q$ è l'insieme degli stati accettanti.

La funzione di transizione δ deve soddisfare la seguente condizione. Per ogni $q \in Q$, $a \in \Sigma$ e $x \in \Gamma$, esattamente uno dei valori

$$\delta(q, a, x), \delta(q, a, \varepsilon), \delta(q, \varepsilon, x) \text{ e } \delta(q, \varepsilon, \varepsilon)$$

è diverso dal $\emptyset$.

La funzione di transizione può dare in uscita una singola mossa della forma (r, y) o può indicare nessuna azione dando in uscita $\emptyset$. Per illustrare queste possibilità, consideriamo un esempio. Supponiamo che un DPDA M con funzione di transizione δ sia nello stato q, abbia a come suo successivo simbolo di input e abbia il simbolo x sulla cima della sua pila. Se $\delta(q, a, x) = (r, y)$ allora M legge a, elimina x dalla pila, passa nello stato r ed inserisce y sulla pila. Invece, se $\delta(q, a, x) = \emptyset$ allora quando M è nello stato q, non ha nessuna mossa per leggere a ed eliminare x. In questo caso, la condizione su δ richiede che uno tra $\delta(q, \varepsilon, x)$, $\delta(q, a, \varepsilon)$, o $\delta(q, \varepsilon, \varepsilon)$ sia diverso dal vuoto, e allora M si muove di conseguenza. La condizione obbliga a un comportamento deterministico impedendo al DPDA di fare due azioni diverse nella stessa situazione, come sarebbe il caso se $\delta(q, a, x) \neq \emptyset$ e $\delta(q, a, \varepsilon) \neq \emptyset$. Un DPDA ha esattamente una mossa lecita in ogni situazione in cui la sua pila non è vuota. Se la pila è vuota, un DPDA può fare mosse solo se la funzione di transizione specifica una mossa che elimina ε. Altrimenti il DPDA non ha mosse lecite ed esso rifiuta senza leggere il resto dell'input.

L'accettazione per i DPDA funziona nello stesso modo che per i PDA. Se un DPDA entra in uno stato accettante dopo aver letto l'ultimo simbolo di una stringa di input, esso accetta quella stringa. In tutti gli altri casi, rifiuta quella stringa. Si verifica un rifiuto se il DPDA legge tutto l'input ma non entra in uno stato accettante quando è alla fine, oppure se il DPDA non

riesce a leggere l'intera stringa di input. Quest'ultimo caso può presentarsi se il DPDA prova a eliminare simboli da una pila vuota o se il DPDA fa una sequenza infinita di mosse ε-input senza leggere l'input oltre un certo punto.

Il linguaggio di un DPDA è chiamato un ***linguaggio context-free deterministico***.

ESEMPIO 2.40

Il linguaggio $\{0^n1^n | \ n \geq 0\}$ nell'Esempio 2.14 è un DCFL. Possiamo facilmente trasformare il suo PDA M_1 in un DPDA aggiungendo, per ogni combinazione di stato, simbolo di input e simbolo di pila mancante, transizioni a uno stato "trappola" da cui l'accettazione non è possibile.

Gli Esempi 2.16 e 2.18 forniscono i CFL $\{a^ib^jc^k | i, j, k \geq 0 \text{ e } i = j \text{ o } i = k\}$ e $\{ww^{\mathcal{R}} | \ w \in \{0,1\}^*\}$, che non sono DCFL. I Problemi 2.27 e 2.28 mostrano che il non determinismo è necessario per riconoscere questi linguaggi.

I ragionamenti che coinvolgono i DPDA tendono a essere piuttosto tecnici per natura, e sebbene ci sforzeremo di enfatizzare le idee fondamentali che sono dietro le costruzioni, i lettori possono trovare questa sezione più onerosa delle altre sezioni nei primi capitoli. Il materiale nel seguito del libro non dipende da questa sezione, quindi essa può essere saltata se lo si desidera.

Inizieremo con un lemma tecnico che nel seguito semplificherà la discussione. Come osservato, i DPDA possono rifiutare un input poiché non riescono a leggere l'intera stringa di input, ma questi DPDA introducono casi complessi. Fortunatamente, il lemma seguente mostra che possiamo trasformare un DPDA in uno che evita questo comportamento non conveniente.

LEMMA 2.41

Ogni DPDA ha un DPDA equivalente che legge sempre l'intera stringa di input.

IDEA. Un DPDA può non riuscire a leggere tutta la stringa di input se prova a eliminare simboli da una pila vuota o se effettua una sequenza infinita di mosse ε-input. Chiamiamo la prima situazione *hanging* e la seconda situazione *looping*. Risolviamo il problema dell'hanging inizializzando la pila con un simbolo speciale. Se questo simbolo è in seguito eliminato dalla pila prima della fine dell'input, il DPDA legge fino alla fine dell'input e rifiuta. Risolviamo il problema del looping identificando le situazioni di looping, cioè quelle da cui nessun ulteriore simbolo è più letto, e riprogrammando il DPDA in modo che esso legga e rifiuti l'input invece di essere in looping. Dobbiamo sistemare queste modifiche per risolvere il caso in cui l'hanging o il looping si presenta sull'ultimo simbolo dell'input. Se il DPDA entra in uno stato accettante in un qualsiasi momento dopo che ha letto l'ultimo simbolo, il DPDA modificato accetta invece di rifiutare.

DIMOSTRAZIONE. Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ un DPDA. Innanzitutto aggiungiamo un nuovo stato iniziale q_{start}, un ulteriore stato accettante q_{accept}, un nuovo stato q_{reject}, e anche altri nuovi stati come descritto nel seguito. Eseguiamo le seguenti variazioni per ogni $r \in Q$, $a \in \Sigma_\varepsilon$, e $x, y \in \Gamma_\varepsilon$.

In primo luogo modifichiamo P in modo che, non appena entra in uno stato accettante, resta in stati accettanti fino a quando non legge il simbolo di input successivo. Aggiungiamo un nuovo stato accettante q_a per ogni $q \in Q$. Per ogni $q \in Q$, se $\delta(q, \varepsilon, x) = (r, y)$, poniamo $\delta(q_a, \varepsilon, x) = (r_a, y)$ e poi se $q \in F$, modifichiamo anche δ ponendo $\delta(q, \varepsilon, x) = (r_a, y)$. Per ogni $q \in Q$ e $a \in \Gamma$, se $\delta(q, a, x) = (r, y)$ poniamo $\delta(q_a, a, x) = (r, y)$. Sia F' l'insieme dei nuovi e vecchi stati accettanti.

Poi, modifichiamo P in modo che rifiuti quando prova a eliminare simboli da una pila vuota, inizializzando la pila con un nuovo simbolo di pila speciale \$. Se successivamente P rileva \$ in uno stato non accettante, entra in q_{reject} e scandisce l'input fino alla fine. Se P rileva \$ in uno stato accettante, entra in q_{accept}. In seguito, se resta input non letto, entra in q_{reject} e scandisce l'input fino alla fine. Formalmente, poniamo $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = (q_0, \$)$. Per $x \in \Gamma$ e $\delta(q, a, x) \neq \emptyset$, se $q \notin F'$ allora poniamo $\delta(q, a, \$) = (q_{\text{reject}}, \varepsilon)$, e se $q \in F'$ allora poniamo $\delta(q, a, \$) = (q_{\text{accept}}, \varepsilon)$. Per $a \in \Sigma$, poniamo $\delta(q_{\text{reject}}, a, \varepsilon) = (q_{\text{reject}}, \varepsilon)$ e $\delta(q_{\text{accept}}, a, \varepsilon) = (q_{\text{reject}}, \varepsilon)$.

Infine, modifichiamo P in modo che rifiuti invece di fare una sequenza infinita di mosse ε-input prima della fine dell'input. Per ogni $q \in Q$ e $x \in \Gamma$, chiamiamo (q, x) una *situazione di looping* se, quando P è nello stato q con $x \in \Gamma$ sulla cima della pila, non elimina mai nulla sotto x e non legge mai un simbolo di input. Diciamo che la situazione di looping è *accettante* se P entra in uno stato accettante durante le sue mosse successive, altrimenti essa è *di rifiuto*. Se (q, x) è una situazione di looping accettante, poniamo $\delta(q, \varepsilon, x) = (q_{\text{accept}}, \varepsilon)$, mentre se (q, x) è una situazione di looping di rifiuto, poniamo $\delta(q, \varepsilon, x) = (q_{\text{reject}}, \varepsilon)$.

Per semplicità, assumeremo d'ora in avanti che i DPDA leggano il loro input fino alla fine.

Proprietà dei DCFL

Esamineremo proprietà di chiusura e di non chiusura della classe dei DCFL e le useremo per esibire un CFL che non è un DCFL.

TEOREMA 2.42

La classe dei DCFL è chiusa rispetto al complemento.

IDEA. Scambiando gli stati accettanti e non accettanti di un DFA si ottiene un nuovo DFA che riconosce il complemento del linguaggio, mostrando in tal modo che la classe dei linguaggi regolari è chiusa rispetto al complemento. La stessa strategia funziona per i DPDA tranne che per un problema. Il DPDA può accettare il suo input entrando sia in stati accettanti che in stati non accettanti in una sequenza di mosse alla fine della stringa di input. Scambiando stati accettanti e non accettanti l'automa modificato accetterebbe ancora in questo caso.

Risolviamo questo problema modificando il DPDA in modo da limitare quando si può presentare l'accettazione. Per ogni simbolo dell'input, il DPDA modificato può entrare in uno stato accettante solo quando è sul punto di leggere il simbolo seguente. In altre parole, solo gli stati "di lettura" – stati in cui viene sempre letto un simbolo di input – possono essere stati accettanti. Allora, scambiando accettazione e non accettazione solo tra questi stati di lettura, invertiamo l'output del DPDA.

DIMOSTRAZIONE. In primo luogo modifichiamo P come descritto nella prova del Lemma 2.41 e sia $(Q, \Sigma, \Gamma, \delta, q_0, F)$ la macchina risultante. Questa macchina legge sempre l'intera stringa di input. Inoltre, una volta entrata in uno stato accettante, resta in stati accettanti fino a quando legge il successivo simbolo di input.

Per effettuare l'idea della prova, abbiamo bisogno di identificare gli stati di lettura. Se il DPDA nello stato q legge un simbolo di input $a \in \Sigma$ senza eliminare simboli dalla pila, cioè $\delta(q, a, \varepsilon) \neq \emptyset$, allora q sarà uno stato di lettura. Invece, se esso legge e rimuove anche un simbolo dalla pila, la decisione nella lettura può dipendere dal simbolo eliminato, quindi dividiamo questo passo in due: una rimozione e poi una lettura. Quindi se $\delta(q, a, x) = (r, y)$ con $a \in \Sigma$ e $x \in \Gamma$, aggiungiamo un nuovo stato q_x e modifichiamo δ ponendo $\delta(q, \varepsilon, x) = (q_x, \varepsilon)$ e $\delta(q_x, a, \varepsilon) = (r, y)$. Definiamo q_x come uno stato di lettura. Gli stati q_x non eliminano mai simboli dalla pila, quindi la loro azione è indipendente dal contenuto della pila. Definiamo q_x come uno stato accettante se $q \in F$. Infine, togliamo il ruolo di stato accettante a ogni stato che non è uno stato di lettura. Il DPDA modificato è equivalente a P, ma esso entra al più una volta in uno stato accettante per ogni simbolo di input, quando è in procinto di leggere il simbolo seguente.

Ora, scambiamo accettazione e non accettazione tra questi stati di lettura. Il DPDA risultante riconosce il linguaggio complemento.

Questo teorema implica che alcuni CFL non sono DCFL. Ogni CFL il cui complemento non è un CFL non è un DCFL. Quindi $A = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ o } j \neq k \text{ con } i, j, k \geq 0\}$ è un CFL ma non un DCFL. Altrimenti $\overline{A}$ sarebbe un CFL, quindi il risultato del Problema 2.30 implicherebbe in maniera errata che $\overline{A} \cap \mathtt{a}^*\mathtt{b}^*\mathtt{c}^* = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ è context-free.

Il Problema 2.23 chiede di mostrare che la classe dei DCFL non è chiusa rispetto ad altre comuni operazioni come l'unione, l'intersezione, lo star e l'inversione.

Per semplificare i ragionamenti, considereremo occasionalmente ***input con simbolo di fine stringa*** dove il simbolo speciale di fine stringa $\dashv$ è aggiunto alla stringa di input. A questo punto aggiungiamo $\dashv$ all'alfabeto dell'input del DPDA. Come mostriamo nel teorema seguente, aggiungere simboli di fine stringa non cambia la potenza dei DPDA. Tuttavia, progettare DPDA su input con simbolo di fine stringa è spesso più facile perché possiamo sfruttare la circostanza di sapere quando la stringa di input termina. Per ogni linguaggio A, definiamo il ***linguaggio con simbolo di fine stringa*** (o ***linguaggio marcato***) $A\dashv$ come la collezione delle stringhe $w\dashv$ dove $w \in A$.

TEOREMA 2.43

A è un DCFL se e solo se $A\dashv$ è un DCFL.

IDEA. Provare la direzione diretta di questo teorema è facile. Sia P un DPDA che riconosce A. Allora il DPDA P' riconosce $A\dashv$ simulando P fino a quando P' legge $\dashv$. A questo punto, P' accetta se P era entrato in uno stato accettante leggendo il simbolo precedente. P' non legge alcun simbolo dopo $\dashv$.

Per provare la direzione inversa, sia P un DPDA che riconosce $A\dashv$ e costruiamo un DPDA P' che riconosce A. Mentre P' legge il suo input, esso simula P. Prima di leggere ogni simbolo di input, P' determina se P accetterebbe qualora quel simbolo fosse $\dashv$. Se così fosse, P' entrebbe in uno stato accettante. Osserva che P può agire sulla pila dopo che ha letto $\dashv$, quindi stabilire se accetta dopo aver letto $\dashv$ può dipendere dal contenuto della pila. Naturalmente, P' non è in grado di rimuovere simboli in tutta la pila su ogni simbolo di input, quindi deve stabilire cosa farebbe P dopo aver letto $\dashv$, ma senza rimuovere simboli dalla pila. Invece, P' memorizza informazioni supplementari sulla pila che consente a P' di stabilire subito se P accetterebbe. Questa informazione indica da quali stati P alla fine accetterebbe mentre (eventualmente) utilizza la pila, ma senza leggere ulteriore input.

DIMOSTRAZIONE. Diamo i dettagli solo della prova della direzione inversa. Come descritto nell'idea della prova, sia $P = (Q, \Sigma \cup \{\dashv\}, \Gamma, \delta, q_0, F)$ il DPDA che riconosce $A\dashv$ e costruiamo un DPDA $P' = (Q', \Sigma, \Gamma', \delta', q_0', F')$ che riconosce A. In primo luogo, modifichiamo P in modo che ciascuna delle sue mosse faccia esattamente una delle operazioni seguenti: leggere un simbolo di input; inserire un simbolo sulla pila; o eliminare un simbolo dalla pila. Fare questa modifica è semplice introducendo nuovi stati.

P' simula P, mantenendo una copia del suo contenuto della pila con informazione supplementare inserita sulla pila. Ogni volta che P' inserisce uno dei simboli di pila di P, P' fa seguire a questo il push di un simbolo che rappresenta un sottoinsieme dell'insieme degli stati di P. Quindi poniamo $\Gamma' = \Gamma \cup \mathcal{P}(Q)$. La pila in P' alterna elementi di Γ con elementi di $\mathcal{P}(Q)$. Se $R \in \mathcal{P}(Q)$ è il simbolo sulla cima della pila, allora se P parte da uno degli stati di R, P alla fine accetterà senza leggere più input.

Inizialmente, P' inserisce l'insieme R_0 sulla pila, dove R_0 contiene ogni stato q tale che, quando P si trova in q con la pila vuota, esso alla fine accetta senza leggere alcun simbolo di input. Poi P' inizia a simulare P. Per simulare una mossa di pop, P' prima rimuove ed elimina l'insieme degli stati che appare come simbolo in cima alla pila, poi esegue nuovamente una mossa di pop per ottenere il simbolo che P avrebbe rimosso a questo punto e lo usa per stabilire la mossa successiva di P. Per simulare una mossa di push $\delta(q, a, \varepsilon) = (r, x)$, in cui P inserisce x quando passa dallo stato q allo stato r, procediamo come segue. Innanzitutto P' esamina l'insieme degli stati R sulla cima della sua pila, e poi inserisce x e dopo l'insieme S, dove $q \in S$ se $q \in F$ o se $\delta(q, \varepsilon, x) = (r, \varepsilon)$ ed $r \in R$. In altre parole, S è l'insieme degli stati che sono accettanti o che condurrebbero a uno stato in R dopo aver eliminato x. Infine, P' simula una mossa di lettura $\delta(q, a, \varepsilon) = (r, \varepsilon)$, esaminando l'insieme R sulla cima della pila ed entrando in uno stato accettante se $r \in R$. Se P' è alla fine della stringa di input quando esso entra in questo stato, accetterà l'input. Se non è alla fine della stringa di input, continuerà a simulare P, quindi questo stato accettante deve anche memorizzare lo stato di P. Pertanto creiamo questo stato come una seconda copia dello stato originario di P, contrassegnandolo come uno stato accettante in P'.

Grammatiche context-free deterministiche

In questa sezione definiamo le grammatiche context-free deterministiche, la controparte agli automi a pila deterministici. Mostreremo che questi modelli sono computazionalmente equivalenti, purché limitiamo la nostra attenzione ai linguaggi con simbolo di fine stringa, dove le stringhe terminano con ⊣. Quindi la corrispondenza non è esattamente così forte come vedemmo per le espressioni regolari e gli automi finiti, o per le CFG e i PDA, dove il modello generativo e il modello riconoscitivo descrivono esattamente la stessa classe di linguaggi senza necessità di simboli di fine stringa. Comunque, nel caso dei DPDA e delle DCFG, i simboli di fine stringa sono necessari perché altrimenti l'equivalenza non sussiste.

In un automa deterministico, ogni passo in una computazione determina il passo successivo. L'automa non può fare scelte su come procedere poiché una sola possibilità è permessa in ogni momento. Per definire il determinismo in una grammatica, nota che le computazioni negli automi corrispondono alle derivazioni nelle grammatiche. In una grammatica deterministica, le derivazioni sono vincolate, come vedrai.

Le derivazioni nelle CFG iniziano con la variabile iniziale e procedono in modo "top down" con una serie di sostituzioni in base alle regole della grammatica, finché la derivazione conduce a una stringa di terminali. Per definire le DCFG adotteremo un approccio "bottom up", iniziando con una stringa di terminali ed eseguendo la derivazione in ordine inverso, usando una serie di passi di riduzione fino a raggiungere la variabile iniziale. Ogni ***passo di riduzione*** è una sostituzione in ordine inverso, attraverso la quale la stringa di terminali e variabili sul lato destro di una regola è sostituita dalla variabile sul corrispondente lato sinistro. La stringa sostituita è chiamata la ***stringa di riduzione***. L'intera derivazione in ordine inverso è chiamata una ***riduzione***. Le CFG deterministiche sono definite in termini di riduzioni che hanno una certa proprietà.

Più formalmente, se u e v sono stringhe di variabili e terminali, scriviamo $u \rightarrowtail v$ per intendere che v può essere ottenuta da u mediante un passo di riduzione. In altre parole, $u \rightarrowtail v$ ha lo stesso significato di $v \Rightarrow u$. Una ***riduzione da u a v*** è una sequenza

$$u = u_1 \rightarrowtail u_2 \rightarrowtail \ldots \rightarrowtail u_k = v,$$

diciamo che ***u è riducibile a v*** e lo denotiamo con $u \overset{*}{\rightarrowtail} v$. Quindi $u \overset{*}{\rightarrowtail} v$ quando $v \overset{*}{\Rightarrow} u$. Una ***riduzione da u*** è una riduzione da u alla variabile iniziale. In una ***riduzione a sinistra***, ogni stringa di riduzione è ridotta solo dopo tutte le altre stringhe di riduzione che si trovano interamente alla sua sinistra. Con una piccola riflessione possiamo vedere che una riduzione sinistra è una derivazione destra in ordine inverso.

Ecco l'idea dietro il determinismo nelle CFG. In una CFG con variabile iniziale S e tale che w sia una stringa nel suo linguaggio, poniamo che una riduzione a sinistra di w sia

$$w = u_1 \rightarrowtail u_2 \rightarrowtail \ldots \rightarrowtail u_k = S.$$

In primo luogo, stabiliamo che ogni u_i determina il successivo passo di riduzione e quindi u_{i+1}. Quindi w determina tutta la sua riduzione sinistra. Questo requisito implica solo che la grammatica è non ambigua. Per ottenere il determinismo, abbiamo bisogno di andare oltre. In ogni u_i, il successivo passo di riduzione deve essere univocamente determinato dal prefisso di u_i fino alla stringa di riduzione h di quel passo di riduzione compresa. In altre parole, il passo di riduzione sinistra in u_i non dipende dai simboli in u_i alla destra della sua stringa di riduzione.

Introdurre una terminologia ci aiuterà a precisare questa idea. Sia w una stringa nel linguaggio di una CFG G, e sia u_i una stringa che è presente in una riduzione sinistra di w. Poniamo che nel passo di riduzione $u_i \rightarrowtail u_{i+1}$, la

regola $T \to h$ sia stata applicata all'inverso. Questo significa che possiamo scrivere $u_i = xhy$ e $u_{i+1} = xTy$, dove h è la stringa di riduzione, x è la parte di u_i che compare a sinistra di h, e y è la parte di u_i che compare a destra di h. Attraverso un disegno,

$$u_i = \overbrace{x_1 \cdots x_j}^{x}\overbrace{h_1 \cdots h_k}^{h}\overbrace{y_1 \cdots y_l}^{y} \rightarrowtail \overbrace{x_1 \cdots x_j}^{x}\overbrace{T}^{T}\overbrace{y_1 \cdots y_l}^{y} = u_{i+1}.$$

FIGURA 2.44
Visione estesa di $xhy \rightarrowtail xTy$

Chiamiamo h, con la sua regola di riduzione $T \to h$, una maniglia o ***handle*** di u_i. In altre parole, un handle di una stringa u_i, presente in una riduzione sinistra di $w \in L(G)$, è l'occorrenza della stringa di riduzione in u_i, insieme alla regola di riduzione per u_i in questa riduzione. Occasionalmente associeremo l'handle solo alla stringa di riduzione, quando non siamo interessati alla regola di riduzione. Una stringa che compare in una riduzione sinistra di qualche stringa in $L(G)$ è chiamata una ***stringa valida***. Definiamo gli handle solo per le stringhe valide.

Una stringa valida può avere diversi handle, ma solo se la grammatica è ambigua. Le grammatiche non ambigue possono generare stringhe che hanno un solo albero sintattico e perciò anche le riduzioni sinistre, e quindi gli handle, sono unici. In questo caso, possiamo riferirci a "*l'* handle" di una stringa valida.

Osserva che y, la parte di u_i che segue un handle, è sempre una stringa di terminali poiché la riduzione è sinistra. Altrimenti, y conterrebbe un simbolo di variabile e questo potrebbe apparire solo da un precedente passo di riduzione la cui stringa di riduzione era tutta alla destra di h. Ma allora la riduzione sinistra dovrebbe aver ridotto l'handle in un passo precedente.

ESEMPIO 2.45

Consideriamo la grammatica G_1:

$$\begin{aligned} R &\to S \mid T \\ S &\to \mathtt{a}S\mathtt{b} \mid \mathtt{ab} \\ T &\to \mathtt{a}T\mathtt{bb} \mid \mathtt{abb} \end{aligned}$$

Il suo linguaggio è $B \cup C$ dove $B = \{\mathtt{a}^m\mathtt{b}^m \mid m \geq 1\}$ e $C = \{\mathtt{a}^m\mathtt{b}^{2m} \mid m \geq 1\}$. In questa riduzione sinistra della stringa $\mathtt{aaabbb} \in L(G_1)$, abbiamo sottolineato l'handle a ogni passo:

$$\mathtt{aa\underline{ab}bb} \rightarrowtail \mathtt{a\underline{aSbb}} \rightarrowtail \underline{\mathtt{aSb}} \rightarrowtail \underline{S} \rightarrowtail R.$$

Analogamente, questa è una riduzione a sinistra della stringa $\mathtt{aaabbbbbb}$:

$$\mathtt{aa\underline{abb}bbbb} \rightarrowtail \mathtt{a\underline{aTbb}bb} \rightarrowtail \underline{\mathtt{aTbb}} \rightarrowtail \underline{T} \rightarrowtail R.$$

In emtrambi i casi, si verifica che la riduzione sinistra mostrata sia la sola riduzione possibile; ma in altre grammatiche dove possono presentarsi diverse riduzioni, dobbiamo usare una riduzione sinistra per definire gli handle. Nota che gli handle di $\mathtt{aaabbb}$ e $\mathtt{aaabbbbbb}$ sono diversi, anche se le parti iniziali di queste stringhe coincidono. Discuteremo più dettagliatamente questo punto tra poco quando definiremo le DCFG.

Un PDA può riconoscere $L(G_1)$ usando il suo non determinismo per indovinare se il suo input è in B o in C. Poi, dopo che ha inserito le a sulla pila, rimuove le a e fa corrispondere ciascuna di esse con b o bb a seconda dell'ipotesi. Il Problema 2.25 chiede di mostrare che $L(G_1)$ non è un DCFL. Se provi a costruire un DPDA che riconosce questo linguaggio, vedrai che la macchina non può sapere in anticipo se l'input è in B o in C quindi non sa come abbinare le a con le b. Mostra la differenza tra questa grammatica e la grammatica G_2:

$$\begin{aligned} R &\to \mathtt{1}S \mid \mathtt{2}T \\ S &\to \mathtt{a}S\mathtt{b} \mid \mathtt{ab} \\ T &\to \mathtt{a}T\mathtt{bb} \mid \mathtt{abb} \end{aligned}$$

dove il primo simbolo nell'input fornisce questa informazione. La nostra definizione delle DCFG deve includere G_2 ma anche escludere G_1.

ESEMPIO 2.46

Sia G_3 la grammatica seguente:

$$\begin{aligned} S &\to T\dashv \\ T &\to T(T) \mid \varepsilon \end{aligned}$$

Questa grammatica illustra diverse caratteristiche. In primo luogo, essa genera un linguaggio con simbolo di fine stringa. Noi ci concentreremo sui linguaggi marcati più tardi quando proveremo l'equivalenza tra i DPDA e le DCFG. Inoltre, nelle riduzioni si possono presentare handle uguali a ε, come indicato con piccoli trattini in basso nella riduzione sinistra della stringa $()()\dashv$:

$$\_()()\dashv \rightarrowtail T(\_)()\dashv \rightarrowtail \underline{T(T)}()\dashv \rightarrowtail T(\_)\dashv \rightarrowtail \underline{T(T)}\dashv \rightarrowtail \underline{T\dashv} \rightarrowtail S.$$

Gli handle giocano un ruolo importante nel definire le DCFG poiché gli handle determinano le riduzioni. Non appena conosciamo l'handle di una stringa, conosciamo il successivo passo di riduzione. Per capire il senso della prossima definizione, ricorda il nostro obiettivo: puntiamo a definire le DCFG in modo che esse corrispondano ai DPDA. Stabiliremo questa corrispondenza mostrando come trasformare le DCFG in DPDA equivalenti e viceversa. Affinché questa conversione funzioni, il DPDA ha bisogno di trovare gli handle in modo che possa trovare le riduzioni. Ma trovare un handle può essere complicato. Sembra che abbiamo bisogno di conoscere il

passo successivo di riduzione di una stringa per identificare il suo handle, ma un DPDA non conosce la riduzione in anticipo. Risolveremo questo problema limitando gli handle in una DCFG così che il DPDA possa trovarli più facilmente.

Per formulare la definizione, consideriamo le grammatiche ambigue, dove alcune stringhe hanno diversi handle. Scegliere uno specifico handle può richiedere di conoscere in anticipo quale albero sintattico deriva la stringa, informazione che è certamente non disponibile al DPDA. Vedremo che le DCFG sono non ambigue cosicché gli handle sono unici. Comunque, l'unicità da sola non basta per definire le DCFG come mostra la grammatica G_1 nell'Esempio 2.45.

Perché handle unici non implicano che abbiamo una DCFG? La risposta è evidente esaminando gli handle in G_1. Se $w \in B$, l'handle è ab, mentre se $w \in C$, l'handle è abb. Sebbene w determini quali di questi casi applicare, scoprire quale tra ab e abb è l'handle può richiedere di esaminare tutta w, e un DPDA non deve leggere l'intero input quando deve scegliere l'handle.

Per definire le DCFG che corrispondono ai DPDA, imponiamo un requisito più forte agli handle. La parte iniziale di una stringa valida, fino al suo handle compreso, deve essere sufficiente a determinare l'handle. Quindi, se stiamo leggendo una stringa valida da sinistra a destra, non appena leggiamo l'handle sappiamo che l'abbiamo. Non abbiamo bisogno di leggere oltre l'handle per identificare l'handle. Ricorda che la parte non letta della stringa valida contiene solo terminali perché la stringa valida è stata ottenuta da una riduzione sinistra di una stringa iniziale di terminali, e la parte non letta non è stata ancora elaborata. In accordo con quanto detto prima, diciamo che un handle h di una stringa valida $v = xhy$ è un ***handle obbligato*** se h è l'unico handle in *ogni* stringa valida $xh\hat{y}$ dove $\hat{y} \in \Sigma^*$.

DEFINIZIONE **2.47**

Una ***grammatica context-free deterministica*** è una grammatica context-free tale che ogni stringa valida ha un handle obbligato.

Per semplicità, assumeremo in tutta questa sezione sui linguaggi context-free deterministici che la variabile iniziale di una CFG non è presente nel lato destro di una qualsiasi regola e che ogni variabile in una grammatica appare in una riduzione di qualche stringa nel linguaggio della grammatica, cioè le grammatiche non contengono variabili inutili.

Nonostante la nostra definizione di DCFG sia matematicamente precisa, essa non fornisce un modo esplicito per determinare se una CFG è deterministica. In seguito presenteremo una procedura che fa esattamente questo, chiamata il DK-test. Useremo la costruzione che è alla base del DK-test per permettere a un DPDA di trovare gli handle, quando mostriamo come trasformare una DCFG in un DPDA.

Il DK-test si basa su un semplice ma sorprendente fatto. Per ogni CFG G possiamo costruire un DFA associato DK che è in grado di identificare gli handle. Precisamente, DK accetta il suo input z se

1. z è prefisso di qualche stringa valida $v = zy$ e
2. z termina con un handle di v.

Inoltre, ogni stato accettante di DK indica la regola (o le regole) di riduzione associata. In una arbitraria CFG possono essere applicate più regole di riduzioni, a seconda di quale stringa valida v estenda z. Ma in una DCFG, come vedremo, ogni stato accettante corrisponde a esattamente una regola di riduzione.

Descriveremo il DK-test dopo aver presentato DK formalmente e stabilito le sue proprietà, ma ecco il piano. In una DCFG, tutti gli handle sono obbligati. Quindi se zy è una stringa valida con un prefisso z che termina in un handle di zy, questo handle è unico ed esso è anche l'handle per tutte le stringhe valide $z\hat{y}$. Affinché queste proprietà valgano, ciascuno degli stati accettanti di DK deve essere associato a un solo handle e quindi a una sola regola di riduzione applicabile. Inoltre, lo stato accettante non deve avere un cammino uscente che conduce a uno stato accettante leggendo una stringa in Σ^*. Altrimenti, l'handle di zy non sarebbe unico o dipenderebbe da y. Nel DK-test, costruiamo DK e poi concludiamo che G è deterministica se tutti i suoi stati accettanti hanno queste proprietà.

Per costruire il DFA DK, costruiremo un NFA K equivalente e trasformeremo K in DK[1] attraverso la costruzione per sottoinsiemi introdotta nel Teorema 1.39. Per comprendere K, consideriamo prima un NFA J che realizza un compito più semplice. Esso accetta ogni stringa di input che termina con il lato destro di una regola. Costruire J è facile. Ipotizza quale regola usare e ipotizza anche il punto in cui iniziare a far corrispondere l'input con il lato destro di quella regola. Mentre abbina l'input, J conserva traccia del suo avanzamento attraverso il lato destro scelto. Rappresentiamo questo progresso mettendo un puntino nel corrispondente punto della regola, producendo una ***regola marcata***, chiamata anche ***item*** in alcuni altri libri che trattano questi argomenti. Per ogni regola $B \to u_1u_2\cdots u_k$ con k simboli sul lato destro, otteniamo $k+1$ regole marcate:

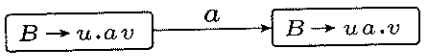
$$\begin{aligned} B &\to .u_1u_2\cdots u_k \\ B &\to u_1.u_2\cdots u_k \\ &\vdots \\ B &\to u_1u_2\cdots.u_k \\ B &\to u_1u_2\cdots u_k. \end{aligned}$$

[1] Il nome DK serve a ricordare "K deterministico" ma sta anche per Donald Knuth, che per primo propose questa idea.

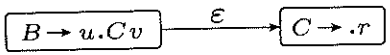
Ciascuna di queste regole marcate corrisponde a uno stato di J. Indichiamo lo stato associato alla regola marcata $B \to u.v$ con un rettangolo intorno, $\boxed{B \to u.v}$. Gli stati accettanti $\boxed{\boxed{B \to u.}}$ corrispondono alle ***regole completate*** che hanno il punto alla fine. Aggiungiamo uno stato iniziale a parte con un self-loop su ogni simbolo e una ε-mossa a $\boxed{B \to .u}$ per ogni regola $B \to u$. Quindi J accetta se l'abbinamento si completa con successo alla fine dell'input. Se si presenta una coppia male abbinata o se la fine dell'abbinamento non coincide con la fine dell'input, questo ramo della computazione di J rifiuta.

NFA K opera in modo simile, ma è più prudente sulla scelta di una regola per l'abbinamento. Sono consentite solo potenziali regole di riduzione. Come J, i suoi stati corrispondono a tutte le regole marcate. Esso ha uno stato iniziale speciale che ha una ε-mossa a $\boxed{S_1 \to .u}$ per ogni regola che coinvolge la variabile iniziale S_1. Su ogni ramo della sua computazione, K fa corrispondere una potenziale regola di riduzione con una sottostringa dell'input. Se il lato destro di quella regola contiene una variabile, K può passare non deterministicamente a qualche regola che espande quella variabile. Il Lemma 2.48 formalizza questa idea. In primo luogo descriviamo dettagliatamente K.

Le transizioni ricadono in due categorie: mosse di spostamento (o shift-mosse) ed ϵ-mosse. Le mosse di spostamento si presentano per ogni terminale o variabile a e ogni regola $B \to uav$:



Le ϵ-mosse si presentano per tutte le regole $B \to uCv$ e $C \to r$:



Gli stati accettanti sono tutti i $\boxed{\boxed{B \to u.}}$, corrispondenti a una regola completata. Gli stati accettanti non hanno transizioni uscenti e sono denotati con un doppio rettangolo.

Il lemma seguente e il suo corollario mostrano che K accetta tutte le stringhe z che terminano con degli handle per qualche estensione valida di z. Poiché K è non deterministico, diciamo che "può" entrare in uno stato per intendere che K entra in quello stato su qualche ramo del suo non determinismo.

LEMMA 2.48

K può entrare nello stato $\boxed{T \to u.v}$ leggendo l'input z se, e solo se, $z = xu$ e $xuvy$ è una stringa valida con handle uv e regola di riduzione $T \to uv$, per qualche $y \in \Sigma^*$.

IDEA. K opera abbinando il lato destro di una regola scelta con una parte dell'input. Se la corrispondenza si completa con successo, accetta. Se quel lato destro contiene una variabile C, si può presentare una delle seguenti due situazioni. Se C è il simbolo di input seguente, allora semplicemente continua l'abbinamento con la regola scelta. Se C è stata trasformata, l'input conterrà simboli derivati da C, quindi K sceglie non deterministicamente una regola di sostituzione per C e inizia l'abbinamento dall'inizio del lato destro di quella regola. Accetta quando il lato destro della regola scelta in quel momento è stato completamente abbinato.

DIMOSTRAZIONE. Proviamo prima la parte diretta. Assumiamo che K su w sia in $\boxed{T \to u.v}$. Esaminiamo il cammino di K dal suo stato iniziale a $\boxed{T \to u.v}$. Pensa al cammino come esecuzione di sequenze (o run) di mosse di spostamento separate da ε-mosse. Le mosse di spostamento sono transizioni tra stati che condividono la stessa regola, spostando il punto verso destra su simboli letti dall'input. Poniamo che la regola nell'$i$esimo run sia $S_i \to u_i S_{i+1} v_i$, dove S_{i+1} è la variabile trasformata nel run successivo. Il penultimo run è per la regola $S_l \to u_l T v_l$, e il run finale ha regola $T \to uv$.

L'input z deve allora essere uguale a $u_1 u_2 \ldots u_l u = xu$ perché le stringhe u_i e u erano i simboli letti dall'input nelle mosse di riduzione. Ponendo $y' = v_l \ldots v_2 v_1$, vediamo che $xuvy'$ è derivabile in G perché le regole summenzionate danno una derivazione come mostrato nell'albero sintattico illustrato nella Figura 2.49.

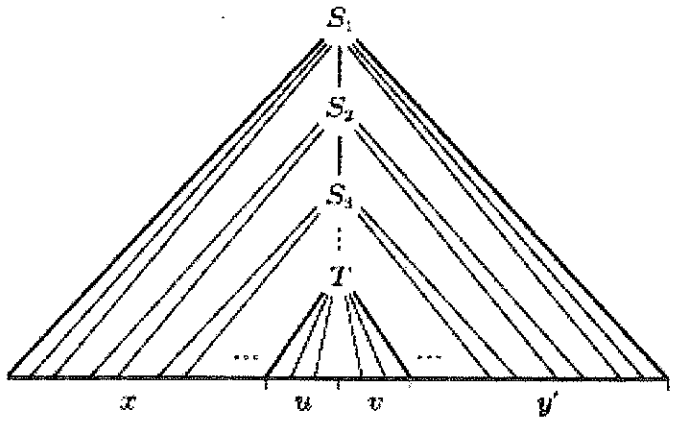


FIGURA 2.49
Albero sintattico che conduce a $xuvy'$

Per ottenere una stringa valida, espandiamo completamente tutte le variabili che compaiono in y' fino a quando ciascuna variabile deriva qualche stringa di terminali, e chiamiamo y la stringa risultante. La stringa $xuvy$

è valida perché si presenta in una riduzione a sinistra di $w \in L(G)$, una stringa di terminali ottenuta espandendo completamente tutte le variabili in $xuvy$.

Come è evidente dalla figura sottostante, uv è l'handle nella riduzione e la sua regola di riduzione è $T \to uv$.

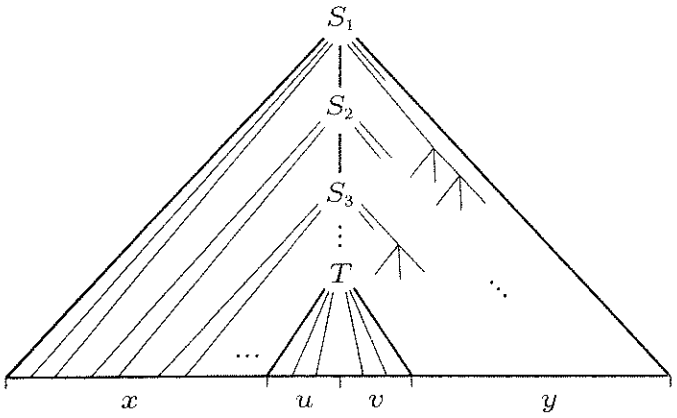


FIGURA 2.50
Albero sintattico che conduce alla stringa valida $xuvy$ con handle uv

Ora proviamo la direzione inversa del lemma. Assumiamo che la stringa $xuvy$ sia una stringa valida con handle uv e regola di riduzione $T \to uv$. Mostriamo che K sull'input xu può entrare nello stato $\boxed{T \to u.v}$.

L'albero sintattico per $xuvy$ è nella figura precedente. È radicato nella variabile iniziale S_1 e deve contenere la variabile T poiché $T \to uv$ è il primo passo di riduzione nella riduzione di $xuvy$. Siano $S_2, \ldots, S_l$ le variabili sul cammino da S_1 a T come mostrato. Nota che tutte le variabili nell'albero sintattico che compaiono a sinistra di questo cammino devono essere non espanse, altrimenti uv non sarebbe l'handle.

In questo albero sintattico, ciascun S_i conduce a S_{i+1} mediante qualche regola $S_i \to u_i S_{i+1} v_i$. Quindi la grammatica deve contenere le seguenti regole, dove u_i e v_i sono stringhe.

$$\begin{aligned} S_1 &\to u_1 S_2 v_1 \\ S_2 &\to u_2 S_3 v_2 \\ &\vdots \\ S_l &\to u_l T v_l \\ T &\to uv \end{aligned}$$

K contiene il cammino seguente dal suo stato iniziale allo stato $\boxed{T \to u.v}$ leggendo l'input $z = xu$. In primo luogo, K fa una ϵ-mossa verso $\boxed{S_1 \to .u_1 S_2 v_1}$. Poi, mentre legge i simboli di u_1, compie le corrispondenti mosse di spostamento finché entra nello stato $\boxed{S_1 \to u_1 . S_2 v_1}$ alla fine di u_1. Poi fa una ϵ-mossa verso $\boxed{S_2 \to .u_2 S_3 v_2}$ e continua con mosse di spostamento leggendo u_2 finché raggiunge $\boxed{S_2 \to u_2 . S_3 v_2}$ e così via. Dopo aver letto u_l entra in $\boxed{S_l \to u_l . T v_l}$ che porta mediante una ϵ-mossa a $\boxed{T \to .uv}$ e infine dopo aver letto u è in $\boxed{T \to u.v}$.

Il corollario seguente mostra che K accetta tutte le stringhe che terminano con un handle di qualche estensione valida. Esso segue dal Lemma 2.48 prendendo $u = h$ e $v = \varepsilon$.

COROLLARIO 2.51

K può entrare nello stato accettante $\boxed{T \to h.}$ sull'input z se e solo se $z = xh$ e h è un handle di qualche stringa valida xhy con regola di riduzione $T \to h$.

Infine, trasformiamo l'NFA K nel DFA DK usando la costruzione per sottoinsiemi nella prova del Teorema 1.39 a pagina 58 e poi rimuovendo tutti gli stati che non sono raggiungibili dallo stato iniziale. Ciascuno degli stati di DK quindi contiene una o più regole marcate. Ciascun stato accettante contiene almeno una regola completata. Noi possiamo applicare il Lemma 2.48 e il Corollario 2.51 a DK riferendoci agli stati che contengono le regole marcate indicate.

Ora siamo pronti a descrivere il ***DK-test***.

Iniziamo con una CFG G e costruiamo il DFA associato DK. Determiniamo se G è deterministica esaminando gli stati accettanti di DK. Il DK-test stabilisce che ogni stato accettante contiene

1. esattamente una regola completata e
2. nessuna regola marcata con un punto in cui un simbolo terminale segue immediatamente il punto, cioè nessuna regola marcata della forma $B \to u.av$ for $a \in \Sigma$.

TEOREMA 2.52

G supera il DK-test se e solo se G è una DCFG.

IDEA. Mostreremo che il DK-test è superato se e solo se tutti gli handle sono obbligati. Equivalentemente, il test fallisce se e solo se qualche handle

non è obbligato. In primo luogo, supponiamo che qualche stringa valida ha un handle non obbligato. Se eseguiamo DK su questa stringa, il Corollario 2.51 ci dice che DK entra in uno stato accettante alla fine dell'handle. Il DK-test fallisce perché quello stato accettante ha una seconda regola completata oppure un cammino uscente che conduce a uno stato accettante, dove il cammino uscente inizia con un simbolo terminale. Nel secondo caso, lo stato accettante conterrebbe una regola marcata con un simbolo terminale che segue il punto.

Viceversa, se il DK-test fallisce perché uno stato accettante ha due regole completate, estendi la stringa associata a due stringhe valide con differenti handle in quel punto. Analogamente, se ha una regola completata e una regola marcata con un simbolo terminale che segue il punto, usa il Lemma 2.48 per ottenere due estensioni valide con differenti handle in quel punto. Costruire l'estensione valida corrispondente alla seconda regola è un pó delicato.

DIMOSTRAZIONE. Iniziamo con la direzione diretta. Assumiamo che G non sia deterministica e mostriamo che essa non supera il DK-test. Prendiamo una stringa valida xhy che ha un handle h non obbligato. Quindi qualche stringa valida xhy' ha un diverso handle $\hat{h} \neq h$, dove y' è una stringa di terminali. Possiamo allora scrivere xhy' come $xhy' = \hat{x}\hat{h}\hat{y}$.

Se $xh = \hat{x}\hat{h}$, le regole di riduzioni sono diverse perché h e $\hat{h}$ non sono lo stesso handle. Quindi, l'input xh manda DK in uno stato che contiene due regole completate, una violazione del DK-test.

Se $xh \neq \hat{x}\hat{h}$, una di queste estende l'altra. Assumiamo che xh sia un prefisso proprio di $\hat{x}\hat{h}$. Il ragionamento è lo stesso con le stringhe scambiate e y a posto di y', se $\hat{x}\hat{h}$ è la stringa più corta. Sia q lo stato in cui DK entra sull'input xh. Lo stato q deve essere accettante perché h è un handle di xhy. Un arco di transizione deve uscire da q perché $\hat{x}\hat{h}$ manda DK in uno stato accettante passando per q. Inoltre, questo arco di transizione è etichettato con un simbolo terminale, perché $y' \in \Sigma^+$. In questo caso $y' \neq \varepsilon$ perché $\hat{x}\hat{h}$ estende xh. Quindi q contiene una regola marcata con un simbolo terminale che segue immediatamente il punto, il che viola DK-test.

Per provare la direzione inversa, assumiamo che G non superi il DK-test in qualche stato accettante q e mostriamo che G non è deterministica esibendo un handle non obbligato. Poiché q è accettante, esso ha una regola completata $T \to h.$. Sia z una stringa che conduce DK a q. Allora $z = xh$ dove qualche stringa valida xhy ha un handle h con regola di riduzione $T \to h$, per $y \in \Sigma^*$. Ora consideriamo due casi, che dipendono da come il DK-test fallisce.

In primo luogo, supponiamo che q abbia un'altra regola completata $B \to \hat{h}.$. Allora qualche stringa valida xhy' deve avere un differente handle $\hat{h}$ con regola di riduzione $B \to \hat{h}$. Quindi, h non è un handle obbligato.

In secondo luogo, supponiamo che q contenga una regola $B \to u.av$ dove $a \in \Sigma$. Poiché xh porta DK a q, abbiamo $xh = \hat{x}u$, dove $\hat{x}uav\hat{y}$ è valida e ha un handle uav con regola di riduzione $B \to uav$, per qualche $\hat{y} \in \Sigma^*$. Per mostrare che h è non obbligato, espandiamo completamente tutte le variabili in v ottenendo come risultato $v' \in \Sigma^*$, poi poniamo $y' = av'\hat{y}$ e notiamo che $y' \in \Sigma^*$. La riduzione sinistra seguente mostra che xhy' è una stringa valida e h non è l'handle.

$$xhy' = xhav'\hat{y} = \hat{x}\,uav'\,\hat{y} \overset{*}{\rightarrowtail} \hat{x}\,uav\,\hat{y} \rightarrowtail \hat{x}\,B\,\hat{y} \overset{*}{\rightarrowtail} S$$

dove S è la variabile iniziale. Sappiamo che $\hat{x}\,uav\,\hat{y}$ è valida e possiamo ottenere $\hat{x}\,uav'\hat{y}$ da essa usando una derivazione destra quindi anche $\hat{x}\,uav'\,\hat{y}$ è valida. Inoltre, l'handle di $\hat{x}\,uav'\hat{y}$ o è all'interno di v' (se $v \neq v'$) o è uav (se $v = v'$). In entrambi i casi, l'handle include a o segue a e quindi non può essere h perché h precede a. Quindi h non è un handle obbligato.

Nella pratica, quando si costruisce il DFA DK, una costruzione diretta può essere più rapida che costruire prima l'NFA K. Iniziamo aggiungendo un punto nella posizione iniziale di tutte le regole che coinvolgono la variabile iniziale e mettiamo queste regole ora marcate nello stato iniziale di DK. Se un punto precede una variabile C in una di queste regole, mettiamo punti nella posizione iniziale in tutte le regole che hanno C sul lato sinistro e aggiungiamo queste regole allo stato, continuando questo processo fino a non ottenere più nuove regole marcate. Per ogni simbolo c che segue un punto, aggiungiamo un arco uscente etichettato c a un nuovo stato contenente le regole marcate ottenute spostando il punto dopo c in ognuna delle regole marcate dove il punto precede c, e aggiungendo regole corrispondenti alle regole dove un punto precede una variabile come in precedenza.

ESEMPIO 2.53

Qui illustriamo come il DK-test fallisce per la grammatica seguente.

$$S \to E\dashv$$
$$E \to E + T \mid T$$
$$T \to T \times \mathtt{a} \mid \mathtt{a}$$

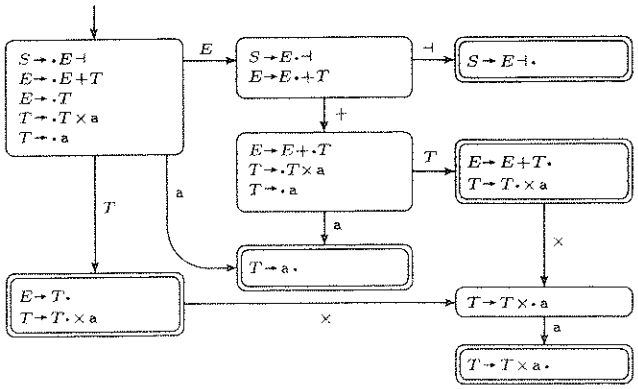


FIGURA 2.54
Esempio di un *DK*-test non superato

Nota i due stati problematici in basso a sinistra e il secondo dall'alto a destra, dove uno stato accettante contiene una regola marcata dove un simbolo terminale segue il punto.

ESEMPIO 2.55

Ecco il DFA DK che mostra che la grammatica segente è una DCFG.

$$S \to T\dashv$$
$$T \to T(T) \mid \varepsilon$$

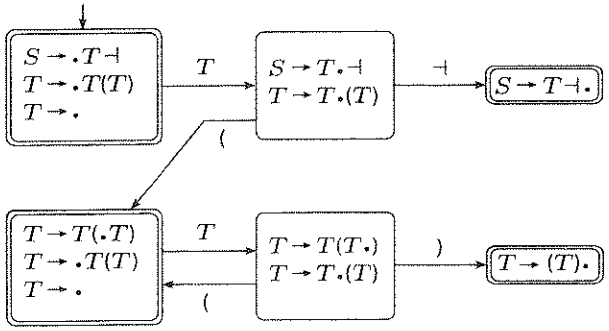


FIGURA 2.56
Esempio di un *DK*-test superato

Osserva che tutti gli stati accettanti soddisfano le condizioni del DK-test.

Relazione tra i DPDA e le DCFG

In questa sezione mostreremo che i DPDA e le DCFG descrivono la stessa classe e cioè quella dei linguaggi marcati. In primo luogo, dimostreremo come trasformare le DCFG in DPDA equivalenti. Questa conversione funziona in tutti i casi. In secondo luogo, mostreremo come fare la conversione inversa, dai DPDA alle DCFG equivalenti. La seconda trasformazione funziona solo per i linguaggi marcati. Limitiamo l'equivalenza ai linguaggi marcati, perché i modelli non sono equivalenti senza questa restrizione. Abbiamo mostrato in precedenza che i simboli di fine stringa non hanno l'effetto di cambiare la classe dei linguaggi che i DPDA riconoscono, ma essi influenzano la classe di linguaggi che le DCFG generano. Senza simboli di fine stringa, le DCFG generano solo una sottoclasse dei DCFL – quelli che sono prefissi (vedi il Problema 2.22). Osserva che ogni linguaggio marcato è prefisso.

TEOREMA 2.57

Un linguaggio marcato è generato da una grammatica context-free deterministica se e solo se è context-free deterministico.

Dobbiamo provare l'enunciato in entrambe le due direzioni. Prima mostreremo che per ogni DCFG esiste un DPDA equivalente. Infine mostreremo che per ogni DPDA che riconosce un linguaggio marcato esiste una DCFG equivalente. Tratteremo queste due direzioni in lemmi separati.

LEMMA 2.58

Per ogni DCFG esiste un DPDA equivalente.

IDEA. Mostriamo come trasformare una DCFG G in un DPDA equivalente P. P usa il DFA DK per operare come segue. Simula DK sui simboli che esso legge dall'input finché DK accetta. Come mostrato nella prova del Teorema 2.52, uno stato accettante di DK indica una specifica regola marcata perché G è deterministica, e quella regola identifica un handle per qualche stringa valida che estende l'input che ha visto finora. Inoltre, questo handle è un handle per *ogni* estensione valida poiché G è deterministica, e in particolare esso lo è per l'intero input a P, se quell'input è in $L(G)$. Quindi P può usare questo handle per identificare il primo passo di riduzione per la sua stringa di input, nonostante abbia letto in quel momento solo una parte del suo input.

Come P identifica il secondo e i successivi passi di riduzione? Un'idea è eseguire il passo di riduzione direttamente sulla stringa di input, e poi operare sull'input modificato attraverso DK come abbiamo fatto prima.

Ma l'input non puó essere modificato né riletto, quindi questa idea non funziona. Un'altra strategia sarebbe copiare l'input nella pila ed effettuare il passo di riduzione là, ma allora P avrebbe bisogno di fare operazioni di pop nell'intera pila per operare sull'input modificato attraverso DK e quindi l'input modificato non resterebbe disponibile per i passi successivi seguenti.

Il trucco qui è memorizzare gli stati di DK nella pila, invece di memorizzarvi la stringa di input. Ogni volta che P legge un simbolo di input e simula una mossa in DK, registra lo stato di DK inserendolo sulla pila. Quando compie un passo di riduzione usando la regola di riduzione $T \to u$, rimuove $|u|$ stati dalla pila, mostrando lo stato in cui era DK prima di leggere u. Riporta DK in quello stato, poi lo simula sull'input T e inserisce lo stato risultante sulla pila. In seguito P procede leggendo ed elaborando i simboli input come prima.

Quando P inserisce la variabile iniziale sulla pila, ha trovato una riduzione del suo input alla variabile iniziale, quindi entra in uno stato accettante.

Ora proviamo l'altra direzione del Teorema 2.57.

LEMMA 2.59

Per ogni DPDA che riconosce un linguaggio marcato esiste una DCFG equivalente.

IDEA. Questa prova è una variazione della costruzione nel Lemma 2.27 a pagina 124 che descrive la conversione di un PDA P in una CFG G equivalente. In questo caso P e G sono deterministici. Nell'idea della prova per il Lemma 2.27, abbiamo modificato P in modo che svuotasse la sua pila ed entrasse in uno specifico stato accettante q_{accept} quando accettava. Un PDA non può determinare direttamente che è alla fine del suo input, quindi P usava il suo non determinismo per ipotizzare di essere in quella situazione. Noi non vogliamo introdurre il non determinismo nel costruire il DPDA P. Invece usiamo l'assunzione che $L(P)$ sia un linguaggio marcato. Modifichiamo P in modo che svuoti la sua pila ed entri in q_{accept} quando entra in uno dei suoi stati accettanti originari dopo che ha letto il simbolo di fine stringa $\dashv$.

Dopo applichiamo la costruzione della grammatica per ottenere G. Applicare semplicemente la costruzione originaria a un DPDA produce una grammatica quasi deterministica perché le derivazioni della CFG corrispondono minuziosamente alle computazioni del DPDA. Questa grammatica non è deterministica per un aspetto marginale e riparabile.

La costruzione originaria introduce regole della forma $A_{pq} \to A_{pr}A_{rq}$ e queste possono causare ambiguità. Queste regole coprono il caso in cui A_{pq}

genera una stringa che porta P dallo stato p allo stato q con la pila vuota in entrambi gli stati, e la pila si svuota a metà strada. La sostituzione corrisponde a dividere la computazione in quel punto. Ma se la pila si svuota diverse volte, sono possibili diverse divisioni. Ciascuna di queste divisioni produce differenti alberi sintattici, quindi la grammatica risultante è ambigua. Risolviamo questo problema modificando la grammatica in modo che divida la computazione solo all'ultimissimo punto in cui la pila si svuota a metà strada, eliminando in tal modo l'ambiguità. Per esempio, una situazione simile ma più semplice si presenta nella grammatica ambigua

$$S \to T\dashv$$
$$T \to TT \mid (T) \mid \varepsilon$$

che è equivalente alla grammatica non ambigua e deterministica

$$S \to T\dashv$$
$$T \to T(T) \mid \varepsilon.$$

Poi mostriamo che la grammatica modificata è deterministica usando il DK-test. La grammatica è progettata per simulare il DPDA. Come provato nel Lemma 2.27, A_{pq} genera esattamente quelle stringhe su cui P va dallo stato p con pila vuota allo stato q con pila vuota. Proveremo il determinismo di G usando il determinismo di P che quindi risulterà utile per definire la computazione di P sulle stringhe valide osservando la sua azione sugli handle. Poi possiamo usare il comportamento deterministico di P per mostrare che gli handle sono obbligati.

DIMOSTRAZIONE. Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ e costruiamo G. La variabile iniziale è $A_{q_0,q_{\text{accept}}}$. La costruzione a pagina 124 contiene le parti 1, 2 e 3, ripetute qui per comodità.

1. Per ogni $p, q, r, s \in Q$, $u \in \Gamma$ e $a, b \in \Sigma_\varepsilon$, se $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε), poni la regola $A_{pq} \to aA_{rs}b$ in G.
2. Per ogni $p, q, r \in Q$, poni la regola $A_{pq} \to A_{pr}A_{rq}$ in G.
3. Per ogni $p \in Q$, poni la regola $A_{pp} \to \varepsilon$ in G.

Modifichiamo la costruzione per evitare di introdurre l'ambiguità, unendo le regole di tipo 1 e 2 in una singola regola di tipo 1-2 che realizza lo stesso risultato.

1-2. Per ogni $p, q, r, s, t \in Q$, $u \in \Gamma$ e $a, b \in \Sigma_\varepsilon$, se $\delta(r, a, \varepsilon) = (s, u)$ e $\delta(t, b, u) = (q, \varepsilon)$, poni la regola $A_{pq} \to A_{pr}aA_{st}b$ in G.

Per vedere che la grammatica modificata genera lo stesso linguaggio, consideriamo una qualsiasi derivazione nella grammatica iniziale. Per ogni sostituzione dovuta a una regola di tipo 2 $A_{pq} \to A_{pr}A_{rq}$, possiamo assumere che r è lo stato di P quando esso è nel punto più a destra in cui la

pila si svuota a metà strada, modificando la prova del Claim 2.31 a pagina 127 per scegliere r in questo modo. Poi la sostituzione successiva di A_{rq} deve espanderla usando una regola di tipo 1 $A_{rq} \to aA_{st}b$. Possiamo unire queste due sostituzioni in una singola regola di tipo 1-2 $A_{pq} \to A_{pr}aA_{st}b$.

Viceversa, in una derivazione usando la grammatica modificata, se sostituiamo ogni regola di tipo 1-2 $A_{pq} \to A_{pr}aA_{st}b$ con la regola di tipo 2 $A_{pq} \to A_{pr}A_{rq}$ seguita dalla regola di tipo 1 $A_{rq} \to aA_{st}b$, otteniamo lo stesso risultato.

Ora usiamo il DK-test per mostrare che G è deterministica. Per fare ciò, analizzeremo come P opera sulle stringhe valide estendendo il suo alfabeto di input e la funzione di transizione per elaborare simboli di variabile oltre ai simboli terminali. Aggiungiamo tutti i simboli A_{pq} all'alfabeto di input di P ed estendiamo la sua funzione di transizione δ definendo $\delta(p, A_{pq}, \varepsilon) = (q, \varepsilon)$. Poniamo tutte le altre transizioni che coinvolgono A_{pq} a $\emptyset$. Per preservare il comportamento deterministico di P, se P legge A_{pq} dall'input non permettiamo una mossa ε-input.

L'enunciato seguente si applica a ogni derivazione di una stringa w in $L(G)$ tale che

$$A_{q_0,q_{\text{accept}}} = v_0 \Rightarrow v_1 \Rightarrow \cdots \Rightarrow v_i \Rightarrow \cdots \Rightarrow v_k = w.$$

FATTO 2.60

Se P legge v_i che contiene una variabile A_{pq}, entra nello stato p subito prima di leggere A_{pq}.

La prova usa un'induzione su i, il numero di passi per derivare v_i da $A_{q_0,q_{\text{accept}}}$.

Base: $i = 0$.
In questo caso, $v_i = A_{q_0,q_{\text{accept}}}$ e P inizia nello stato q_0 quindi la base è vera.

Passo induttivo: Assumiamo l'enunciato vero per i e proviamolo per $i+1$. In primo luogo consideriamo il caso in cui $v_i = xA_{pq}y$ e A_{pq} è la variabile sostituita nel passo $v_i \Rightarrow v_{i+1}$. L'ipotesi induttiva implica che P entra nello stato p dopo aver letto x, prima di leggere il simbolo A_{pq}. In base alla costruzione di G, le regole di sostituzione possono essere di due tipi:

1. $A_{pq} \to A_{pr}aA_{st}b$ oppure
2. $A_{pp} \to \varepsilon$.

Quindi $v_{i+1} = xA_{pr}aA_{st}by$ oppure $v_{i+1} = xy$, a seconda di quale tipo di regola è stata usata. Nel primo caso, quando P legge $A_{pr}aA_{st}b$ in v_{i+1}, sa che inizia nello stato p, perché ha appena finito di leggere x. Mentre P legge $A_{pr}aA_{st}b$ in v_{i+1}, entra nella sequenza di stati r, s, t e q, a causa della costruzione della regola di sostituzione. Quindi, entra nello stato p subito prima di leggere A_{pr} ed entra nello stato s subito prima di leggere A_{st}, dimostrando così l'enunciato per queste due occorrenze di variabili. L'enunciato è vero per le occorrenze di variabili nella parte y poiché, dopo che P legge b, esso entra nello stato q e poi legge la stringa y. E anche sull'input v_i, entra in q subito prima di leggere y, quindi le computazioni coincidono sulle parti y di v_i e v_{i+1}. Ovviamente, le computazioni coincidono sulla parte x. Pertanto, l'enunciato è vero per v_{i+1}. Nel secondo caso, non viene introdotta nessuna nuova variabile, quindi dobbiamo solo osservare che le computazioni coincidono sulla parte x e y di v_i e v_{i+1}. Questo dimostra l'enunciato.

FATTO 2.61

G supera il DK-test.

Mostriamo che ciascuno degli stati accettanti di DK soddisfa i requisiti del DK-test. Scegliamo uno di questi stati accettanti. Esso contiene una regola completata R. Questa regola completata deve essere di una delle due forme seguenti:

1. $A_{pq} \to A_{pr}aA_{st}b$.
2. $A_{pp} \to$.

In entrambe le situazioni, dobbiamo mostrare che lo stato accettante non può contenere

a. un'altra regola completata e
b. una regola marcata che ha un simbolo terminale immediatamente dopo il punto.

Consideriamo ciascuno di questi quattro casi separatamente. In ogni caso, iniziamo considerando una stringa z su cui DK va nello stato accettante che noi abbiamo scelto prima.

Caso 1a. In questo caso R è una regola completata di tipo 1-2. Per ogni regola in questo stato accettante, z deve terminare con i simboli che precedono il punto in quella regola perché DK va in quello stato su z. Quindi i simboli che precedono il punto devono essere in accordo in tutte queste regole. Questi simboli sono $A_{pr}aA_{st}b$ in R quindi ogni altra regola completata di tipo 1-2 deve avere esattamente gli stessi simboli sul lato destro. Ne consegue che anche le variabili sul lato sinistro devono coincidere, quindi le regole devono essere le stesse.
Supponiamo che lo stato accettante contenga R e qualche ε-regola completata T di tipo 3. Dall'ipotesi su R sappiamo che z termina con $A_{pr}aA_{st}b$. Inoltre sappiamo che P effettua un pop sulla pila proprio alla fine di z perché un pop si verifica a quel punto in R, a causa della costruzione di G. In base al modo con cui costruiamo DK, una ε-regola completata in uno

stato deve derivare da una regola marcata che sta nello stesso stato, dove il punto non è proprio all'inizio e il punto viene subito prima di qualche variabile. (Un'eccezione si verifica nello stato iniziale di DK, dove questo punto può essere presente all'inizio della regola, ma questo stato accettante non può essere lo stato iniziale poiché contiene una regola completata di tipo 1-2.) In G, questo significa che T deriva da una regola marcata di tipo 1-2 in cui il punto precede la seconda variabile. Dalla costruzione di G si deve verificare un push subito prima del punto. Questo implica che P effettua una mossa di push proprio alla fine di z, contraddicendo la nostra affermazione precedente. Quindi la ϵ-regola completata T non può esistere. In entrambi i casi, una seconda regola completata di qualunque tipo non può essere presente in questo stato accettante.

Caso 2a. In questo caso R è una ϵ-regola completata $A_{pp} \rightarrow .$. Mostriamo che nessun'altra ϵ-regola completata $A_{qq} \rightarrow .$ può coesistere con R. Se così fosse, l'enunciato precedente mostra che P deve essere in p dopo aver letto z e deve anche essere in q dopo aver letto z. Quindi $p = q$ e allora le due ϵ-regole completate sono uguali.

Caso 1b. In questo caso R è una regola completata di tipo 1-2. Dal Caso 1a, sappiamo che P effettua un pop sulla pila alla fine di z. Supponiamo che lo stato accettante contenga anche una regola marcata T in cui un simbolo terminale segue immediatamente il punto. Dall'ipotesi su T sappiamo che P non effettua un pop sulla pila alla fine di z. Tale contraddizione mostra che questa situazione non può verificarsi.

Caso 2b. In questo caso R è una ϵ-regola completata. Assumiamo che lo stato accettante contenga anche una regola marcata T in cui un simbolo terminale segue immediatamente il punto. Poiché T è di tipo 1-2, un simbolo di variabile precede immediatamente il punto e quindi z termina con quel simbolo di variabile. Inoltre, dopo che P legge z esso è pronto a leggere un simbolo di input diverso da ϵ poiché un terminale segue il punto. Come nel Caso 1a, la ϵ-regola completata R deriva da una regola marcata S di tipo 1-2 in cui il punto precede immediatamente la seconda variabile. (Di nuovo, questo stato accettante non può essere lo stato iniziale di DK perché il punto non è all'inizio di T.) Quindi qualche simbolo $\hat{a} \in \Sigma_\epsilon$ precede immediatamente il punto in S e quindi z termina con $\hat{a}$. Ora $\hat{a} \in \Sigma$ oppure $\hat{a} = \epsilon$, ma poiché z termina con un simbolo di variabile, $\hat{a} \notin \Sigma$ quindi $\hat{a} = \epsilon$. Pertanto, dopo che P legge z ma prima che faccia la ϵ-input mossa per elaborare $\hat{a}$, è pronto a leggere un input ϵ. Ma abbiamo anche mostrato prima che in questo punto P è pronto a leggere un simbolo di input diverso da ϵ. Ma a un DPDA non è consentito fare sia una ϵ-input mossa che una mossa in cui legge un simbolo di input diverso da ϵ in un dato stato e con un dato contenuto della pila, quindi la situazione precedente è impossibile. In conclusione, questa situazione non può presentarsi.

Parsing e grammatiche LR(k)

I linguaggi context-free deterministici sono di considerevole importanza per le applicazioni. I loro algoritmi per l'appartenenza e il parsing sono basati sui DPDA e quindi sono efficienti, ed essi abbracciano un'ampia classe di CFL che include la maggior parte dei linguaggi di programmazione. Tuttavia, le DCFG sono a volte scomode per esprimere particolari DCFL. Il requisito che tutti gli handle siano obbligati è spesso un ostacolo nel progettare DCFG intuitive.

Fortunatamente, una classe più ampia di grammatiche, chiamate grammatiche LR(k), ci fornisce le migliori caratteristiche di entrambe le situazioni. Sono abbastanza vicine alle DCFG da permettere una conversione diretta nei DPDA. E sono anche abbastanza espressive per molte applicazioni.

Gli algoritmi per le grammatiche LR(k) introducono il *lookahead*. In una DCFG, tutti gli handle sono obbligati. Un handle dipende solo dai simboli in una stringa valida fino all'handle incluso, ma non dai simboli terminali che seguono l'handle. In una grammatica LR(k), un handle può anche dipendere da simboli che seguono l'handle, ma solo dai primi k di questi. L'acronimo LR(k) sta per: elaborazione dell'input da sinistra (Left) a destra, derivazioni a destra (Rightmost) (o equivalentemente, riduzioni a sinistra) e k simboli di lookahead.

Precisamente, sia h un handle di una stringa valida $v = xhy$. Diciamo che h è ***obbligato con lookahead k*** se h è l'unico handle di ogni stringa valida $xh\hat{y}$ dove $\hat{y} \in \Sigma^*$ e dove y e $\hat{y}$ coincidono sui loro primi k simboli. (Se una delle due stringhe è più corta di k, le stringhe devono essere uguali fino alla lunghezza della più corta.)

DEFINIZIONE **2.62**

Una ***grammatica* LR(k)** è una grammatica context-free tale che l'handle di ogni stringa valida è obbligato con lookahead k.

Quindi essere una DCFG è lo stesso che essere una grammatica LR(0). Possiamo mostrare che per ogni k è possibile convertire grammatiche LR(k) in DPDA. Abbiamo già mostrato che i DPDA sono equivalenti alle grammatiche LR(0). Quindi le grammatiche LR(k) sono computazionalmente equivalenti per ogni k e tutti descrivono esattamente i DCFL. L'esempio seguente mostra che le grammatiche LR(1) sono più convenienti delle DCFG per specificare alcuni linguaggi.

Per evitare notazioni complicate e dettagli tecnici, mostreremo come trasformare grammatiche LR(k) in DPDA solo nel caso particolare in cui $k = 1$. La conversione nel caso generale funziona sostanzialmente nello stesso modo.

Per iniziare, presenteremo una variante del DK-test, modificato per le grammatiche LR(1). Lo chiameremo il DK-test con lookahead 1 o semplicemente il DK_1-test. Come prima, costruiremo un NFA, chiamato qui K_1, e lo convertiremo in un DFA DK_1. Ciascuno degli stati di K_1 ha una regola marcata $T \to u.v$ e ora anche un simbolo terminale a, chiamato il ***simbolo di lookahead***, e sarà denotato con $\boxed{T \to u.v \quad a}$. Questo stato indica che K_1 ha appena letto la stringa u, che sarebbe una parte di un handle uv a condizione che v venga dopo u e a venga dopo v.

La costruzione formale funziona proprio come prima. Lo stato iniziale ha una ε-mossa a $\boxed{S_1 \to .u \quad a}$ per ogni regola che coinvolge la variabile iniziale S_1 e ogni $a \in \Sigma$. Le transizioni di spostamento portano da $\boxed{T \to u.xv \quad a}$ a $\boxed{T \to ux.v \quad a}$ sull'input x dove x è un simbolo di variabile o un simbolo terminale. Le ε-transizioni portano da $\boxed{T \to u.Cv \quad a}$ a $\boxed{C \to .r \quad b}$ per ogni regola $C \to r$, dove b è il primo simbolo di una qualsiasi stringa di terminali che può essere derivata da v. Se v deriva ε, aggiungiamo $b = a$. Gli stati accettanti sono tutti i $\boxed{\boxed{B \to u. \quad a}}$ per regole completate $B \to u.$ e $a \in \Sigma$.

Sia R_1 una regola completata con simbolo di lookahead a_1, e sia R_2 una regola marcata con simbolo di lookahead a_2. Diciamo che R_1 ed R_2 sono ***coerenti*** se

1. R_2 è completata e $a_1 = a_2$ oppure
2. R_2 non è completata e a_1 segue immediatamente il suo punto.

Siamo ora pronti a descrivere il ***DK_1-test.*** Costruiamo il DFA DK_1. Il test stabilisce che ogni stato accettante non deve contenere due qualsiasi regole marcate coerenti.

TEOREMA 2.63

G supera il DK_1-test se e solo se G è una grammatica LR(1).

IDEA. Il Corollario 2.51 si applica ancora a DK_1 poichè possiamo ignorare i simboli di lookahead.

ESEMPIO 2.64

Questo esempio mostra che la grammatica seguente supera il DK_1-test. Ricorda che nell'Esempio 2.53 abbiamo mostrato che questa grammatica non superava il DK-test. Quindi è un esempio di una grammatica che è LR(1) ma non è una DCFG.

$$S \to E\dashv$$
$$E \to E + T \mid T$$
$$T \to T \times \mathsf{a} \mid \mathsf{a}$$

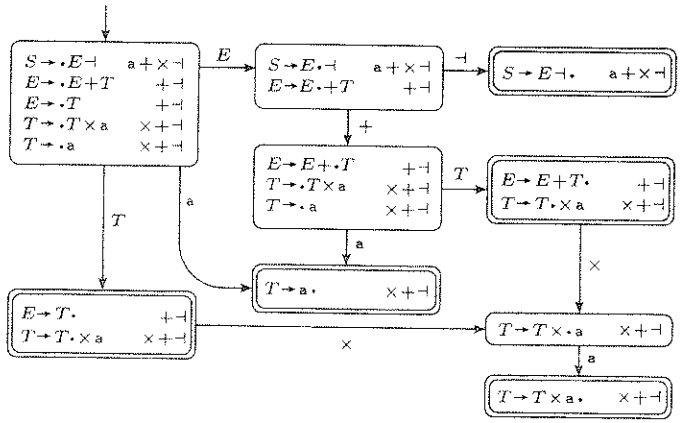


FIGURA 2.65
Esempio di DK_1-test superato

TEOREMA 2.66

Un linguaggio marcato è generato da una grammatica LR(1) se e solo se è un DCFL.

Abbiamo già mostrato che ogni DCFL è generato da una grammatica LR(0), poichè una grammatica LR(0) è lo stesso che una DCFG. Questo prova la direzione inversa del teorema. Quello che resta è il lemma seguente, che mostra come trasformare una grammatica LR(1) in un DPDA.

LEMMA 2.67

Per ogni grammatica LR(1) esiste un DPDA equivalente.

IDEA. Costruiamo P_1, una versione modificata del DPDA P presentato nel Lemma 2.67. P_1 legge il suo input e simula DK_1, usando la pila per mantenere traccia dello stato in cui sarebbe DK_1 se tutti i passi di riduzione fossero applicati all'input fino a questo punto. Inoltre, P_1 legge 1 simbolo successivo e memorizza questa informazione sul lookahead nella sua memoria costituita dall'insieme finito degli stati. Ogni volta che DK_1 raggiunge uno stato accettante, P_1 consulta il suo lookahead per vedere se compiere un passo di riduzione, e quale passo fare se appaiono diverse possibilità in questo stato. Solo una scelta può essere applicata perché la grammatica è LR(1).

ESERCIZI

2.1 Ricorda la CFG G_4 che abbiamo dato nell'Esempio 2.4. Per comodità, rinominiamo le sue variabili con singole lettere come segue.

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid \mathtt{a}$$

Fornire gli alberi sintattici e le derivazioni per ciascuna stringa.

a. a
b. a+a
c. a+a+a
d. ((a))

2.2 **a.** Usa i linguaggi $A = \{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \mid m, n \geq 0\}$ e $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m \mid m, n \geq 0\}$ con l'Esempio 2.36 per mostrare che la classe dei linguaggi context-free non è chiusa rispetto all'intersezione.

b. Usa la parte (a) e la legge di DeMorgan (Teorema 0.20) per mostrare che la classe dei linguaggi context-free non è chiusa rispetto al complemento.

A2.3 Rispondere a ciascun punto per la seguente grammatica context-free G.

$$R \to XRX \mid S$$
$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}T\mathtt{a}$$
$$T \to XTX \mid X \mid \varepsilon$$
$$X \to \mathtt{a} \mid \mathtt{b}$$

a. Quali sono le variabili di G?
b. Quali sono i terminali di G?
c. Qual è la variabile iniziale di G?
d. Fornire tre stringhe in $L(G)$.
e. Fornire tre stringhe *non* in $L(G)$.
f. Vero o Falso: $T \Rightarrow \mathtt{aba}$.
g. Vero o Falso: $T \stackrel{*}{\Rightarrow} \mathtt{aba}$.
h. Vero o Falso: $T \Rightarrow T$.
i. Vero o Falso: $T \stackrel{*}{\Rightarrow} T$.
j. Vero o Falso: $XXX \stackrel{*}{\Rightarrow} \mathtt{aba}$.
k. Vero o Falso: $X \stackrel{*}{\Rightarrow} \mathtt{aba}$.
l. Vero o Falso: $T \stackrel{*}{\Rightarrow} XX$.
m. Vero o Falso: $T \stackrel{*}{\Rightarrow} XXX$.
n. Vero o Falso: $S \stackrel{*}{\Rightarrow} \varepsilon$.
o. Dare una descrizione informale di $L(G)$.

2.4 Fornire grammatiche context-free che generino i linguaggi seguenti. Per ognuno di essi, l'alfabeto Σ è $\{0,1\}$.

Aa. $\{w \mid w$ contiene almeno tre simboli uguali a $1\}$
b. $\{w \mid w$ inizia e termina con lo stesso simbolo$\}$
c. $\{w \mid$ la lunghezza di w è dispari$\}$
Ad. $\{w \mid$ la lunghezza di w è dispari e il suo simbolo centrale è uno $0\}$
e. $\{w \mid w = w^{\mathcal{R}}$, cioè w è palindroma$\}$
f. L'insieme vuoto

2.5 Dare descrizioni informali e diagrammi di stato di automi a pila per i linguaggi nell'Esercizio 2.4.

2.6 Fornire grammatiche context-free che generino i linguaggi seguenti.

Aa. L'insieme delle stringhe sull'alfabeto {a,b} con un numero maggiore di a che di b
b. Il complemento del linguaggio $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$
Ac. $\{w\#x \mid w^{\mathcal{R}}$ è una sottostringa di x per $w, x \in \{0,1\}^*\}$
d. $\{x_1\#x_2\#\cdots\#x_k \mid k \geq 1$, ciascun $x_i \in \{\mathtt{a},\mathtt{b}\}^*$, e per qualche i e j, $x_i = x_j^{\mathcal{R}}\}$

A2.7 Dare descrizioni informali di PDA per i linguaggi nell'Esercizio 2.6.

A2.8 Mostrare che la stringa `the girl touches the boy with the flower` ha due diverse derivazioni sinistre nella grammatica G_2 a pagina 106. Descrivere in italiano i due diversi significati di questa frase.

2.9 Fornire una grammatica context-free che generi il linguaggio.

$$A = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ o } j = k \text{ dove } i, j, k \geq 0\}.$$

La grammatica è ambigua? Perché lo è o perché non lo è?

2.10 Dare una descrizione informale di un automa a pila che riconosce il linguaggio A nell'Esercizio 2.9.

2.11 Trasformare la CFG G_4 data nell'Esercizio 2.1 in un PDA equivalente, usando la procedura data nel Teorema 2.20.

2.12 Trasformare la CFG G data nell'Esercizio 2.3 in un PDA equivalente, usando la procedura data nel Teorema 2.20.

2.13 Sia $G = (V, \Sigma, R, S)$ la grammatica seguente. $V = \{S, T, U\}$; $\Sigma = \{0, \#\}$; ed R è l'insieme delle regole:

$$S \to TT \mid U$$
$$T \to 0T \mid T0 \mid \#$$
$$U \to 0U00 \mid \#$$

a. Descrivere informalmente $L(G)$.
b. Provare che $L(G)$ non è regolare.

2.14 Convertire la CFG seguente in una CFG equivalente in forma normale di Chomsky, usando la procedura data nel Teorema 2.9.

$$A \to BAB \mid B \mid \varepsilon$$
$$B \to 00 \mid \varepsilon$$

2.15 Dare un controesempio per mostrare che la costruzione seguente non dimostra che la classe dei linguaggi context-free è chiusa rispetto allo star. Sia A un CFL generato dalla CFG $G = (V, \Sigma, R, S)$. Sia aggiunga la nuova regola $S \to SS$ e si chiami G' la grammatica risultante. Questa grammatica dovrebbe generare A^*.

2.16 Mostrare che la classe dei linguaggi context-free è chiusa rispetto alle operazioni regolari, unione, concatenazione e star.

2.17 Usare i risultati dell'Esercizio 2.16 per dare un'altra prova che ogni linguaggio regolare è context-free, mostrando come convertire direttamente un'espressione regolare in una grammatica context-free equivalente.

PROBLEMI

2.18 Considera la seguente CFG G:

$$S \to SS \mid T$$
$$T \to \mathtt{a}T\mathtt{b} \mid \mathtt{ab}$$

Descrivi $L(G)$ e mostra che G è ambigua. Fornisci una grammatica non ambigua H tale che $L(H) = L(G)$ e schematizza una prova che H è non ambigua.

***2.19** Abbiamo definito la chiusura rotazionale di un linguaggio A come $RC(A) = \{yx \mid xy \in A\}$. Mostrare che la classe dei CFL è chiusa rispetto alla chiusura rotazionale.

***2.20** Abbiamo definito il CUT di un linguaggio A come $CUT(A) = \{yxz \mid xyz \in A\}$. Mostrare che la classe dei CFL non è chiusa rispetto a CUT.

2.21 Mostrare che ogni DCFG è una CFG non ambigua.

[A]***2.22** Mostrare che ogni DCFG genera un linguaggio prefisso.

***2.23** Mostrare che la classe dei DCFL non è chiusa rispetto alle operazioni seguenti:

a. Unione
b. Intersezione
c. Concatenazione
d. Star
e. Inversione

2.24 Sia G la grammatica seguente:

$$S \to T\dashv$$
$$T \to T\mathtt{a}T\mathtt{b} \mid T\mathtt{b}T\mathtt{a} \mid \varepsilon$$

a. Mostrare che $L(G) = \{w\dashv \mid w$ contiene un ugual numero di a e b$\}$. Usare una prova per induzione sulla lunghezza di w.
b. Usare il DK-test per mostrare che G è una DCFG.
c. Descrivere un DPDA che riconosce $L(G)$.

2.25 Sia G_1 la grammatica seguente che abbiamo introdotto nell'Esempio 2.45. Usare il DK-test per mostrare che G_1 non è una DCFG.

$$R \to S \mid T$$
$$S \to \mathtt{a}S\mathtt{b} \mid \mathtt{ab}$$
$$T \to \mathtt{a}T\mathtt{bb} \mid \mathtt{abb}$$

***2.26** Sia $A = L(G_1)$ dove G_1 è definita nel Problema 2.25. Mostrare che A non è un DCFL.
(Suggerimento: Si assuma che A sia un DCFL e si consideri il suo DPDA P. Si modifichi P in modo che il suo alfabeto di input sia $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$. Quando entra in uno stato accettante per la prima volta, agisce come se, da quel momento in poi, nell'input le c fossero b. Quale linguaggio accetterebbe P modificato?)

***2.27** Sia $B = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i,j,k \geq 0$ e $i = j$ o $i = k\}$. Provare che B non è un DCFL.

***2.28** Sia $C = \{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$. Provare che C non è un DCFL. (Suggerimento: Si assuma che quando un DPDA P è inizializzato nello stato q con il simbolo x sulla cima della sua pila, P non esegua pop sulla pila al di sotto di x, indipendentemente dalla stringa di input che P legge da quel punto in poi. In questo caso, il contenuto della pila di P in quel momento non può avere un effetto sul suo comportamento successivo, quindi il comportamento seguente di P può dipendere solo da q e x.)

***2.29** Se non permettiamo le ε-regole nelle CFG, possiamo semplificare il DK-test. Nel test semplificato, dobbiamo solo verificare che ogni stato accettante del DK ha una singola regola. Provare che una CFG senza ε-regole supera il DK-test semplificato se e solo se è una DCFG.

[A]**2.30** **a.** Sia C un linguaggio context-free e sia R un linguaggio regolare. Provare che il linguaggio $C \cap R$ è context-free.
b. Sia $A = \{w \mid w \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ e w contiene un ugual numero di a b e c$\}$. Usare la parte (a) per mostrare che A non è un CFL.

***2.31** Sia G la grammatica CFG seguente.

$$S \to \mathtt{a}S\mathtt{b} \mid \mathtt{b}Y \mid Y\mathtt{a}$$
$$Y \to \mathtt{b}Y \mid \mathtt{a}Y \mid \varepsilon$$

Dare una semplice e informale descrizione di $L(G)$. Usare questa descrizione per fornire una CFG per $\overline{L(G)}$, il complemento di $L(G)$.

2.32 Sia $A/B = \{w \mid wx \in A$ per qualche $x \in B\}$. Mostrare che se A è context-free e B è regolare, allora A/B è context-free.

***2.33** Sia $\Sigma = \{\mathtt{a},\mathtt{b}\}$. Fornisci una CFG che generi il linguaggio delle stringhe con un numero di a doppio rispetto al numero di b. Prova che la tua grammatica è corretta.

***2.34** Sia $C = \{x\#y \mid x,y \in \{0,1\}^*$ e $x \neq y\}$. Mostrare che C è un linguaggio context-free.

***2.35** Sia $D = \{xy \mid x,y \in \{0,1\}^*$ e $|x| = |y|$ ma $x \neq y\}$. Mostrare che D è un linguaggio context-free.

***2.36** Sia $E = \{\mathtt{a}^i\mathtt{b}^j \mid i \neq j$ e $2i \neq j\}$. Mostrare che E è un linguaggio context-free.

2.37 Per ogni linguaggio A, sia $SUFFIX(A) = \{v \mid uv \in A$ per qualche stringa $u\}$. Mostrare che la classe dei linguaggi context-free è chiusa rispetto all'operazione $SUFFIX$.

2.38 Mostrare che se G è una CFG in forma normale di Chomsky, allora per ogni stringa $w \in L(G)$ di lunghezza $n \geq 1$, sono richiesti esattamente $2n - 1$ passi per ogni derivazione di w.

***2.39** Sia $G = (V, \Sigma, R, \langle\text{STMT}\rangle)$ la grammatica seguente.

$$\langle\text{STMT}\rangle \to \langle\text{ASSIGN}\rangle \mid \langle\text{IF-THEN}\rangle \mid \langle\text{IF-THEN-ELSE}\rangle$$
$$\langle\text{IF-THEN}\rangle \to \texttt{if condition then } \langle\text{STMT}\rangle$$
$$\langle\text{IF-THEN-ELSE}\rangle \to \texttt{if condition then } \langle\text{STMT}\rangle \texttt{ else } \langle\text{STMT}\rangle$$
$$\langle\text{ASSIGN}\rangle \to \texttt{a:=1}$$

$$\Sigma = \{\texttt{if}, \texttt{condition}, \texttt{then}, \texttt{else}, \texttt{a:=1}\}$$
$$V = \{\langle\text{STMT}\rangle, \langle\text{IF-THEN}\rangle, \langle\text{IF-THEN-ELSE}\rangle, \langle\text{ASSIGN}\rangle\}$$

G è una grammatica che appare naturale per un frammento di un linguaggio di programmazione, ma G è ambigua.

a. Mostrare che G è ambigua.
b. Dare una nuova grammatica non ambigua per lo stesso linguaggio.

*2.40 Fornire CFG non ambigue per i linguaggi seguenti.

a. $\{w|$ in ogni prefisso di w il numero di a è maggiore o uguale al numero di b$\}$

b. $\{w|$ il numero di a e il numero di b in w sono uguali$\}$

c. $\{w|$ il numero di a è maggiore o uguale al numero di b in $w\}$

*2.41 Mostrare che il linguaggio A nell'Esercizio 2.9 è inerentemente ambiguo.

2.42 Usare il pumping lemma per mostrare che i seguenti linguaggi non sono context-free.

a. $\{0^n1^n0^n1^n \mid n \geq 0\}$

A**b.** $\{0^n\#0^{2n}\#0^{3n} \mid n \geq 0\}$

A**c.** $\{w\#t \mid w$ è una sottostringa di t, dove $w, t \in \{\mathtt{a},\mathtt{b}\}^*\}$

d. $\{t_1\#t_2\#\cdots\#t_k \mid k \geq 2$, ciascun $t_i \in \{\mathtt{a},\mathtt{b}\}^*$ e $t_i = t_j$ per qualche $i \neq j\}$

2.43 Sia B il linguaggio di tutte le palindrome su $\{0,1\}$ contenenti lo stesso numero di simboli uguali a 0 e uguali a 1. Mostrare che B non è context-free.

2.44 Sia $\Sigma = \{1,2,3,4\}$ e $C = \{w \in \Sigma^* \mid$ in w, il numero di simboli uguali a 1 è uguale al numero dei simboli uguali a 2 e il numero dei simboli uguali a 3 è uguale al numero dei simboli uguali a 4$\}$. Mostrare che C non è context-free.

*2.45 Mostrare che $F = \{\mathtt{a}^i\mathtt{b}^j \mid i = kj$ per qualche intero positivo $k\}$ non è context-free.

2.46 Considera il linguaggio $B = L(G)$, dove G è la grammatica data nell'Esercizio 2.13. Il pumping lemma per i linguaggi context-free, Teorema 2.34, afferma l'esistenza di una lunghezza p del pumping per B. Qual è il valore minimo di p che funziona nel pumping lemma? Giustifica la tua risposta.

2.47 Sia G una CFG in forma normale di Chomsky che contiene b variabili. Mostrare che se G genera una stringa con una derivazione che ha almeno 2^b passi, $L(G)$ è infinito.

2.48 Dai un esempio di un linguaggio che non è context-free ma che si comporta come un CFL rispetto al pumping lemma. Prova che il tuo esempio funziona. (Cfr. l'analogo esempio per i linguaggi regolari nel Problema 1.49.)

*2.49 Provare la seguente forma più forte del pumping lemma, nella quale *entrambe* le parti v e y devono essere non vuote quando la stringa s è suddivisa.

Se A è un linguaggio context-free, allora c'è un numero k tale che, se s è una stringa in A di lunghezza almeno k, allora s può essere divisa in cinque parti, $s = uvxyz$, che soddisfano le condizioni:

a. per ogni $i \geq 0$, $uv^ixy^iz \in A$,

b. $v \neq \varepsilon$ e $y \neq \varepsilon$ e

c. $|vxy| \leq k$.

A**2.50** Fai riferimento al Problema 1.31 per la definizione dell'operazione di shuffle perfetto. Mostra che la classe dei linguaggi context-free non è chiusa rispetto allo shuffle perfetto.

2.51 Fai riferimento al Problema 1.32 per la definizione dell'operazione di shuffle. Mostra che la classe dei linguaggi context-free non è chiusa rispetto allo shuffle.

*2.52 Diciamo che un linguaggio è ***chiuso rispetto ai prefissi*** se anche tutti i prefissi di ogni stringa nel linguaggio sono nel linguaggio. Sia C un linguaggio infinito, chiuso rispetto ai prefissi e context-free. Mostrare che C contiene un sottoinsieme infinito regolare.

*2.53 Leggi le definizioni di $NOPREFIX(A)$ e $NOEXTEND(A)$ nel Problema 1.45.

a. Mostra che la classe dei CFL non è chiusa rispetto a $NOPREFIX$.

b. Mostra che la classe dei CFL non è chiusa rispetto a $NOEXTEND$.

*2.54 Sia $Y = \{w \mid w = t_1\#t_2\#\cdots\#t_k$ con $k \geq 0$, ogni $t_i \in 1^*$ e $t_i \neq t_j$ se $i \neq j\}$. In questo caso $\Sigma = \{1,\#\}$. Provare che Y non è context-free.

2.55 Per stringhe w and t, scriviamo $w \triangleq t$ se i simboli di w sono una permutazione dei simboli di t. In altre parole, $w \triangleq t$ se t e w hanno gli stessi simboli con ugual numero di occorrenze, ma eventualmente in un diverso ordine.

Per una stringa w, definiamo $SCRAMBLE(w) = \{t \mid t \triangleq w\}$. Per un linguaggio A, sia $SCRAMBLE(A) = \{t \mid t \in SCRAMBLE(w)$ per qualche $w \in A\}$.

a. Mostrare che se $\Sigma = \{0,1\}$, allora lo $SCRAMBLE$ di un linguaggio regolare è context-free.

b. Cosa accade in (a) se Σ contiene tre o più simboli? Prova la tua risposta.

2.56 Se A e B sono linguaggi, definiamo $A \diamond B = \{xy \mid x \in A$ e $y \in B$ e $|x| = |y|\}$. Mostrare che se A e B sono linguaggi regolari, allora $A \diamond B$ è un CFL.

*2.57 Sia $A = \{wtw^{\mathcal{R}} \mid w, t \in \{0,1\}^*$ e $|w| = |t|\}$. Provare che A non è un CFL.

2.58 Sia $\Sigma = \{0,1\}$ e sia B la collezione delle stringhe che contengono almeno un 1 nella loro seconda metà. In altre parole, $B = \{uv \mid u \in \Sigma^*, v \in \Sigma^*1\Sigma^*$ e $|u| \geq |v|\}$.

a. Fornire un PDA che riconosce B.

b. Fornire una CFG che genera B.

2.59 Sia $\Sigma = \{0,1\}$. Sia C_1 il linguaggio di tutte le stringhe che contengono un 1 nella loro parte centrale. Sia C_2 il linguaggio di tutte le stringhe che contengono due 1 nella loro parte centrale. Quindi $C_1 = \{xyz \mid x, z \in \Sigma^*$ ed $y \in \Sigma^*1\Sigma^*$, dove $|x| = |z| \geq |y|\}$ e $C_2 = \{xyz \mid x, z \in \Sigma^*$ ed $y \in \Sigma^*1\Sigma^*1\Sigma^*$, dove $|x| = |z| \geq |y|\}$.

a. Mostrare che C_1 è un CFL.

b. Mostrare che C_2 non è un CFL.

SOLUZIONI SELEZIONATE

2.3 **(a)** R, X, S, T; **(b)** a, b; **(c)** R; **(d)** Tre stringhe in $L(G)$ sono ab, ba e aab; **(e)** Tre stringhe non in $L(G)$ sono a, b ed ε; **(f)** Falso; **(g)** Vero; **(h)** Falso; **(i)** Vero; **(j)** Vero; **(k)** Falso; **(l)** Vero; **(m)** Vero; **(n)** Falso; **(o)** $L(G)$ consiste di tutte le stringhe su a e b che non sono palindrome.

2.4 **(a)** $S \to R1R1R1R$
$R \to 0R \mid 1R \mid \varepsilon$

(d) $S \to 0 \mid 0S0 \mid 0S1 \mid 1S0 \mid 1S1$

2.6 **(a)** $S \to TaT$
$T \to TT \mid aTb \mid bTa \mid a \mid \varepsilon$

T genera tutte le stringhe con almeno tante a quante b ed S impone una a in più.

(c) $S \to TX$
$T \to 0T0 \mid 1T1 \mid \#X$
$X \to 0X \mid 1X \mid \varepsilon$

2.7 (a) Il PDA usa la sua pila per contare il numero di a meno il numero di b. Entra in uno stato accettante ogni volta che questo conteggio è positivo. Più in dettaglio, esso opera come segue. Il PDA esamina tutto l'input. Se vede una b e il simbolo sulla cima della pila è una a, rimuove il simbolo dalla pila. Analogamente, se vede una a e il simbolo sulla cima della pila è una b, rimuove il simbolo dalla pila. In tutti gli altri casi, inserisce il simbolo di input sulla pila. Dopo che il PDA ha terminato l'input, se una a è sulla cima della pila, esso accetta. Altrimenti rifiuta.

(c) Il PDA esamina tutta la stringa di input e inserisce nella pila ogni simbolo che legge fino a quando legge un #. Se un # non è presente, rifiuta. Poi, il PDA salta parte dell'input, decidendo non deterministicamente quando cessare di saltare. A quel punto, confronta i successivi simboli input con i simboli che rimuove dalla pila. Se non corrispondono o se l'input termina ma la pila non è vuota, questo ramo della computazione rifiuta. Se la pila si svuota, la macchina legge il resto dell'input e accetta.

2.8 Ecco una derivazione:
⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩ →
⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩ →
⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩ →
The ⟨NOUN⟩⟨VERB-PHRASE⟩ →
The girl ⟨VERB-PHRASE⟩ →
The girl ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩ →
The girl ⟨VERB⟩⟨NOUN-PHRASE⟩⟨PREP-PHRASE⟩ →
The girl touches ⟨NOUN-PHRASE⟩⟨PREP-PHRASE⟩ →
The girl touches ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩ →
The girl touches ⟨ARTICLE⟩⟨NOUN⟩⟨PREP-PHRASE⟩ →
The girl touches the ⟨NOUN⟩⟨PREP-PHRASE⟩ →
The girl touches the boy ⟨PREP-PHRASE⟩ →
The girl touches the boy ⟨PREP⟩⟨CMPLX-NOUN⟩ →
The girl touches the boy with ⟨CMPLX-NOUN⟩ →
The girl touches the boy with ⟨ARTICLE⟩⟨NOUN⟩ →
The girl touches the boy with the ⟨NOUN⟩ →
The girl touches the boy with the flower

Ecco un'altra derivazione a sinistra:
⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩ →
⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩ →
⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩ →
The ⟨NOUN⟩⟨VERB-PHRASE⟩ →
The girl ⟨VERB-PHRASE⟩ →
The girl ⟨CMPLX-VERB⟩ →
The girl ⟨VERB⟩⟨NOUN-PHRASE⟩ →
The girl touches ⟨NOUN-PHRASE⟩ →
The girl touches ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩ →
The girl touches ⟨ARTICLE⟩⟨NOUN⟩⟨PREP-PHRASE⟩ →
The girl touches the ⟨NOUN⟩⟨PREP-PHRASE⟩ →
The girl touches the boy ⟨PREP-PHRASE⟩ →
The girl touches the boy ⟨PREP⟩⟨CMPLX-NOUN⟩ →
The girl touches the boy with ⟨CMPLX-NOUN⟩ →
The girl touches the boy with ⟨ARTICLE⟩⟨NOUN⟩ →
The girl touches the boy with the ⟨NOUN⟩ →
The girl touches the boy with the flower

Ciascuna di queste derivazioni corrisponde a un diverso significato. Nella prima derivazione, la frase significa che la ragazza usa il fiore per toccare il ragazzo. Nella seconda derivazione, il ragazzo sta tenendo il fiore quando la ragazza lo tocca.

2.22 Usiamo una prova per assurdo. Assumiamo che w e wz siano due stringhe diverse in $L(G)$, dove G è una DCFG. Entrambe sono stringhe valide quindi entrambe hanno handle, e questi handle devono coincidere poiché possiamo scrivere $w = xhy$ e $wz = xhyz = xh\hat{y}$ dove h è l'handle di w. Quindi, i primi passi di riduzione di w e wz producono le stringhe valide u e uz, rispettivamente. Possiamo continuare questo processo fino a quando otteniamo S_1 ed $S_1 z$ dove S_1 è la variabile iniziale. Però S_1 non appare sul lato destro di alcuna regola quindi non possiamo ridurre $S_1 z$. Cidà una contraddizione.

2.30 (a) Sia C un linguaggio context-free e sia R un linguaggio regolare. Sia P il PDA che riconosce C e sia D il DFA che riconosce R. Se Q è l'insieme degli stati di P e Q' è l'insieme degli stati di D, costruiamo un PDA P' che riconosce $C \cap R$ con l'insieme degli stati $Q \times Q'$. P' farà quello che fa P e mantiene anche traccia degli stati di D. Accetta una stringa w se e solo se si ferma in uno stato $q \in F_P \times F_D$, dove F_P è l'insieme degli stati accettanti di P ed F_D è l'insieme degli stati accettanti di D. Poiché $C \cap R$ è riconosciuto da P', esso è context-free.

(b) Sia R il linguaggio regolare $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$. Se A fosse un CFL allora $A \cap R$ sarebbe un CFL per la parte (a). Però $A \cap R = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ e l'Esempio 2.36 prova che $A \cap R$ non è context-free. Quindi A non è un CFL.

2.42 (b) Sia $B = \{0^n\#0^{2n}\#0^{3n} \mid n \geq 0\}$. Sia p la lunghezza del pumping data dal pumping lemma. Sia $s = 0^p\#0^{2p}\#0^{3p}$. Mostriamo che $s = uvxyz$ non può essere iterata.

Né v né y possono contenere #, altrimenti uv^2xy^2z conterrebbe più di due #. Quindi, se dividiamo s in tre segmenti mediante i #: $0^p, 0^{2p}$e 0^{3p}, almeno uno dei segmenti non è contenuto in v o y. Pertanto uv^2xy^2z non è in B poichè non rispetta il rapporto $1:2:3$ delle lunghezze dei segmenti.

(c) Sia $C = \{w\#t \mid w$ è una sottostringa di t, dove $w, t \in \{\mathtt{a},\mathtt{b}\}^*\}$. Sia p la lunghezza del pumping data dal pumping lemma. Sia $s = \mathtt{a}^p\mathtt{b}^p\#\mathtt{a}^p\mathtt{b}^p$. Mostriamo che la stringa $s = uvxyz$ non può essere iterata.

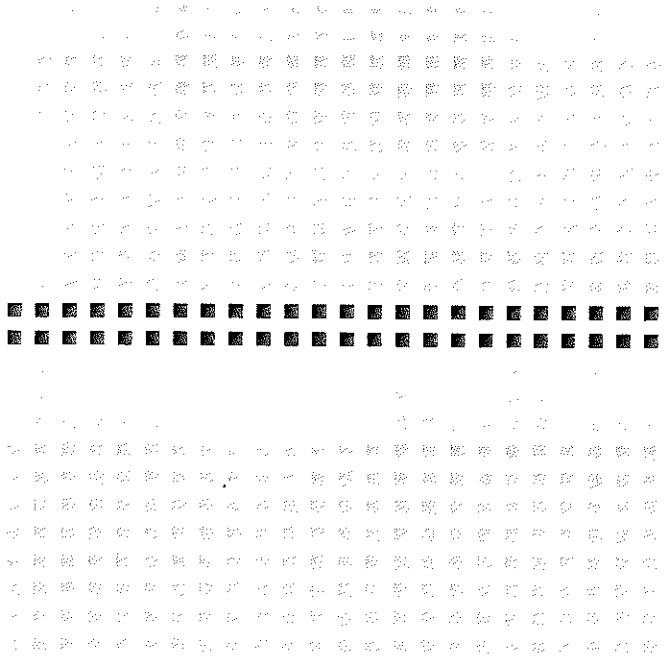
Né v né y possono contenere #, altrimenti uv^0xy^0z non conterrebbe # e allora non sarebbe in C. Se entrambe v e y sono alla sinistra di #, la stringa uv^2xy^2z non può essere in C perché è più lunga dal lato sinistro di #. Analogamente, se entrambe le stringhe sono alla destra di #, la stringa uv^0xy^0z non può essere in C perché è di nuovo più lunga dal lato sinistro di #. Se una tra v e y è vuota (non possono essere entrambe vuote), trattale come se entrambe fossero nello stesso lato di # come sopra.

Il solo caso che resta è quando entrambe v ed y non sono vuote e sono a cavallo del #. Ma allora v consiste di b ed y consiste di a a causa della terza condizione $|vxy| \leq p$ del pumping lemma. Quindi, uv^2xy^2z contiene più b sul lato sinistro di #, allora non può essere un elemento di C.

2.50 Sia A il linguaggio $\{0^k1^k \mid k \geq 0\}$ e sia B il linguaggio $\{\mathtt{a}^k\mathtt{b}^{3k} \mid k \geq 0\}$. Lo shuffle perfetto di A e B è il linguaggio $C = \{(0\mathtt{a})^k(0\mathtt{b})^k(1\mathtt{b})^{2k} \mid k \geq 0\}$. È facile vedere che i linguaggi A e B sono CFL ma C non è un CFL, come vedremo. Se C fosse

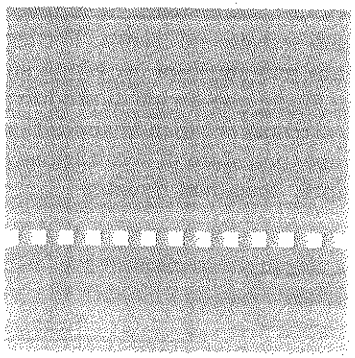
un CFL, sia p la lunghezza del pumping data dal pumping lemma e sia s la stringa $(\mathtt{0a})^p(\mathtt{0b})^p(\mathtt{1b})^{2p}$. Poiché s è più lunga di p ed $s \in C$, possiamo dividere $s = uvxyz$ in modo da soddisfare le tre condizioni del pumping lemma. Le stringhe in C hanno esattamente un quarto di simboli uguali a 1 e un ottavo di simboli uguali ad a. Affinché uv^2xy^2z abbia questa proprietà, la stringa vxy deve contenere sia simboli uguali a 1 che ad a. Ciò è impossibile perché i simboli uguali a 1 e le a sono separati da $2p$ simboli in s, ma la terza condizione afferma che $|vxy| \leq p$. Quindi C non è context-free.

PARTE SECONDA



TEORIA DELLA COMPUTABILITÀ

3



LA TESI DI CHURCH–TURING

Finora nello sviluppare la teoria della computazione abbiamo presentato vari modelli di dispositivi di calcolo. Gli automi finiti sono un buon modello per quei dispositivi che hanno una ridotta quantità di memoria. Gli automi a pila costituiscono un buon modello per quei dispositivi che hanno una memoria illimitata, ma utilizzabile solo nella modalità last in, first out di una pila. Abbiamo dimostrato che alcuni calcoli, pur se molto semplici, vanno oltre le capacità di questi modelli. Quindi essi sono troppo limitati per essere utilizzati come modelli di computer universale.

3.1 MACCHINE DI TURING

Introduciamo ora un modello molto più potente, proposto inizialmente da Alan Turing nel 1936, chiamato ***macchina di Turing***. Simile ad un automa finito, ma con una memoria illimitata e senza restrizioni, una macchina di Turing è un modello molto più preciso di un computer. Una macchina di Turing può fare tutto ciò che può fare un computer reale. Tuttavia, anche una macchina di Turing non è in grado di risolvere alcuni problemi. In pratica, questi problemi vanno oltre i limiti teorici della computazione. Una macchina di Turing utilizza un nastro infinito come propria memoria illimitata. Ha una testina che è in grado di leggere e scrivere simboli ed è

libera di muoversi lungo il nastro. Inizialmente il nastro contiene solo la stringa di input e tutto il resto è vuoto. Se la macchina deve memorizzare un'informazione, la può scrivere sul nastro. Per leggere l'informazione che ha scritto, la macchina può spostare la testina su di essa. La macchina continua a computare finché non decide di produrre un output. Gli output *accetta* e *rifiuta* sono ottenuti occupando appositi stati di accettazione e di rifiuto. Se non raggiunge uno stato di accettazione o di rifiuto, la macchina andrà avanti per sempre, senza mai fermarsi.

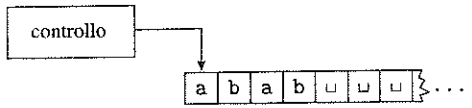


FIGURA 3.1
Schema di una macchina di Turing

L'elenco seguente riassume le differenze tra automi finiti e macchine di Turing.

1. Una macchina di Turing può sia scrivere che leggere sul nastro.
2. La testina di lettura-scrittura può muoversi sia verso sinistra che verso destra.
3. Il nastro è infinito.
4. Gli stati speciali di accettazione e rifiuto hanno effetto immediato.

Introduciamo una macchina di Turing M_1 per testare l'appartenenza al linguaggio $B = \{w\#w | \, w \in \{0,1\}^*\}$. Vogliamo che M_1 accetti se l'input è un elemento di B e che rifiuti altrimenti. Per comprendere meglio M_1, mettetevi al suo posto immaginando di essere fisicamente davanti ad un input lungo milioni di caratteri. Il vostro obiettivo è quello di determinare se l'input è un elemento di B, cioè se l'input comprende due stringhe identiche separate da un simbolo #. L'input è troppo lungo per voi per ricordarlo tutto, ma vi è permesso di muovervi avanti e indietro lungo l'input e porre dei contrassegni. La strategia ovvia è quella di muovervi a zig-zag in maniera simmetrica attorno al simbolo # e stabilire se gli elementi nelle posizioni di ugual indice su i due lati corrispondono. Contrassegnate poi il nastro per tenere traccia degli elementi che corrispondono. Progettiamo M_1 in modo da lavorare in questo modo. Esegue più passaggi sulla stringa in input con la testina di lettura-scrittura. Ad ogni passaggio verifica la corrispondenza di un carattere ad ogni lato del simbolo #. Per tenere traccia dei simboli già controllati, M_1 barra ogni simbolo già esaminato. Se ha barrato tutti i simboli, significa che tutti i simboli corrispondevano ed M_1 va in uno stato

di accettazione. Se si accorge di una mancata corrispondenza, M_1 entra in uno stato di rifiuto. In pratica, l'algoritmo di M_1 funziona come segue.

M_1 = "Su input w:

1. Si muove a zig–zag lungo il nastro, raggiungendo posizioni corrispondenti su i due lati del simbolo #, per controllare se queste posizioni contengono lo stesso simbolo. In caso negativo, o nel caso in cui non si trovi il simbolo #, *rifiuta*. Barra gli elementi già controllati.
2. Quando tutti gli elementi a sinistra del simbolo # sono stati barrati, verifica la presenza di eventuali simboli rimanenti a destra di #. Se rimane qualche elemento, allora *rifiuta*, altrimenti *accetta*."

La figura seguente contiene varie istantanee non consecutive del nastro di M_1 una volta avviata sull'input 011000#011000.

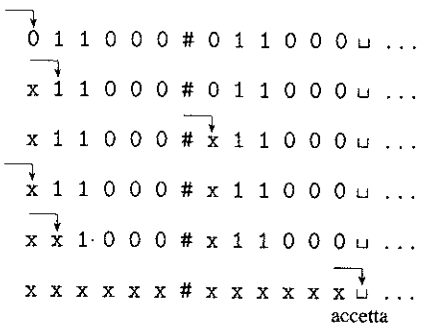


FIGURA 3.2
Istantanee della macchina di Turing M_1 mentre computa sull'input 011000#011000

Questa descrizione della macchina di Turing M_1 ne illustra il funzionamento, ma non ne mostra tutti i dettagli. Possiamo descrivere macchine di Turing in modo più dettagliato dandone descrizioni formali analoghe a quelle introdotte per automi finiti ed automi a pila. Le descrizioni formali specificano ogni parte della definizione formale del modello di macchina di Turing che presenteremo a breve. In pratica, non forniamo quasi mai descrizioni formali delle macchine di Turing, in quanto tendono ad essere eccessivamente lunghe.

Definizione formale di macchina di Turing

Il cuore della definizione di una macchina di Turing è la funzione di transizione δ perché essa ci dice come la macchina effettua un passo. Per una macchina di Turing, δ prende la forma $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$. Cioè,

quando la macchina occupa un certo stato q e la testina punta alla casella del nastro contenente un simbolo a, e se $\delta(q,a) = (r,b,\mathrm{L})$, la macchina scrive il simbolo b al posto di a, e passa nello stato r. La terza componente è L oppure R ed indica se la testina si sposta verso sinistra o verso destra dopo la scrittura. In questo caso la L indica una mossa a sinistra.

DEFINIZIONE **3.3**

Una ***macchina di Turing*** è una 7-tupla, $(Q,\Sigma,\Gamma,\delta,q_0,q_{\text{accept}},q_{\text{reject}})$, dove Q, Σ, Γ sono tutti insiemi finiti e

1. Q è l'insieme degli stati,
2. Σ è l'alfabeto di input non contenente il simbolo ***blank*** $\sqcup$,
3. Γ è l'alfabeto del nastro, con $\sqcup \in \Gamma$ e $\Sigma \subseteq \Gamma$,
4. $\delta\colon Q\times\Gamma \longrightarrow Q\times\Gamma\times\{\mathrm{L},\mathrm{R}\}$ è la funzione di transizione,
5. $q_0 \in Q$ è lo stato iniziale,
6. $q_{\text{accept}} \in Q$ è lo stato di accettazione e
7. $q_{\text{reject}} \in Q$ è lo stato di rifiuto, con $q_{\text{reject}} \neq q_{\text{accept}}$.

Una macchina di Turing $M = (Q,\Sigma,\Gamma,\delta,q_0,q_{\text{accept}},q_{\text{reject}})$ computa nel seguente modo. Inizialmente M riceve il suo input $w = w_1w_2\ldots w_n \in \Sigma^*$ sulle n celle più a sinistra del nastro, mentre il resto del nastro è vuoto (cioè, contiene tutti simboli blank). La testina parte dalla posizione più a sinistra del nastro. Si noti che Σ non contiene il simbolo blank, in tal modo il primo sombolo blank che compare sul nastro segna la fine dell'input. Quando M inizia la computazione, questa procede secondo le regole descritte dalla funzione di transizione. Se M tenta di spostare la testina a sinistra quando si trova all'estremità sinistra del nastro, allora la testina rimane nello stesso posto per quella mossa, anche se la funzione di transizione indica L. La computazione prosegue fin quando la macchina raggiunge lo stato di accettazione oppure di rifiuto, a tal punto si ferma. Se nessuno dei due stati viene raggiunto, M va avanti per sempre. Durante la computazione di una macchina di Turing, si verificano cambiamenti dello stato corrente, del contenuto corrente del nastro, e della posizione corrente della testina. Un'impostazione di questi tre elementi è chiamata una ***configurazione*** della macchina di Turing. Le configurazioni sono spesso rappresentate in modo speciale. Per uno stato q e due stringhe u e v sull'alfabeto Γ del nastro, scriviamo $u\,q\,v$ per indicare la configurazione, dove lo stato corrente è q, il contenuto corrente del nastro è uv e la posizione attuale della testina è il primo simbolo di v. Dopo l'ultimo simbolo di v, il nastro contiene solo simboli blank. Per esempio, $1011q_701111$ rappresenta la configurazione in

cui il nastro contiene 101101111, lo stato corrente è q_7 e la testina è posizionata sul secondo 0. La figura 3.4 mostra una macchina di Turing con tale configurazione.

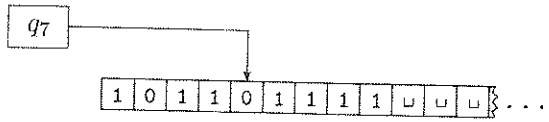


FIGURA 3.4
Una macchina di Turing con cofigurazione $1011q_701111$

Nel seguito formalizziamo la nostra comprensione intuitiva del modo in cui una macchina di Turing computa. Si dice che la configurazione C_1 ***produce*** la configurazione C_2 se la macchina di Turing può passare da C_1 a C_2 in un unico passo. Definiamo formalmente questa nozione nel seguente modo. Supponiamo di avere a, b e c in Γ, così come u e v in Γ^* e gli stati q_i e q_j. In tal caso $ua\,q_i\,bv$ e $u\,q_j\,acv$ sono due configurazioni. Diciamo che

$$ua\,q_i\,bv \quad \text{produce} \quad u\,q_j\,acv$$

se nella funzione di transizione $\delta(q_i,b) = (q_j,c,\mathrm{L})$. Questo nel caso in cui la macchina di Turing effettua uno spostamento verso sinistra. Per uno spostamento a destra, diciamo che

$$ua\,q_i\,bv \quad \text{produce} \quad uac\,q_j\,v$$

se $\delta(q_i,b) = (q_j,c,\mathrm{R})$. Casi particolari si verificano quando la testina occupa una delle estremità della configurazione. Per l'estremità sinistra, la configurazione $q_i\,bv$ produce $q_j\,cv$ se la transizione comporta una mossa a sinistra (per impedire alla macchina di superare l'estremità sinistra del nastro) e produce $c\,q_j\,v$ se la transizione comporta una mossa a destra. Per l'estremità destra, la configurazione $ua\,q_i$ è equivalente a $ua\,q_i\sqcup$ perché assumiamo che i simboli blank seguano la parte del nastro rappresentata dalla configurazione. In tal modo possiamo gestire questo caso come prima, quando la testina non occupa l'estremità destra. La ***configurazione iniziale*** di M su input w è la configurazione $q_0\,w$ che indica che la macchina è nello stato iniziale q_0 con la testina nella posizione più a sinistra sul nastro. In una ***configurazione di accettazione*** lo stato della configurazione è q_{accept}. In una ***configurazione di rifiuto*** lo stato della configurazione è q_{reject}. Le configurazioni di accettazione e rifiuto sono ***configurazioni di arresto*** e non producono ulteriori configurazioni. Poichè la macchina è definita in modo da fermarsi negli stati q_{accept} e q_{reject}, avremmo potuto

equivalentemente definire la funzione di transizione in una forma più complicata $\delta: Q' \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$, dove Q' è Q senza q_{accept} e q_{reject}. Una macchina di Turing M ***accetta*** l'input w se esiste una sequenza di configurazioni $C_1, C_2, \ldots, C_k$ tale che

1. C_1 è la configurazione iniziale di M su input w,
2. ogni C_i produce C_{i+1} e
3. C_k è una configurazione di accettazione.

L'insieme di stringhe che M accetta rappresenta il ***linguaggio di M***, o il ***linguaggio riconosciuto da M***, denotato con $L(M)$.

DEFINIZIONE **3.5**

Un linguaggio si dice ***Turing-riconoscibile*** se esiste una macchina di Turing che lo riconosce.[1]

Quando attiviamo una macchina di Turing su un input, tre risultati sono possibili. La macchina può *accettare*, *rifiutare*, oppure andare in *loop* (ciclare). Per ***loop*** si intende semplicemente che la macchina non si ferma. Un ciclo, in quanto tale, comporta un qualche comportamento semplice o complesso che non porta ad uno stato di arresto. Una macchina di Turing M può non accettare un input sia raggiungendo lo stato q_{reject} e rifiutando oppure andando in loop. A volte distinguere una macchina che è entrata in un loop da una che sta semplicemente impiegando molto tempo risulta difficile. Per questo motivo preferiamo macchine di Turing che si fermano su ogni singolo input; tali macchine non ciclano mai. Una tale macchina è detta ***decisore*** perché prende in ogni caso una decisione, sia essa di accettare o di rifiutare. Diciamo che un decisore ***decide*** un certo linguaggio se riconosce tale linguaggio.

DEFINIZIONE **3.6**

Un linguaggio si dice ***Turing-decidibile*** o semplicemente ***decidibile*** se esiste una macchina di Turing che lo decide.[2]

[1] In alcuni testi è detto ***linguaggio ricorsivamente enumerabile***.
[2] In alcuni testi è detto ***linguaggio ricorsivo***.

Nel seguito, diamo alcuni esempi di linguaggi decidibili. Ogni linguaggio decidibile è Turing-riconoscibile. Presenteremo alcuni esempi di linguaggi che sono Turing riconoscibili ma non decidibili dopo aver sviluppato, nel Capitolo 4, una tecnica per dimostrare l'indecidibilità.

Esempi di macchine di Turing

Come abbiamo fatto nel caso degli automi finiti e a pila, possiamo descrivere formalmente una particolare macchina di Turing specificando ciascuna delle sue sette parti. Tuttavia, scendere a quel livello di dettaglio può essere scomodo, tranne che per macchine di Turing molto piccole. Di conseguenza, non sprecheremo molte risorse per dare tali descrizioni. Per lo più daremo solo descrizioni ad alto livello perché sono abbastanza precise per i nostri scopi e sono molto più semplici da comprendere. Tuttavia, è importante ricordare che ogni descrizione ad alto livello è solo un'abbreviazione della sua controparte formale. Con pazienza e attenzione potremmo fornire una descrizione formale completa di ognuna delle macchine di Turing in questo libro. Per aiutarvi a stabilire il nesso tra le descrizioni formali e le descrizioni ad alto livello, nei prossimi due esempi mostreremo i diagrammi di stato.

ESEMPIO 3.7

In questo esempio descriviamo una macchina di Turing (TM) M_2 che decide $A = \{0^{2^n} \mid n \geq 0\}$, il linguaggio formato da tutte le stringhe di 0 la cui lunghezza è una potenza di 2.

M_2 = "su input w:

1. Muove la testina da sinistra a destra sul nastro, cancellando ogni secondo 0.
2. Se al passo 1, il nastro conteneva un solo 0, *accetta*.
3. Se al passo 1, il nastro conteneva più di un singolo 0 ed il loro numero era dispari, *rifiuta*.
4. Riporta la testina all'estremità sinistra del nastro.
5. Va al passo 1."

Ogni iterazione del passo 1 dimezza il numero di 0. Quando la macchina si muove lungo il nastro nel passo 1, ricorda se il numero di 0 è pari o dispari. Se tale numero è dispari e maggiore di 1, il numero di 0 iniziale in input non poteva essere una potenza di 2. Pertanto la macchina, su questa istanza, rifiuta. Tuttavia, se il numero di 0 è 1, il numero iniziale doveva essere una

potenza di 2. Quindi, in questo caso la macchina accetta. Ora diamo la descrizione formale di $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$,
- $\Gamma = \{0, \text{x}, \sqcup\}$.
- Descriviamo δ mediante un diagramma di stato (cfr. Figura 3.8).
- Gli stati iniziale, di accettazione e di rifiuto sono rispettivamente q_1, q_{accept} e q_{reject}.

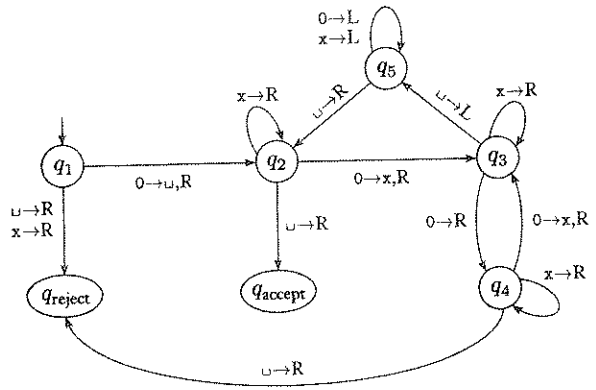


FIGURA 3.8
Diagramma di stato per la macchina di Turing M_2

In questo diagramma di stato, nella transizione da q_1 a q_2 compare l'etichetta 0→⊔,R. Questa etichetta indica che quando siamo nello stato q_1 e la testina legge 0, la macchina si porta nello stato q_2, scrive ⊔ e sposta la testina a destra. In altre parole, $\delta(q_1,0) = (q_2,\sqcup,\text{R})$. Per chiarezza utilizziamo l'abbreviazione 0→R per indicare la transizione da q_3 a q_4, il che significa che la macchina si sposta verso destra durante la lettura di 0 quando si trova nello stato q_3 ma non modifica il nastro, così $\delta(q_3,0) = (q_4,0,\text{R})$.

Questa macchina inizia scrivendo un simbolo blank sullo 0 più a sinistra del nastro in modo da poter identificare l'estrema sinistra del nastro nel passo 4. Mentre normalmente utilizzeremmo un simbolo più indicativo, quale #, come delimitatore di inizio nastro qui utilizziamo un blank per rispettare l'alfabeto del nastro e quindi mantenere piccolo il diagramma di stato. L'esempio 3.11 fornisce un altro metodo per determinare la fine del nastro a sinistra. In seguito diamo un esempio di esecuzione di questa

macchina su input 0000. La configurazione iniziale è $q_1$0000. La sequenza delle configurazioni della macchina è la seguente; leggete le colonne verso il basso e da sinistra a destra.

$q_1$0000	⊔q_5x0x⊔	⊔x$q_5$xx⊔
⊔$q_2$000	q_5⊔x0x⊔	⊔$q_5$xxx⊔
⊔x$q_3$00	⊔q_2x0x⊔	q_5⊔xxx⊔
⊔x0$q_4$0	⊔x$q_2$0x⊔	⊔$q_2$xxx⊔
⊔x0xq_3⊔	⊔xxq_3x⊔	⊔x$q_2$xx⊔
⊔x0q_5x⊔	⊔xxxq_3⊔	⊔xxq_2x⊔
⊔x$q_5$0x⊔	⊔xxq_5x⊔	⊔xxxq_2⊔
		⊔xxx⊔q_{accept}

ESEMPIO 3.9

Quella che segue è la descrizione formale di $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$, la macchina di Turing che abbiamo descritto in maniera informale (cfr. pagina 175) per decidere il linguaggio $B = \{w\#w \mid w \in \{0,1\}^*\}$.

- $Q = \{q_1, \ldots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0,1,\#\}$ e $\Gamma = \{0,1,\#,\text{x},\sqcup\}$.
- Descriviamo δ mediante il diagramma di stato (cfr. la figura seguente).
- Gli stati iniziale, di accettazione e di rifiuto sono rispettivamenmte q_1, q_{accept} e q_{reject}.

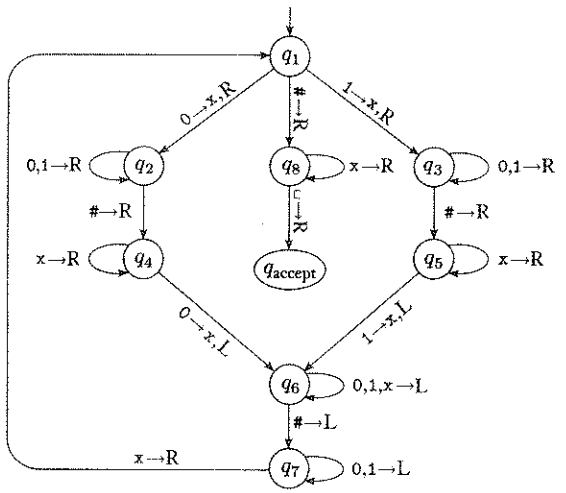


FIGURA 3.10
Diagramma dei stato per la macchina di Turing M_1

Nella Figura 3.10, che illustra il diagramma di stato della TM M_1, compare l'etichetta 0,1→R sulla transizione che va da q_3 a se stesso. Tale etichetta significa che la macchina rimane in q_3 ed effettua una mossa a destra quando legge uno 0 oppure un 1 nello stato q_3. Non cambia il simbolo sul nastro.

La fase 1 è attuata mediante gli stati da q_1 a q_7, e la fase 2 dagli stati rimanenti. Per semplificare la figura, non mostriamo lo stato di rifiuto e le transizioni che vanno nello stato di rifiuto. Tali transizioni avvengono implicitamente ogni volta che uno stato è privo di una transizione di uscita per un determinato simbolo. Quindi, poiché nello stato q_5 non è presente nessuna freccia uscente con il simbolo #, se un # è presente sotto la testina quando la macchina si trova nello stato q_5, la macchina va nello stato di rifiuto. Per completezza, aggiungiamo che la testina si muove a destra in ciascuna di queste transizioni, nello stato di rifiuto.

ESEMPIO 3.11

In questo esempio, una TM M_3 esegue operazioni di aritmetica elementare. Essa decide il linguaggio $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \times j = k \text{ e } i,j,k \geq 1\}$.

M_3 = "Sulla stringa di input w:

1. Muove la testina da sinistra a destra sul nastro per determinare se è un elemento di $\mathtt{a}^+\mathtt{b}^+\mathtt{c}^+$ e *rifiuta* se non lo è.
2. Riporta la testina all'estremità sinistra del nastro.
3. Barra una a e muove la testina a destra fino a trovare una b. Fa da spola tra le b e le c, barrandone una di ognuna fino al termine delle b. Se tutte le c sono state barrate e rimane qualche b, *rifiuta*.
4. Ripristina le b barrate e ripete la fase 3 se rimane qualche a da barrare. Se tutte le a sono state barrate, controlla se anche tutte le c sono state barrate. Se sì, *accetta*; altrimenti, *rifiuta*."

Esaminiamo più da vicino le quattro fasi di M_3. Nella fase 1, la macchina funziona come un automa finito. Nessuna scrittura è necessaria in quanto la testina si sposta da sinistra a destra, utilizzando gli stati al fine di determinare se l'input è nella forma corretta. La fase 2 sembra altrettanto semplice, ma contiene una sottigliezza. Come fa la TM a trovare l'estremità sinistra del nastro? Trovare l'estremità destra dell'input è facile perché esso termina con un simbolo blank. Ma, inizialmente l'estremità sinistra non ha alcun simbolo che permette di individuarla. Una tecnica che permette alla macchina di trovare l'estremità sinistra del nastro consiste nel marcare in qualche modo il simbolo più a sinistra quando la macchina parte con la testina su tale simbolo. A questo punto la macchina può eseguire una scansione a sinistra finché non trova il simbolo marcato, quando vuole riportare

la testina fino all'estremità sinistra del nastro. L'esempio 3.7 illustra questa tecnica; si utilizza un simbolo blank per marcare l'estremità sinistra del nastro.

Un metodo più sottile per trovare l'estremità sinistra del nastro sfrutta il modo in cui abbiamo definito il modello di macchina di Turing. Ricordiamo che, se la macchina tenta di muovere la testina oltre l'estremità sinistra del nastro, rimane ferma in quella stessa posizione. Possiamo usare questa caratteristica per creare un rilevatore dell'estremità sinistra. Per rilevare se la testina è situata sul lato estremo sinistro, la macchina può scrivere un simbolo speciale sulla posizione corrente, mentre memorizza nel controllo il simbolo che ha sostituito. Può quindi tentare di spostare la testina a sinistra. Se è ancora sopra il simbolo speciale, il movimento verso sinistra non è avvenuto e quindi la testina deve essere per forza all'estremità sinistra. Se invece si ritrova su un simbolo diverso, alcuni simboli stanno a sinistra di quella posizione sul nastro. Prima di andare oltre, la macchina deve assicurarsi di risostituire il simbolo modificato con quello originale. Le fasi 3 e 4 hanno implementazioni semplici e utilizzano ciascuna stati diversi.

ESEMPIO 3.12

Qui, una TM M_4 sta risolvendo quello che viene chiamato il problema degli *elementi distinti*. Ha in input un elenco di stringhe su $\{0,1\}$ separate da simboli # ed il suo compito è accettare se tutte le stringhe sono diverse. Il linguaggio è

$$E = \{\#x_1\#x_2\#\cdots\#x_l \mid \text{ogni } x_i \in \{0,1\}^* \text{ e } x_i \neq x_j \text{ per ogni } i \neq j\}.$$

La macchina M_4 funziona confrontando x_1 con x_2 mediante x_l, poi confrontando x_2 con x_3 mediante x_l, e così via. Le seguente è una descrizione informale della TM M_4 che decide questo linguaggio.

M_4 = "Sulla stringa di input w:

1. Mette un segno sul simbolo del nastro più a sinistra. Se questo simbolo era un blank, *accetta*. Se il simbolo era un #, continua con la fase successiva. Altrimenti, *rifiuta*.
2. Scorre a destra fino al successivo # e vi mette sopra un secondo segno. Se nessun # viene trovato prima di un simbolo blank, allora era presente solo x_1, quindi *accetta*.
3. Procede a zig-zag confrontando le due stringhe a destra dei simboli # segnati. Se sono uguali, *rifiuta*.
4. Sposta il più a destra dei due segni sul successivo simbolo # a destra. Se non viene trovato nessun simbolo # prima di un simbolo blank, sposta il segno più a sinistra sul successivo # alla sua destra e sposta il segno più a destra sul successivo #. Questa volta, se un simbolo # non è disponibile dopo il

segno più a destra, allora vuol dire che tutte le stringhe sono state confrontate, quindi *accetta*.

5. Va alla fase 3."

Questa macchina illustra la tecnica di marcatura dei simboli del nastro. Nella fase 2, la macchina pone un segno al di sopra di un simbolo, # in questo caso. Nell'effettiva implementazione, la macchina ha due simboli differenti, # e $\dot{\#}$, nell'alfabeto del proprio nastro. Dire che la macchina mette un segno sopra un # significa che la macchina scrive il simbolo $\dot{\#}$ in quella locazione. Rimuovere il segno vuol dire che la macchina scrive il simbolo # senza il puntino sopra. In generale, potremmo voler marcare vari simboli del nastro. Per fare ciò si può semplicemente includere le versioni col puntino di tutti questi simboli nell'alfabeto del nastro.

Concludiamo dagli esempi precedenti che i linguaggi descritti A, B, C ed E sono decidibili. Tutti i linguaggi decidibili sono Turing-riconoscibili, quindi questi linguaggi sono anche Turing-riconoscibili. Dimostrare che un linguaggio è Turing-riconoscibile, ma non decidibile è più difficile, lo faremo nel Capitolo 4.

3.2 VARIANTI DI MACCHINE DI TURING

Esistono molte definizioni alternative di macchine di Turing, comprese le versioni multinastro o non deterministiche. Sono dette ***varianti*** del modello macchina di Turing. Il modello originale e le sue ragionevoli varianti hanno tutti lo stesso potere computazionale – riconoscono la stessa classe di linguaggi. In questa sezione descriviamo alcune di queste varianti e le relative prove di equivalenza in termini di potenzialità. Chiamiamo ***robustezza*** questa invarianza ad alcune variazioni nella definizione. Sia automi a stati finiti che automi a pila sono modelli piuttosto robusti, ma le macchine di Turing hanno un sorprendente grado di robustezza. Per illustrare la robustezza del modello di macchina di Turing cerchiamo di variare il tipo di funzione di transizione consentito. Nella nostra definizione, la funzione di transizione forza la testina a spostarsi verso sinistra o destra ad ogni passo; la testina non può semplicemente restare ferma. Supponiamo di aver permesso alla macchina di Turing la capacità di restare ferma. La funzione di transizione avrebbe allora la forma $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$. Questa caratteristica permette alle macchine di Turing di riconoscere ulteriori linguaggi, aggiungendo così potenza al modello? Certo che no, perché siamo in grado di convertire qualsiasi TM con la possibilità di "restar ferma" in una che non ha tale capacità. Lo facciamo sostituendo ogni transizione "resta ferma" con due transizioni, una che sposta la testina a destra e una che la riporta a sinistra. Questo piccolo esempio contiene la chiave per

mostrare l'equivalenza di varianti di TM. Per dimostrare che due modelli sono equivalenti abbiamo semplicemente bisogno di dimostrare che ognuno dei due può simulare l'altro.

Macchina di Turing multinastro

Una ***macchina di Turing multinastro*** è come una normale macchina di Turing con vari nastri. Ogni nastro ha la sua testina per la lettura e la scrittura. Inizialmente l'input si trova sul nastro 1, mentre gli altri nastri sono vuoti. La funzione di transizione viene modificata per consentire la lettura, la scrittura e lo spostamento della testina contemporaneamente su alcuni o tutti i nastri. Formalmente,

$$\delta\colon Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k,$$

dove k è il numero di nastri. L'espressione

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \mathrm{L}, \mathrm{R}, \ldots, \mathrm{L})$$

significa che, se la macchina si trova nello stato q_i e le testine da 1 a k leggono i simboli da a_1 ad a_k, la macchina va nello stato q_j, scrive i simboli da b_1 a b_k e muove ogni testina a sinistra o a destra, o la fa restare ferma, come specificato. Le macchine di Turing multinastro sembrano più potenti delle macchine di Turing ordinarie, ma possiamo dimostrare che sono equivalenti in potenza di calcolo. Ricordiamo che due macchine sono equivalenti se riconoscono lo stesso linguaggio.

TEOREMA 3.13

Per ogni macchina di Turing multinastro esiste una macchina di Turing a nastro singolo equivalente.

DIMOSTRAZIONE. Mostriamo come convertire una macchina di Turing multinastro M in una TM S equivalente a nastro singolo. L'idea chiave è quella di mostrare come simulare M con S. Supponiamo che M abbia k nastri. Allora S simula l'effetto di k nastri memorizzando le loro informazioni sul suo singolo nastro. Essa utilizza il nuovo simbolo # come delimitatore per separare i contenuti dei diversi nastri. Oltre al contenuto dei nastri, S deve tenere traccia delle posizioni delle testine. Lo fa scrivendo un simbolo con un punto sopra per contrassegnare la posizione in cui si troverebbe la testina di quel nastro. Pensate a questi come nastri e testine "virtuali". Come in precedenza, i simboli del nastro "puntati" sono semplicemente simboli nuovi che sono stati aggiunti all'alfabeto del nastro. La figura seguente illustra come un singolo nastro può essere utilizzato per rappresentare tre nastri.

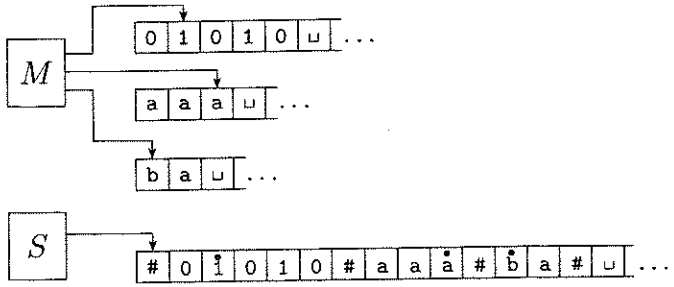


FIGURA 3.14
Rappresentazione di tre nastri con uno solo

S = "Su input $w = w_1 \cdots w_n$:

1. Inizialmente S mette il suo nastro nel formato che rappresenta tutti i k nastri di M. Il nastro formattato contiene
$$\#\dot{w}_1 w_2 \cdots w_n \#\dot{\sqcup}\#\dot{\sqcup}\# \cdots \#.$$
2. Per simulare una singola mossa, S scansiona il suo nastro dal primo #, che segna l'estremità sinistra, al $(k+1)$mo #, che segna l'estremità di destra, per determinare i simboli puntati dalle testine virtuali. Successivamente S fa un secondo passaggio per aggiornare i nastri in accordo alla funzione di transizione di M.
3. Se in qualsiasi momento S sposta una delle testine virtuali a destra su un #, questa azione significa che M ha spostato la testina corrispondente sulla parte di nastro vuota non letta in precedenza. Quindi S scrive un simbolo blank in questa cella del nastro e sposta il contenuto del nastro, da questa cella fino al simbolo # più a destra, di una unità a destra. Poi prosegue la simulazione come prima."

COROLLARIO 3.15

Un linguaggio è Turing-riconoscibile se e solo se qualche macchina di Turing multinastro lo riconosce.

DIMOSTRAZIONE. Un linguaggio Turing-riconoscibile è riconosciuto da una macchina di Turing ordinaria (ad un solo nastro), che è un caso particolare di una macchina di Turing multinastro. Questo prova un verso di questo corollario. L'altro verso segue dal Teorema 3.13.

Macchina di Turing non deterministica

Una macchina di Turing non deterministica è definita come ci si aspetta. In qualsiasi punto della computazione la macchina può procedere effettuando varie scelte. La funzione di transizione di una macchina di Turing non deterministica è della forma

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

La computazione di una macchina di Turing non deterministica è un albero i cui rami corrispondono alle diverse scelte possibili previste per la macchina. Se qualche ramo della computazione porta allo stato di accettazione, allora la macchina accetta l'input. Se sentite la necessità di rivedere il non determinismo, andate alla Sezione 1.2 (pagina 50). Ora mostriamo che il non determinismo non influisce sulla potenza di calcolo del modello macchina di Turing.

TEOREMA 3.16

Per ogni macchina di Turing non deterministica esiste una macchina di Turing deterministica equivalente.

IDEA. Possiamo simulare qualsiasi TM non deterministica N con una TM deterministica D. L'idea di base della simulazione consiste nel provare tutte le possibili scelte che può fare N durante la sua computazione nondeterministica. Se D trova lo stato di accettazione su uno qualsiasi di questi rami, allora D accetta. Altrimenti, la simulazione di D non terminerà. Guardiamo alla computazione di N su un input w come ad un albero. Ogni ramo dell'albero rappresenta una scelta non deterministica. Ogni nodo dell'albero è una configurazione di N. La radice dell'albero è la configurazione iniziale. La TM D esplora questo albero alla ricerca di una configurazione di accettazione. Eseguire accuratamente questa ricerca è cruciale perché D potrebbe non visitare l'intero albero. Un'idea allettante, anche se cattiva, è quella di fare esplorare a D l'albero utilizzando una visita in profondità (*depth-first search*). La strategia della visita in profondità esplora un cammino fino in fondo prima di risalire ed eseguire l'esplorazione di altri cammini. Se D esplorasse l'albero in questo modo, essa potrebbe rimanere per sempre ad esplorare un eventuale cammino infinito e non trovare una configurazione accettante in qualche altro cammino. Quindi progettiamo D in modo da esplorare l'albero utilizzando, invece una visita in ampiezza (*breadth-first search*). Questa strategia esplora tutti i cammini che terminano alla stessa profondità prima di andare a esplorare ogni cammino che termina alla profondità successiva. Questo metodo garantisce che D visita ogni nodo dell'albero finché non incontra una configurazione di accettazione.

DIMOSTRAZIONE La TM D deterministica che simula ha tre nastri. Dal Teorema 3.13, questa disposizione equivale ad avere un singolo nastro. La macchina D utilizza i suoi tre nastri in maniera particolare, come illustrato nella figura seguente. Il nastro 1 contiene sempre la stringa di input e non viene mai modificato. Il nastro 2 mantiene una copia del nastro di N corrispondente a qualche diramazione della sua computazione non deterministica. Il nastro 3 tiene traccia della posizione di D nell'albero delle computazioni di N.

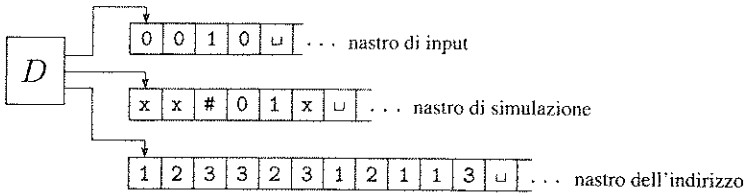


FIGURA 3.17
La TM deterministica D che simula la TM non deterministica N

Consideriamo in primo luogo la rappresentazione dei dati sul nastro 3. Ogni nodo dell'albero può avere al massimo b figli, dove b è la dimensione del più grande insieme di scelte possibili date dalla funzione di transizione di N. Ad ogni nodo della struttura assegniamo un indirizzo che è una stringa sull'alfabeto $\Gamma_b = \{1, 2, \ldots, b\}$. Assegniamo l'indirizzo 231 al nodo a cui si arriva partendo dalla radice, spostandosi al suo secondo figlio, spostandosi ancora da tale nodo al suo terzo figlio ed infine spostandosi al primo figlio di quest'ultimo nodo. Ogni simbolo della stringa ci dice quale deve essere la scelta successiva, durante la simulazione di un passo in una ramificazione di una computazione non deterministica di N. A volte un simbolo può non corrispondere ad una scelta se sono disponibili troppe poche scelte per una configurazione. In tal caso l'indirizzo non è valido e non corrisponde ad alcun nodo. Il nastro 3 contiene una stringa su Γ_b. Essa rappresenta la ramificazione della computazione di N dalla radice al nodo indirizzato da tale stringa, a meno che l'indirizzo sia non valido. La stringa vuota è l'indirizzo della radice dell'albero. Siamo ora pronti a descrivere D.

1. Inizialmente il nastro 1 contiene l'input w e i nastri 2 e 3 sono vuoti.
2. Copia il nastro 1 sul nastro 2 ed inizializza la stringa sul nastro 3 a ε.
3. Utilizza il nastro 2 per simulare N con input w su una ramificazione della sua computazione non deterministica. Prima di ogni passo di N, consulta il simbolo successivo sul nastro 3 per determinare quale scelta fare tra quelle consentite dalla funzione di transizione di n. Se non rimangono più simboli sul nastro 3 o se questa scelta non deterministica non è valida, interrompe questo cammino andando alla fase 4. Va alla fase 4 anche quando si verifica una configurazione di rifiuto. Se incontra una configurazione di accettazione, allora accetta l'input.
4. Sostituisce la stringa sul nastro 3 con la stringa successiva rispetto all'ordine sulle stringhe. Simula la ramificazione successiva della computazione di N andando al passo 2.

COROLLARIO 3.18

Un linguaggio è Turing-riconoscibile se e solo se esiste una macchina di Turing non deterministica che lo riconosce.

DIMOSTRAZIONE. Qualsiasi TM deterministica è automaticamente una TM non deterministica e quindi una direzione di questo teorema è già dimotrata. L'altra direzione segue dal Teorema 3.16.

Possiamo modificare la dimostrazione del Teorema 3.16 in modo che se N si ferma sempre su tutte le ramificazioni durante la computazione, D si ferma sempre. Chiameremo una macchina non deterministica ***decisore*** se tutte le ramificazioni si fermano su ogni input. L'Esercizio 3.3 chiede di modificare la dimostrazione in tal senso per ottenere il seguente corollario al Teorema 3.16.

COROLLARIO 3.19

Un linguaggio è decidibile se e solo se esiste una macchina di Turing non deterministica che lo decide.

Enumeratori

Come accennato in precedenza, alcune persone usano il termine *linguaggio ricorsivamente enumerabile* per indicare un linguaggio Turing-riconoscibile. Questo termine deriva da una variante di macchina di Turing chiamata ***enumeratore***. Definito in modo informale, un enumeratore è una macchina di Turing con una stampante collegata. La macchina di Turing può utilizzare tale stampante come dispositivo di output per stampare stringhe. Ogni volta che la macchina di Turing vuole aggiungere una stringa alla lista, invia la stringa alla stampante. L'Esercizio 3.4 chiede di fornire una definizione formale di enumeratore. La figura seguente fornisce uno schema di questo modello.

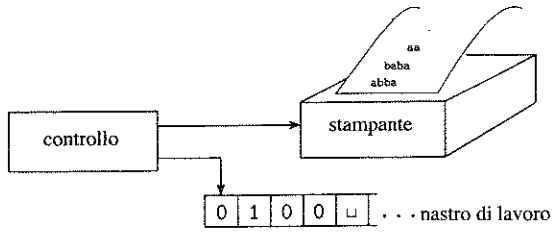


FIGURA 3.20
Schema di enumeratore

Un enumeratore E inizia con un nastro di input vuoto. Se l'enumeratore non si ferma, esso può stampare un elenco infinito di stringhe. Il linguaggio enumerato da E è la collezione di tutte le stringhe che esso stampa. Inoltre, E potrebbe generare le stringhe del linguaggio in qualsiasi ordine, eventualmente anche con ripetizioni. Adesso siamo pronti a stabilire il nesso tra enumeratori e linguaggi Turing-riconoscibili.

TEOREMA 3.21

Un linguaggio è Turing-riconoscibile se e solo se esiste un enumeratore che lo enumera.

DIMOSTRAZIONE. Come prima cosa, dimostriamo che se abbiamo un enumeratore E che enumera un linguaggio A, allora esiste una TM M che riconosce A. La TM M funziona come segue.

M = "Su input w:

1. Esegue E. Ogni volta che E genera una stringa, la confronta con w.
2. Se w appare nell'output di E, *accetta*."

Chiaramente, M accetta quelle stringhe che compaiono sulla lista di E'.

Ora occupiamoci dell'altra direzione. Se la TM M riconosce un linguaggio A, possiamo costruire il seguente enumeratore E per A. Sia $s_1, s_2, s_3, \ldots$ una lista di tutte possibili stringhe in Σ^*.

E = "Ignora l'input.

1. Ripete i seguenti passi per $i = 1, 2, 3, \ldots$.
2. Esegue M per i passi su ogni input, $s_1, s_2, \ldots, s_i$.
3. Se una qualche computazione accetta, stampa la corrispondente s_j."

Se M accetta una particolare stringa s, alla fine s apparirà nella lista generata da E. In realtà, essa apparirà nella lista un numero infinito di volte, perché M ritorna all'inizio su ogni stringa per ogni ripetizione del passo 1.

Questa procedura fa sì che M lavori in parallelo su tutte le possibili stringhe di input.

Equivalenza con altri modelli

Finora abbiamo presentato diverse varianti di macchina di Turing e abbiamo dimostrato che sono equivalenti dal punto di vista della potenza di calcolo. Sono stati proposti molti altri modelli di computazione universali. Alcuni di questi modelli sono molto simili alle macchine di Turing, mentre altri risultano molto diversi. Tutti condividono la stessa caratteristica fondamentale delle macchine di Turing – un accesso non restrittivo ad una memoria illimitata – che li distingue da modelli più deboli, come gli automi a stati finiti e gli automi a pila. Sorprendentemente, *tutti* i modelli con tale caratteristica risultano essere equivalenti purché soddisfino alcuni ragionevoli requisiti.[3]

Per capire questo fenomeno consideriamo la situazione analoga per i linguaggi di programmazione. Molti, come Pascal e LISP, sembrano abbastanza diversi tra loro per stile e struttura. Esiste un algoritmo che può essere programmato in uno di loro e non nell'altro? Certamente no, siamo in grado di compilare LISP in Pascal e Pascal in LISP, il che significa che i due linguaggi descrivono *esattamente* la stessa classe di algoritmi. Lo stesso avviene per gli altri linguaggi di programmazione ragionevoli. L'equivalenza diffusa dei modelli di calcolo vale per la stessa ragione. Ogni coppia di modelli di computazione che soddisfano determinati requisiti possono simularsi a vicenda e sono quindi equivalenti in termini di potere computazionale.

Questo fenomeno di equivalenza ha un importante corollario filosofico. Anche se siamo in grado di immaginare molteplici modelli di calcolo, la classe di algoritmi che essi descrivono rimane la stessa. Mentre ogni modello di computazione ha una sua arbitrarietà nella propria definizione, la classe degli algoritmi che esso descrive è naturale, perché gli altri modelli arrivano alla stessa, unica classe. Questo fenomeno ha avuto profonde implicazioni per la matematica, come vedremo nella prossima sezione.

3.3
LA DEFINIZIONE DI ALGORITMO

Parlando in maniera informale, un *algoritmo* è un insieme di istruzioni semplici per l'esecuzione di un certo compito. Nella vita di tutti i giorni,

[3]Ad esempio, un requisito è la capacità di eseguire solamente una quantità finita di lavoro in un unico passo

gli algoritmi vengono a volte chiamati *procedure* o *ricette*. Gli algoritmi svolgono anche un ruolo importante in matematica. L'antica letteratura matematica contiene descrizioni di algoritmi per vari compiti, come la ricerca di numeri primi o del massimo comun divisore. Nella matematica contemporanea gli algoritmi abbondano. Anche se gli algoritmi hanno avuto una lunga storia nel campo della matematica, la nozione di algoritmo non è stata formalizzata con precisione fino al ventesimo secolo. Prima di allora, i matematici avevano una nozione intuitiva di algoritmo, e invocavano questo concetto quando li usavano o li descrivevano. Ma tale nozione intuitiva non era sufficiente per raggiungere una comprensione più profonda degli algoritmi. La seguente storia racconta come una definizione precisa di algoritmo sia stata cruciale nel caso di un importante problema matematico.

I problemi di Hilbert

Nel 1900, il matematico David Hilbert pronunciò al Congresso Internazionale dei Matematici a Parigi un discorso divenuto ormai famoso. Nella sua conferenza, identificò 23 problemi matematici e li pose come sfida per il secolo successivo. Il decimo problema sulla sua lista riguardava gli algoritmi.

Prima di descrivere il problema, introduciamo brevemente i polinomi. Un ***polinomio*** è una somma di termini, dove ogni ***termine*** è il prodotto di alcune variabili ed una costante, chiamata ***coefficiente***. Per esempio

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

è un termine con coefficiente 6, e

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

è un polinomio con quattro termini sulle variabili x, y e z.

In questa discussione, consideriamo come coefficienti solo numeri interi. Una ***radice*** di un polinomio è un'assegnazione di valori alle sue variabili per cui il valore del polinomio è 0. Questo polinomio ha radice $x = 5$, $y = 3$ e $z = 0$. Questa radice è una ***radice intera***, perché a tutte le variabili sono assegnati valori interi. Alcuni polinomi hanno una radice intera ed altri no.

Il decimo problema di Hilbert consiste nell'ideare un algoritmo per verificare se un polinomio abbia o meno una radice intera. Hilbert on usò il termine *algoritmo*, piuttosto "un processo in base al quale esso può essere determinato da un numero finito di operazioni"[4]. È interessante notare che nel modo in cui ha formulato questo problema, Hilbert ha chiesto esplicitamente che fosse "progettato" un algoritmo. In tal modo, egli diede per

[4]Tradotto dall'originale in tedesco

scontato l'esistenza di un tale algoritmo – era solo necessario che qualcuno lo trovasse. Come ora sappiamo, non esiste un algoritmo per questo problema; si tratta di un problema non risolvibile algoritmicamente. Per i matematici di quel periodo sarebbe stato praticamente impossibile giungere ad una tale conclusione sulla base del loro concetto intuitivo di algoritmo. Il concetto intuitivo poteva essere sufficiente per fornire algoritmi per alcuni compiti, ma era inutilizzabile per dimostrare che non esisteva alcun algoritmo per un determinato tipo di problema. Per poter provare che un algoritmo non esiste è necessario avere una definizione di algoritmo chiara. Per poter avere progressi sul decimo problema si è dovuto attendere una tale definizione. La definizione è arrivata nel 1936 da parte di Alonzo Church ed Alan Turing.

Church utilizzò un sistema di notazione detto λ-calcolo per definire gli algoritmi. Turing lo fece con le sue "macchine". È stato dimostrato che queste due definizioni sono equivalenti. Questa connessione tra la nozione informale di algoritmo e la relativa definizione precisa è ora chiamata: la ***tesi di Church–Turing***.

La tesi di Church–Turing fornisce la definizione di algoritmo necessaria per risolvere il decimo problema di Hilbert. Nel 1970, Yuri Matijasevič, basandosi sui lavori di Martin Davis, Hilary Putnam, e Julia Robinson, dimostrò che non esiste alcun algoritmo capace di verificare se un polinomio ha radici intere. Nel Capitolo 4 svilupperemo le tecniche che costituiscono la base per dimostrare che questo e altri problemi sono algoritmicamente irrisolvibili.

Nozione intuitiva di algoritmo	equivalente a	*Algoritmi mediante Macchina di Turing*

FIGURA 3.22
La tesi di Church–Turing

Riscriviamo il decimo problema di Hilbert nella nostra terminologia. In tal modo introduciamo alcuni temi che analizzeremo nei Capitoli 4 e 5. Sia

$$D = \{p \mid p \text{ è un polinomio avente radice intera}\}.$$

Il decimo problema di Hilbert chiede in sostanza se l'insieme D è decidibile. La risposta è negativa. Di contro possiamo dimostrare che D è Turing-riconoscibile. Prima di fare ciò, consideriamo un problema più semplice. Si tratta di un analogo del decimo problema di Hilbert per i polinomi che hanno un'unica variabile, come per esempio $4x^3 - 2x^2 + x - 7$. Sia

$$D_1 = \{p \mid p \text{ è un polinomio su } x \text{ avente una radice intera}\}.$$

Ecco una TM M_1 che riconosce D_1:

M_1 = "Su input $\langle p \rangle$: dove p è un polinomio sulla variabile x.

1. Valuta p con x posta successivamente ai valori 0, 1, -1, 2, -2, 3, -3, ... Se in un qualsiasi momento la valutazione del polinomio è 0, *accetta*."

Se p ha una radice intera, M_1 ad un certo punto la trova e accetta. Se p non ha una radice intera, M_1 sarà in esecuzione per sempre. Nel caso di più variabili, possiamo presentare una TM M simile che riconosce D. Ora M considera tutte le possibili impostazioni delle variabili con valori interi. Sia M_1 che M sono riconoscitori, ma non decisori. Possiamo convertire M_1 in un decisore per D_1, perché possiamo calcolare dei limiti all'intervallo di valori in cui possono trovarsi le radici di un polinomio a singola variabile e limitare la ricerca a questo intervallo di valori. Nel Problema 3.10, viene chiesto di dimostrare che le radici di tale polinomio devono essere comprese tra i valori

$$\pm k \frac{c_{\max}}{c_1},$$

dove k è il numero di termini del polinomio, $c_{\max}$ è il coefficiente avente il massimo valore assoluto, e c_1 è il coefficiente del termine di ordine più elevato. Se non si trova una radice all'interno di questi limiti, la macchina *rifiuta*. Il teorema di Matijaseviù c mostra che non è possibile calcolare limiti analoghi per polinomi a più variabili.

Terminologia per la descrizione di macchine di Turing

Siamo giunti a un punto di svolta nello studio della teoria della computazione. Continuiamo a parlare di macchine di Turing, ma il centro reale del nostro interesse saranno gli algoritmi. Cioè, le macchine di Turing servono soltanto come modello preciso per la definizione di algoritmo. Salteremo la vasta teoria delle macchine di Turing in quanto tali e non dedicheremo molto tempo alla programmazione a basso livello delle macchine di Turing. Abbiamo solamente bisogno di prendere abbastanza familiarità con le macchine di Turing da credere che esse catturano tutti gli algoritmi. A tal scopo, cerchiamo di standardizzare il modo in cui descriviamo gli algoritmi mediante una macchina di Turing. Inizialmente, ci chiediamo: qual è il giusto livello di dettaglio da fornire quando descriviamo un tale algoritmo? Gli studenti generalmente pongono questa domanda, soprattutto quando preparano soluzioni ad esercizi e problemi. Consideriamo tre possibilità. La prima è la *descrizione formale* che definisce in pieno gli stati della macchina di Turing, la funzione di transizione e così via. Rappresenta il livello di descrizione più basso e dettagliato. La seconda è una descrizione ad un livello più alto, detta *descrizione implementativa*, in cui descriviamo verbalmente il modo in cui la macchina di Turing muove la testina e il modo in cui memorizza i dati sul nastro. A questo livello non

diamo dettagli sugli stati o sulla funzione di transizione. La terza è la *descrizione ad alto livello* in cui usiamo la lingua italiana per descrivere un algoritmo, ignorando i dettagli dell'implementazione. A questo livello, non abbiamo bisogno di spiegare come la macchina gestisce il nastro o la testina.

In questo capitolo abbiamo formalizzato e dato una descrizione implementativa di vari esempi di macchine di Turing. Far pratica con descrizioni a basso livello di macchine di Turing aiuta a capire le macchine di Turing e ad acquisire dimestichezza nel loro utilizzo. Una volta acquisita una certa sicurezza, saranno sufficienti descrizioni di alto livello.

Stabiliamo ora un formato e la notazione che utilizzeremo per descrivere le macchine di Turing. L'input di una macchina di Turing è sempre una stringa. Se volessimo fornire in input un oggetto diverso da una stringa, dovremmo prima rappresentare tale oggetto come una stringa. Le stringhe possono rappresentare facilmente polinomi, grafi, grammatiche, automi e qualsiasi combinazione di questi oggetti. Una macchina di Turing può essere programmata per decodificarne la rappresentazione in modo che possa essere interpretata nel modo corretto. La nostra notazione per la codifica di un oggetto O nella sua rappresentazione sotto forma di stringa è $\langle O \rangle$. Se abbiamo vari oggetti $O_1, O_2, \ldots, O_k$, denotiamo la loro codifica con una singola stringa $\langle O_1, O_2, \ldots, O_k \rangle$. La codifica stessa può essere fatta in vari modi ragionevoli. Non importa quale scegliamo perché una macchina di Turing può sempre trasformare una di tali codifiche in un'altra.

Nel nostro formato, descriviamo gli algoritmi delle macchine di Turing con un segmento di testo tra parentesi. Dividiamo l'algoritmo in fasi, ciascuna fase di solito consiste di più passi di calcolo della macchina di Turing. Indichiamo la struttura a blocchi dell'algoritmo con un' indentazione ulteriore. La prima riga dell'algoritmo descrive l'input alla macchina. Se la descrizione dell'input è semplicemente w, l'input è considerato una stringa. Se la descrizione dell'input è la codifica di un oggetto del tipo $\langle A \rangle$, la macchina di Turing dapprima implicitamente controlla se l'input codifica correttamente un oggetto della forma desiderata, rifiutando in caso contrario.

ESEMPIO 3.23

Sia A il linguaggio costituito da tutte le stringhe che rappresentano grafi non orientati connessi. Ricordiamo che un grafo si dice ***connesso*** se ogni nodo può essere raggiunto da ogni altro nodo spostandosi lungo gli archi del grafo. Scriviamo

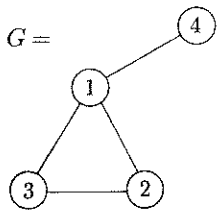
$$A = \{\langle G \rangle | \ G \text{ è un grafo connesso non orientato}\}.$$

La seguente è una descrizione ad alto livello di una TM M che decide A.

M = "Su input $\langle G \rangle$, la codifica di un grafo G:

1. Seleziona il primo nodo di G e lo marca.
2. Ripete la fase seguente fino a quando non vengono più marcati nuovi nodi:
3. per ogni nodo in G, marcalo se esso è connesso con un arco ad un nodo già marcato.
4. Esamina tutti i nodi di G per determinare se sono tutti marcati. Se lo sono, *accetta*, altrimenti, *rifiuta*."

Come ulteriore esercizio, esaminiamo alcuni dettagli implementativi della macchina di Turing M. Di solito nel seguito non forniremo questo livello di dettaglio e neppure ne avrete bisogno, se non espressamente richiesto in un esercizio. In primo luogo, dobbiamo capire come $\langle G \rangle$ codifica il grafo G come una stringa. Consideriamo una codifica che consiste in una lista dei nodi di G seguita da un elenco degli archi di G. Ogni nodo è un numero decimale ed ogni arco è denotato dalla coppia di numeri decimali che rappresentano i nodi ai due estremi dell'arco. La figura seguente mostra un grafo e la sua codifica .



$\langle G \rangle$ =

(1,2,3,4)((1,2),(2,3),(3,1),(1,4))

FIGURA 3.24
Un grafo G e la sua codifica $\langle G \rangle$

Quando M riceve l'input $\langle G \rangle$, verifica innanzitutto che l'input sia effettivamente la codifica di un grafo. Per fare ciò, M scandisce il nastro per assicurarsi che ci siano due liste e che siano nella forma corretta. Il primo elenco deve essere un elenco di numeri decimali distinti ed il secondo dovrebbe essere un elenco di coppie di numeri decimali. Successivamente, M controlla varie cose. In primo luogo, l'elenco dei nodi non deve contenere ripetizioni; in secondo luogo, ogni nodo che figura nell'elenco degli archi deve apparire anche nella lista dei nodi. Per il primo, si può utilizzare la procedura descritta nell'Esempio 3.12 per la TM M_4 che controlla la diversità degli elementi. Un metodo simile funziona per il secondo controllo. Se l'input supera questi controlli, allora esso è la codifica di qualche grafo G. Questa verifica completa il controllo dell'input, ed M va alla fase 1. Per la

fase 1, M segna il primo nodo con un punto sulla cifra più a sinistra. Per la fase 2, M scorre la lista dei nodi per trovare un nodo n_1 non puntato (cioè marcato con il punto) e lo contrassegna in modo diverso – diciamo, sottolineando il primo simbolo. Poi M scorre nuovamente la lista per trovare un altro nodo n_2 puntato e sottolinea anche questo. Ora M scorre la lista degli archi. Per ogni arco, M controlla se i due nodi sottolineati n_1 e n_2 sono quelli che compaiono in tale arco. Se lo sono, M punta n_1, rimuove la sottolineatura e riprende dall'inizio della fase 2. Altrimenti, se non lo sono, M controlla l'arco successivo sulla lista. Se non ci sono più archi, $\{n_1, n_2\}$ non è un arco di G. Quindi M muove la sottolineatura da n_2 al successivo nodo puntato e chiama ora questo nuovo nodo n_2. Ripete la procedura descritta in questo paragrafo per controllare, come prima, se la nuova coppia $\{n_1, n_2\}$ è un arco. Se non ci sono più nodi puntati, n_1 non è collegato a nessun nodo puntato. Quindi M imposta la sottolineatura in modo che n_1 risulti il successivo nodo non puntato ed n_2 sia il primo nodo puntato e ripete la procedura descritta in questo paragrafo. Se non ci sono più nodi senza punto, M non è stata in grado di trovare nuovi nodi da marcare, quindi passa alla fase 4. Durante la fase 4, M scorre l'elenco dei nodi per determinare se sono tutti puntati. Se lo sono, entra nello stato di accettazione, altrimenti entra nello stato di rifiuto. Questo completa la descrizione della TM M.

ESERCIZI

3.1 Questo esercizio riguarda la TM M_2, la cui descrizione e diagramma di stato appaiono nell'Esempio 3.7. Per ciascuna delle parti, fornire la sequenza di configurazioni di M_2 quando parte avendo in input la stringa indicata.

a. 0.
[A]b. 00.
c. 000.
d. 000000.

3.2 Questo esercizio riguarda la TM M_1, la cui descrizione e diagramma di stato appaiono nell'Esempio 3.9. Per ciascuna delle parti, fornire la sequenza di configurazioni di M_1 quando parte avendo in input la stringa indicata.

[A]a. 11.
b. 1#1.
c. 1##1.
d. 10#11.
e. 10#10.

[A]3.3 Modificare la dimostrazione del Teorema 3.16 per ottenere il Corollario 3.19 che mostra che un linguaggio è decidibile se e solo se esiste una qualche macchina di

Turing non deterministica che lo decide. (Si può assumere il seguente teorema sugli alberi. Se ogni nodo in un albero ha un numero finito di figli e ogni cammino dell'albero ha un numero finito di nodi, allora l'albero ha un numero finito di nodi.)

3.4 Fornire una definizione formale di enumeratore. Considerare un tipo di macchina di Turing a due nastri, che usa il secondo nastro come stampante. Includere una definizione del linguaggio enumerato.

[A]3.5 Esaminare la definizione formale di macchina di Turing per rispondere alle seguenti domande, e spiegare il ragionamento fatto.

a. Può una macchina di Turing eventualmente scrivere il simbolo $\sqcup$ sul suo nastro?

b. Può l'alfabeto del nastro Γ essere lo stesso dell'alfabeto di input Σ?

c. Può una testina di una macchina di Turing trovarsi nella stessa posizione in due passi consecutivi?

d. Può una macchina di Turing avere un unico stato?

3.6 Nel Teorema 3.21, abbiamo dimostrato che un linguaggio è Turing-riconoscibile se e solo se qualche enumeratore lo enumera. Perché non usiamo il seguente algoritmo per la parte diretta della dimostrazione? Come prima, $s_1, s_2, \ldots$ è una lista di tutte le stringhe in Σ^*.

E = "Ignora l'input.

1. Ripete per $i = 1, 2, 3, \ldots$.
2. Esegue M su s_i.
3. Se M accetta, stampa s_i."

3.7 Spiegare perché la seguente non è una descrizione valida di macchina di Turing.

M_{bad} = "Su input $\langle p \rangle$, un polinomio sulle variabili $x_1, \ldots, x_k$:

1. Prova per tutte le possibili assegnazioni di valori interi a $x_1, \ldots, x_k$.
2. Valuta p per ogni assegnazione.
3. Se una di tali assegnazioni dà valore 0, *accetta*; altrimenti, *rifiuta*."

3.8 Fornire le descrizioni a livello implementativo di macchine di Turing che decidono i seguenti linguaggi sull'alfabeto {0,1}.

[A]a. $\{w \mid w$ contiene lo stesso numero di 0 e di 1$\}$

b. $\{w \mid w$ contiene un numero di 0 doppio rispetto al numero di 1$\}$

c. $\{w \mid w$ non contiene un numero di 0 doppio rispetto al numero di 1$\}$

PROBLEMI

[A]3.9 Sia A il linguaggio che contiene solo ed unicamente la stringa s,

$$s = \begin{cases} 0 & \text{se la vita non sarà mai trovata su Marte} \\ 1 & \text{se un giorno la vita sarà trovata su Marte.} \end{cases}$$

Risulta A decidibile? Giustificare la risposta. Ai fini di questo problema, assumere che la questione se la vita sarà trovata su Marte ammette una risposta non ambigua SI o NO.

3.10 Sia $c_1 x^n + c_2 x^{n-1} + \cdots + c_n x + c_{n+1}$ un polinomio con radice $x = x_0$. Sia $c_{\max}$ il massimo valore assoluto di un c_i. Mostrare che

$$|x_0| < (n+1)\frac{c_{\max}}{|c_1|}.$$

***3.11** Mostrare che le TM a nastro singolo che non possono scrivere sulla porzione di nastro che contiene la stringa di input riconoscono solo linguaggi regolari.

***3.12** Mostrare che ogni linguaggio infinito Turing-riconoscibile, ha un sottoinsieme decidibile infinito.

***3.13** Mostrare che un linguaggio è decidibile sse qualche enumeratore lo enumera nell'ordine standard sulle stringhe.

***3.14** Sia $B = \{\langle M_1 \rangle, \langle M_2 \rangle, \ldots\}$ un linguaggio Turing-riconoscibile composto da descrizioni di TM. Dimostrare che esiste un linguaggio decidibile C costituito da descrizioni di TM tale che ogni macchina descritta in B ha una macchina equivalente in C e viceversa.

3.15 Mostrare che la classe dei linguaggi Turing-riconoscibili è chiusa rispetto a

[A]a. unione.

b. concatenazione.

c. star.

d. intersezione.

e. omomorfismo.

3.16 Mostrare che la classe dei linguaggi decidibili è chiusa rispetto alle operazioni di

[A]a. unione.

b. concatenazione.

c. star.

d. complemento.

e. intersezione.

[A]3.17 Chiamiamo ***macchina di Turing a sola scrittura*** una TM a nastro singolo che può modificare ogni cella del nastro al più una volta (inclusa la parte di input del nastro). Mostrare che questa variante di macchina di Turing è equivalente alla macchina di Turing usuale. (Suggerimento: Come primo passo considerare il caso in cui la macchina di Turing può modificare ogni posizione del nastro al massimo due volte. Usa molto nastro.)

3.18 Una ***macchina di Turing a nastro doppiamente infinito*** è simile ad una comune macchina di Turing, ma il suo nastro è infinito sia a sinistra che a destra. Il nastro è inizialmente riempito con blank eccetto per la parte che contiene l'input. Il calcolo è definito come al solito, eccetto il particolare che la testina non arriva mai alla fine del nastro, quando si muove verso sinistra. Mostrare che questo tipo di macchina di Turing riconosce la classe dei linguaggi Turing-riconoscibili.

3.19 Una ***macchina di Turing con reset a sinistra*** è simile ad una comune macchina di Turing, ma la funzione di transizione ha la forma

$$\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{R}, \text{RESET}\}.$$

Se $\delta(q, a) = (r, b, \text{RESET})$, quando la macchina si trova nello stato q e legge a, la testina della macchina scrive b sul nastro, salta all'estremità sinistra del nastro ed entra nello stato r. Si noti che queste macchine non hanno la solita capacità di muovere la testina su un simbolo a sinistra. Mostrare che le macchine di Turing con reset a sinistra riconoscono la classe dei linguaggi Turing-riconoscibili.

3.20 Una ***macchina di Turing con "resta ferma" invece di "muovi a sinistra"*** è simile ad una comune macchina di Turing, ma la funzione di transizione ha la forma

$$\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{R},\mathrm{S}\}.$$

In ogni posizione la macchina può muovere la testina a destra o lasciarla ferma nella posizione corrente. Dimostrare che questa variazione di macchina di Turing *non* è equivalente alla versione comune. Quale classe di linguaggi riconoscono queste macchine?

3.21 Un ***automa a coda*** è simile ad un automa a pila con la differenza che la pila viene sostituita da una coda. Una ***coda*** è un nastro che permette di scrivere solo nella cella all'estremità sinistra del nastro e leggere solo all'estremità destra. Ogni operazione di scrittura (che chiameremo *push*) aggiunge un simbolo all'estremità sinistra della coda e ogni operazione di lettura (che chiameremo *pull*) legge e rimuove un simbolo all'estremità destra. Come con un PDA, l'input è posizionato su un nastro a sola lettura separato, e la testina sul nastro di input può muoversi solo da sinistra a destra. Il nastro di input contiene una cella con un simbolo blank che segue l'input, in modo da poter rilevare la fine dell'input. Una automa a coda accetta l'input entrando in un particolare stato di accettazione in qualsiasi momento. Mostrare che un linguaggio può essere riconosciuto da un automa determinístico a coda sse è Turing-riconoscibile.

3.22 Chiamiamo k-PDA un automa a pila che ha k pile. In particolare, uno 0-PDA è un NFA e un 1-PDA è un PDA convenzionale. Sapete già che gli 1-PDA sono più potenti (riconoscono una classe più ampia di linguaggi) rispetto agli 0-PDA.

a. Mostrare che i 2-PDA sono più potenti degli 1-PDA.

b. Mostrare che i 3-PDA non sono più potenti dei 2-PDA.
(Suggerimento: Simulare il nastro di una macchina di Turing mediante due pile.)

SOLUZIONI SELEZIONATE

3.1 **(b)** $q_1 00, \sqcup q_2 0, \sqcup \mathtt{x} q_3 \sqcup, \sqcup q_5 \mathtt{x} \sqcup, q_5 \sqcup \mathtt{x} \sqcup, \sqcup q_2 \mathtt{x} \sqcup, \sqcup \mathtt{x} q_2 \sqcup, \sqcup \mathtt{x} \sqcup q_{\text{accept}}$.

3.2 **(a)** $q_1 11, \mathtt{x} q_3 1, \mathtt{x}1 q_3 \sqcup, \mathtt{x}1 \sqcup q_{\text{reject}}$.

3.3 Dimostriamo entrambe le direzioni del sse. In primo luogo, se un linguaggio L è decidibile, esso può essere deciso da una macchina di Turing deterministica, che è automaticamente una macchina di Turing non deterministica. In secondo luogo, se un linguaggio L è deciso da una TM N non deterministica, modifichiamo la TM D deterministica descritta nella dimostrazione del Teorema 3.16 come segue.
Rendiamo la fase 4 come fase 5.
Aggiungiamo una nuova fase 4: *Rifiuta* se tutti i cammini del non determinismo di N hanno portato ad un rifiuto.
Possiamo dedurre che la nuova TM D' è un decisore per L. Se N accetta il suo input, allora D' troverà un cammino che porta all'accettazione ed accetta. Se N rifiuta il suo input, tutti i suoi cammini terminano e rifiutano perché è un decisore. Quindi ciascuno dei cammini ha un numero finito di nodi, dove ogni nodo rappresenta un passo di computazione di N, lungo quel cammino. Pertanto l'intero albero di computazione di N su questo input è finito, in virtù del teorema sugli alberi fornito nella traccia dell'esercizio. Quindi, D' si fermerà e rifiuterà quando l'intero albero sarà stato esplorato.

3.5 **(a)** Sì. L'alfabeto del nastro Γ contiene $\sqcup$. Una macchina di Turing può scrivere tutti i caratteri in Γ sul suo nastro.

(b) No. Σ non contiene $\sqcup$, ma Γ contiene sempre $\sqcup$. Quindi essi non possono essere uguali.

(c) Sì. Se la macchina di Turing tenta di muovere la testina oltre l'estremità sinistra del nastro, rimane nella posizione corrente.

(d) No. Qualsiasi macchina di Turing deve contenere due stati distinti q_{accept} e q_{reject}. Quindi, una macchina di Turing contiene almeno due stati.

3.8 **(a)** "Su input w:

1. Scorre il nastro e marca il primo 0 che non è stato marcato. Se non viene trovato alcuno 0 non marcato, passa alla Fase 4. Altrimenti, sposta la testina indietro all'inizio del nastro.
2. Esegue la scansione del nastro e segna il primo 1 che non è stato marcato. Se non viene trovato alcun 1 non marcato, *rifiuta*.
3. Sposta la testina indietro all'inizio del nastro ed esegue il passo 1.
4. Sposta la testina indietro all'inizio del nastro. Scorre il nastro per controllare se rimane qualche 1 non marcato. Se non viene trovato alcun 1 non marcato, *accetta*; altrimenti, *rifiuta*."

3.9 Il linguaggio A è uno dei due linguaggi, $\{0\}$ o $\{1\}$. In entrambi i casi il linguaggio è finito, e quindi decidibile. Se non si è in grado di determinare quale di questi due linguaggi è A, non sarà possibile descrivere un decisore per A, si possono tuttavia dare due macchine di Turing, una delle quali è un decisore per A.

3.15 Per ogni coppia di linguaggi decidibili L_1 e L_2, siano M_1 e M_2 le TM che li decidono. Costruiamo una TM M' che decide l'unione di L_1 e L_2:

"Su input w:

1. Esegue M_1 e M_2 alternativamente su w passo dopo passo. Se una accetta, *accetta*. Se entrambe si fermano e rifiutano, *rifiuta*."

Se M_1 accetta w oppure M_2 accetta w, allora M' accetta w perché la TM che accetta arriva al suo stato-accetta dopo un numero finito di passi. Si noti che se entrambe M_1 e M_2 rifiutano e una va in loop, allora anche M' andrà in loop.

3.16 Per ogni coppia di linguaggi decidibili L_1 e L_2, siano M_1 e M_2 le TM che li decidono. Costruiamo una TM M' che decide l'unione di L_1 e L_2:

"Su input w: 1. Esegue M_1 su w. Se accetta, *accetta*.
2. Esegue M_2 su w. Se accetta, *accetta*. Altrimenti, *rifiuta*."

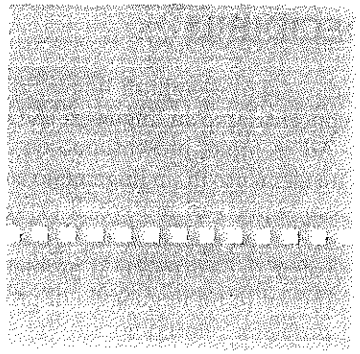
M' accetta w se M_1 accetta oppure M_2 accetta. Se entrambe rifiutano, M' rifiuta.

3.17 In primo luogo simuliamo una macchina di Turing mediante una macchina di Turing a doppia scrittura. La macchina di Turing a doppia scrittura simula un singolo passo della macchina originaria copiando l'intero nastro su una porzione nuova di nastro al lato destro della parte attualmente utilizzata. L'operazione di copia procede carattere per carattere, marcando un carattere appena esso viene copiato. Questa procedura modifica ogni cella del nastro due volte: una volta per scrivere il carattere per la prima volta e una seconda volta per marcare che è stato copiato. La posizione della testina della macchina di Turing originale è segnata sul nastro. Quando si copiano le celle corrispondenti alla posizione marcata o adiacenti a tale

posizione, il contenuto del nastro viene aggiornato in base alle regole della macchina di Turing originale.

Per effettuare la simulazione con una macchina a sola scrittura, si può operare come prima, tranne che qui ogni cella del nastro precedente viene ora rappresentata mediante due celle. La prima contiene il simbolo della macchina originaria e la seconda è dedicata alla marcatura utilizzata nella procedura di copia. L'input non è presentato alla macchina nel formato con due celle per simbolo, così la prima volta che il nastro viene copiato, i marcatori di copiatura sono messi direttamente sopra i simboli in input.

4



DECIDIBILITÀ

Nel Capitolo 3 abbiamo introdotto la macchina d Turing come un modello di calcolatore universale e definito il concetto di algoritmo in termini di macchine di Turing mediante la tesi di Church–Turing.

In questo capitolo iniziamo ad investigare la potenza degli algoritmi nella risoluzione di problemi. Dimostreremo che alcuni problemi possono essere risolti in maniera algoritmica ed altri no. Il nostro obiettivo è esplorare i limiti della risolvibilità algoritmica dei problemi. Avete probabilmente familiarità con soluzioni algoritmiche, perché gran parte dell'informatica è dedicata alla soluzione di problemi. L'insolubilità di alcuni problemi potrebbe essere una sorpresa. Perché dovremmo studiare l'insolubilità? Dopo tutto, mostrare che un problema è irrisolvibile non sembra essere di alcuna utilità se ci serve risolverlo. Vi serve studiare questo fenomeno per due motivi. In primo luogo, sapere quando un problema è irrisolvibile algoritmicamente è utile perché in tal modo vi rendete conto che il problema deve essere semplificato o modificato prima che possiate trovare una soluzione algoritmica. Come ogni strumento, i computer hanno capacità e limiti che devono essere considerati per utilizzarli al meglio. La seconda ragione è di tipo culturale. Anche se avete a che fare con problemi che sono chiaramente risolvibili, uno sguardo all'irrisolvibile può stimolare la vostra fantasia e aiutarvi ad ottenere un'importante prospettiva sulla computazione.

4.1 LINGUAGGI DECIDIBILI

In questa sezione diamo alcuni esempi di linguaggi che sono decidibili mediante algoritmi. Ci concentriamo su linguaggi che riguardano degli automi e delle grammatiche. Ad esempio, presentiamo un algoritmo che verifica se una stringa è o meno un elemento di un linguaggio context-free (CFL). Questi linguaggi sono interessanti per vari motivi. Il primo è che alcuni problemi di questo tipo sono correlati ad applicazioni. Questo problema di verificare se una CFG genera una stringa è correlato al problema del riconoscimento e compilazione dei programmi in un linguaggio di programmazione. Il secondo motivo è che altri problemi relativi ad automi e grammatiche non sono decidibili mediante algoritmi. Partendo da esempi in cui la decidibilità è possibile vi aiuta ad apprezzare esempi di problemi indecidibili.

Problemi decidibili relativi a linguaggi regolari

Iniziamo con alcuni problemi computazionali relativi ad automi finiti. Forniamo algoritmi per testare se un automa finito accetta o meno una stringa, se il linguaggio di un automa finito è vuoto e se due automi finiti sono equivalenti. Si noti che abbiamo scelto di rappresentare i problemi computazionali mediante linguaggi. Si tratta di un modo di agire conveniente perché abbiamo già acquisito la terminologia per trattare i linguaggi. Ad esempio, il ***problema dell'accettazione*** per DFA consistente nel testare se un particolare automa finito deterministico accetta una data stringa, può essere espresso come un linguaggio, A_{DFA}. Questo linguaggio contiene le codifiche di tutti i DFA con le stringhe che essi accettano. Definiamo

$$A_{\text{DFA}} = \{\langle B, w\rangle |\ B \text{ è un DFA che accetta la stringa di input } w\}.$$

Il problema di verificare se un DFA B accetta l'input w coincide con il problema di verificare se $\langle B, w\rangle$ è un elemento del linguaggio A_{DFA}. Analogamente, possiamo formulare altri problemi computazionali in termini di verifica di appartenenza ad un linguaggio. Mostrare che il linguaggio è decidibile equivale a mostrare che il problema computazionale è decidibile. Nel seguente teorema si dimostra che A_{DFA} è decidibile. Quindi il teorema mostra che il problema di verificare se un dato automa finito accetta una determinata stringa è decidibile.

TEOREMA 4.1

A_{DFA} è un linguaggio decidibile.

IDEA. Abbiamo semplicemente bisogno di presentare una TM M che decide A_{DFA}.

M = "Su input $\langle B, w\rangle$, dove B è un DFA e w è una stringa:

1. Simula B su input w.
2. Se la simulazione termina in uno stato di accettazione, *accetta*. Se non termina in uno stato di accettazione, *rifiuta*."

DIMOSTRAZIONE Citiamo solo alcuni dettagli di implementazione di questa dimostrazione. Chi ha familiarità con la scrittura di programmi, in un qualche linguaggio di programmazione standard, immagini come si potrebbe scrivere un programma per effettuare la simulazione. In primo luogo, esaminiamo l'input $\langle B, w\rangle$. Si tratta di una rappresentazione di un DFA B insieme ad una stringa w. Una rappresentazione ragionevole di B è semplicemente una lista delle sue cinque componenti: Q, Σ, δ, q_0, ed F. Quando M riceve il suo input, per prima cosa verifica se esso rappresenta correttamente un DFA B ed una stringa w. In caso contrario, M rifiuta. Poi M effettua direttamente la simulazione. Tiene traccia dello stato corrente di B e della posizione corrente di B nell'input w scrivendo queste informazioni sul suo nastro. Inizialmente, lo stato corrente di B è q_0 e la posizione corrente dell'input di B è il simbolo più a sinistra di w. Gli stati e le posizioni vengono aggiornati in base alla funzione di transizione specificata δ. Quando M termina l'elaborazione dell'ultimo simbolo di w, M accetta l'input se B è in uno stato di accettazione; M rifiuta l'input se B non è in uno stato di accettazione.

Possiamo dimostrare un teorema simile per automi a stati finiti non deterministici. Dato

$$A_{\text{NFA}} = \{\langle B, w\rangle |\ B \text{ è un NFA che accetta la stringa di input } w\}.$$

TEOREMA 4.2

A_{NFA} è un linguaggio decidibile.

DIMOSTRAZIONE. Presentiamo una TM N che decide A_{NFA}. Potremmo progettare N in modo che operi come M, simulando un NFA invece di un DFA. Invece, procederemo in modo diverso per illustrare una nuova idea: N usa M come una sottoprocedura. Poichè M è progettata per funzionare con un DFA, N prima converte l'NFA che riceve come input in un DFA e poi lo passa ad M.

N = "Su input $\langle B, w\rangle$, dove B è un NFA e w è una stringa:

1. Converte l'NFA B in un DFA C equivalente, usando la procedura di conversione data nel Teorema 1.39.
2. Esegue la TM M del Teorema 4.1 su input $\langle C, w\rangle$.
3. Se M accetta, *accetta*; altrimenti, *rifiuta*."

L'esecuzione della TM M nella fase 2 significa incorporare M nella progettazione di N come sottoprocedura.

In modo analogo, possiamo determinare se un'espressione regolare genera una data stringa. Consideriamo

$$A_{\mathsf{REX}} = \{\langle R, w\rangle \mid R \text{ è un'espressione regolare che genera la stringa } w\}.$$

TEOREMA 4.3

A_{REX} è un linguaggio decidibile.

DIMOSTRAZIONE. La seguente TM P decide A_{REX}.

P = "Su input $\langle R, w\rangle$, dove R è un'espressione regolare e w è una stringa:

1. Converte l'espressione regolare R in un NFA A equivalente, mediante la procedura di conversione data nel Teorema 1.54.
2. Esegue la TM N su input $\langle A, w\rangle$.
3. Se N accetta, *accetta*; se N rifiuta, *rifiuta*."

I Teoremi 4.1, 4.2 e 4.3 ci dicono che, ai fini della decidibilità, è equivalente presentare alla macchina di Turing un DFA, un NFA, oppure un'espressione regolare, perché la macchina è in grado di convertire una codifica nell'altra.

Ora affrontiamo un diverso tipo di problema concernente gli automi a stati finiti: il *test del vuoto* per il linguaggio di un automa finito. Nei precedenti tre teoremi dovevamo decidere se un automa finito accettasse una particolare stringa. Nella prossima dimostrazione, dobbiamo determinare se un automa finito accetta una qualche stringa. Consideriamo

$$E_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ è un DFA e } L(A) = \emptyset\}.$$

TEOREMA 4.4

E_{DFA} è un linguaggio decidibile.

DIMOSTRAZIONE. Il DFA accetta almeno una stringa se e solo se dallo stato iniziale può raggiungere uno stato di accettazione percorrendo il verso delle frecce del DFA. Per verificare questa condizione possiamo progettare una TM T che utilizza un algoritmo di marcatura analogo a quello utilizzato nell'Esempio 3.23.

T = "Su input $\langle A\rangle$, dove A è un DFA:

1. Marca lo stato iniziale di A.
2. Ripete fino a quando non vengono più marcati nuovi stati:
3. Marca qualsiasi stato che ha una transizione proveniente da uno stato già marcato.
4. Se nessuno stato di accettazione risulta marcato, *accetta*; altrimenti, *rifiuta*."

Il prossimo teorema afferma che determinare se due DFA riconoscono lo stesso linguaggio è decidibile. Sia

$$EQ_{\mathsf{DFA}} = \{\langle A, B\rangle \mid A \text{ e } B \text{ sono DFA e } L(A) = L(B)\}.$$

TEOREMA 4.5

EQ_{DFA} è un linguaggio decidibile.

DIMOSTRAZIONE. Per dimostrare questo teorema usiamo il Teorema 4.4. Costruiamo un nuovo DFA C a partire da A e B, dove C accetta solo quelle stringhe che sono accettate da A o da B, ma non da entrambi. In particolare, se A e B riconoscono lo stesso linguaggio, C non accetterà nulla. Il linguaggio di C è

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

Questa espressione è a volte denotata come ***differenza simmetrica*** di $L(A)$ e $L(B)$ ed è illustrata nella figura seguente. Qui $\overline{L(A)}$ denota il complemento di $L(A)$. La differenza simmetrica è utile perché $L(C) = \emptyset$ se e solo se $L(A) = L(B)$. Siamo in grado di costruire C da A e B con le costruzioni fatte per dimostrare che la classe dei linguaggi regolari risulta chiusa rispetto alle operazioni di complemento, unione e intersezione. Queste costruzioni sono algoritmi che possono essere eseguiti da macchine di Turing. Una volta che abbiamo costruito C possiamo usare il Teorema 4.4

per verificare se $L(C)$ è vuoto o meno. Se è vuoto, $L(A)$ e $L(B)$ devono essere uguali.

F = "Su input $\langle A, B\rangle$, dove A e B sono DFA:

1. Costruisce il DFA C come descritto.
2. Esegue la TM T del Teorema 4.4 su input $\langle C\rangle$.
3. Se T accetta, *accetta*. Se T rifiuta, *rifiuta*."

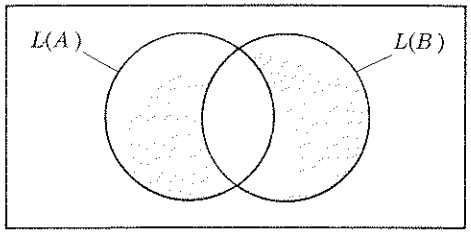


FIGURA 4.6
La differenza simmetrica di $L(A)$ e $L(B)$

Problemi decidibili relativi a linguaggi context-free

Descriviamo ora algoritmi per determinare se una CFG genera una particolare stringa e per determinare se il linguaggio di una CFG è vuoto. Consideriamo

$$A_{\mathsf{CFG}} = \{\langle G, w\rangle |\ G \text{ è una CFG che genera la stringa } w\}.$$

TEOREMA 4.7

A_{CFG} è un linguaggio decidibile.

IDEA. Per la CFG G e la stringa w vogliamo determinare se G genera w. Un'idea è quella di utilizzare G per passare attraverso tutte le derivazioni per determinare se ne esiste una di w. Quest'idea non funziona, poichè bisognerebbe provare infinite derivazioni. Se G non genera w, questo algoritmo potrebbe non fermarsi. Tale idea dà una macchina di Turing che è un riconoscitore, non un decisore, per A_{CFG}. Per rendere questa macchina di Turing un decisore occorre assicurare che l'algoritmo provi solo un numero

finito di derivazioni. Nel Problema 2.38 (page 165) abbiamo mostrato che, se G fosse in forma normale di Chomsky, qualsiasi derivazione di w avrebbe $2n - 1$ passi, dove n è la lunghezza di w. In tal caso sarebbe sufficiente controllare solo derivazioni di $2n - 1$ passi per determinare se G genera w. Tali derivazioni sono in numero finito. Possiamo convertire G in forma normale di Chomsky utilizzando la procedura data nella Sezione 2.1.

DIMOSTRAZIONE. Diamo la TM S per A_{CFG}.

S = "Su input $\langle G, w\rangle$, dove G è una CFG e w è una stringa:

1. Converte G in una grammatica equivalente in forma normale di Chomsky.
2. Lista tutte le derivazioni di $2n-1$ passi, dove n è la lunghezza di w; tranne se $n = 0$, in tal caso lista tutte le derivazioni di un passo.
3. Se una di tali derivazioni genera w, *accetta*; altrimenti, *rifiuta*."

Il problema di determinare se una CFG genera una particolare stringa è correlato al problema della compilazione dei linguaggi di programmazione. L'algoritmo nella TM S è molto inefficiente e non sarebbe mai utilizzato in pratica, ma è facile da descrivere e noi non siamo qui interessati all'efficienza. Nella Terza Parte di questo libro affrontiamo problemi riguardanti il tempo di esecuzione e l'utilizzo di memoria da parte degli algoritmi. Nella dimostrazione del Teorema 7.16, descriviamo un algoritmo più efficiente per il riconoscimento di linguaggi context-free. Ricordiamo che abbiamo dato procedure per la conversione in entrambi i sensi tra CFG e PDA nel Teorema 2.20. Quindi tutto quello che diciamo circa la decidibilità dei problemi concernenti le CFG vale anche per i PDA. Torniamo ora al problema del test del vuoto per il linguaggio di una CFG. Come abbiamo fatto per i DFA, possiamo dimostrare che il problema di determinare se una CFG genera almeno una stringa è decidibile. Sia

$$E_{\mathsf{CFG}} = \{\langle G\rangle |\ G \text{ è una CFG ed } L(G) = \emptyset\}.$$

TEOREMA 4.8

E_{CFG} è un linguaggio decidibile.

IDEA. Per progettare un algoritmo per questo problema, potremmo tentare di utilizzare la TM S del Teorema 4.7. Tale teorema afferma che siamo in grado di verificare se una CFG genera una particolare stringa w. Per

determinare se $L(G) = \emptyset$, l'algoritmo potrebbe tentare di passare attraverso tutte le possibili w, una per una. Però esistono infinite w da testare, quindi questo metodo potrebbe far sì che non termini mai. Abbiamo bisogno di trovare un approccio differente. Per determinare se il linguaggio di una grammatica è vuoto, abbiamo bisogno di testare se la variabile iniziale può generare una stringa di terminali. L'algoritmo fa questo risolvendo un problema più generale. Determina *per ogni variabile* se essa è in grado di generare una stringa di terminali. Quando l'algoritmo ha determinato che una variabile può generare qualche stringa di terminali, l'algoritmo tiene traccia di queste informazioni marcando tale variabile. Dapprima, l'algoritmo marca tutti i simboli terminali della grammatica. Poi, scandisce tutte le regole della grammatica. Se trova una regola che consente a qualche variabile di essere sostituita da una stringa di simboli, che sono già tutti marcati, l'algoritmo sa che anche questa variabile può essere marcata. L'algoritmo continua in questo modo finché non può marcare ulteriori variabili. La TM R implementa questo algoritmo.

DIMOSTRAZIONE.

R = "Su input $\langle G \rangle$, dove G è una CFG:

1. Marca tutti i simboli terminali in G.
2. Ripete fin quando nessuna nuova variabile viene marcata:
3. Marca una qualsiasi variabile A tale che G ha una regola $A \rightarrow U_1 U_2 \cdots U_k$ ed ogni simbolo $U_1, \ldots, U_k$ è gia stato marcato.
4. Se la variabile iniziale non è segnata, *accetta*; altrimenti, *rifiuta*."

Consideriamo ora il problema di determinare se due grammatiche context-free generano lo stesso linguaggio. Sia

$$EQ_{\mathsf{CFG}} = \{\langle G, H \rangle \mid G \text{ e } H \text{ sono CFG e } L(G) = L(H)\}.$$

Il Teorema 4.5 fornisce un algoritmo che decide l'analogo linguaggio EQ_{DFA} per automi finiti. Abbiamo utilizzato la procedura di decisione per E_{DFA} per dimostrare che EQ_{DFA} è decidibile. Poichè E_{CFG} è anch'esso decidibile, si potrebbe pensare che possiamo usare una strategia simile per dimostrare che EQ_{CFG} è decidibile. Ma qualcosa non va con questa idea! La classe dei linguaggi context-free *non* è chiusa per complemento o intersezione, come si è dimostrato nell'Esercizio 2.2. Infatti, EQ_{CFG} non è decidibile. Una dimostrazione di ciò viene presentata nel Capitolo 5. Ora mostriamo che i linguaggi context-free possono essere decisi dalle macchine di Turing.

TEOREMA 4.9

Ogni linguaggio context-free è decidibile.

IDEA. Sia A un CFL. Il nostro obiettivo è mostrare che A è decidibile. Una (cattiva) idea è quella di convertire un PDA per A direttamente in una TM. Ciò non è difficile da realizzare, perché simulare una pila con il nastro di una TM è semplice. Il PDA per A può essere non deterministico, ma ciò sembra andare bene, perché siamo in grado di convertirlo in una TM non deterministica e sappiamo che qualsiasi TM non deterministica può essere convertita in una TM deterministica equivalente. C'è tuttavia una difficoltà. Alcuni rami della computazione del PDA possono andare avanti per sempre, leggendo e scrivendo la pila senza mai arrestarsi. La TM che effettua la simulazione avrebbe quindi alcuni cammini che non terminano mai, quindi la TM non sarebbe un decisore.
È necessaria una diversa soluzione. Dimostriamo il teorema con la TM S che abbiamo progettato nel Teorema 4.7 per decidere A_{CFG}.

DIMOSTRAZIONE. Sia G una CFG per A, progettiamo una TM M_G che decide A. Costruiamo una copia di G in M_G. Essa funziona come segue.

M_G = "Su input w:

1. Esegue la TM S su input $\langle G, w \rangle$.
2. Se questa macchina accetta, *accetta*; se essa rifiuta, *rifiuta*."

Il Teorema 4.9 fornisce il collegamento finale nelle relazioni tra le quattro principali classi di linguaggi che abbiamo descritto finora: regolari, context-free, decidibili e Turing-riconoscibili. La figura 4.10 mostra tali relazioni.

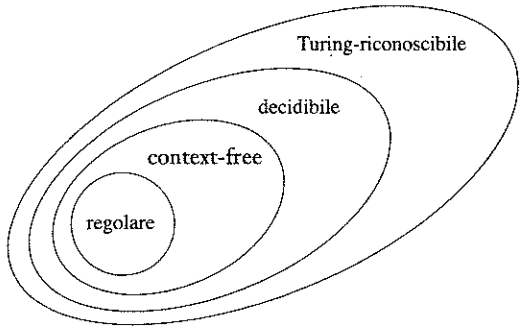


FIGURA 4.10
Le relazioni tra le classi di linguaggi

4.2 INDECIDIBILITÀ

In questa sezione dimostriamo uno dei teoremi più importanti della teoria della computazione: esiste un problema specifico che è algoritmicamente irrisolvibile. I computer sembrano essere così potenti da farvi credere che possano essere utilizzati per risolvere qualsiasi problema. Il teorema che presentatiamo qui dimostra che i computer hanno delle limitazioni intrinseche. Che tipo di problemi non sono risolvibili con un computer? Sono problemi esoterici, esistenti soltanto nella mente dei teorici? No! Anche alcuni problemi comuni che le persone vogliono risolvere si rivelano computazionalmente irrisolvibili.

In un tipo di problema irrisolvibile, sono dati un programma per computer ed una descrizione precisa di ciò che il programma dovrebbe fare (per esempio, ordinare una lista di numeri). Vi si chiede di verificare che il programma esegua quanto specificato (cioè, che sia corretto). Poichè sia il programma che le specifiche sono oggetti matematicamente esatti, voi sperate di essere capaci di automatizzare il processo di verifica mediante un computer opportunamente programmato. Tuttavia, rimarrete delusi. Il problema generale di verifica del software non è risolvibile utilizzando un computer.

In questa sezione e nel Capitolo 5, incontrerete vari problemi computazionalmente non risolvibili. L'obiettivo è quello di aiutarvi a sviluppare una capacità di riconoscere un problema irrisolvibile e di insegnarvi alcune tecniche per dimostrarne l'insolubilità.

Ora affronteremo il nostro primo teorema che stabilisce l'indecidibilità di uno specifico linguaggio: il problema di determinare se una macchina di Turing accetta una determinata stringa in input. Chiamiamo tale linguaggio A_{TM} per analogia con A_{DFA} e A_{CFG}. Tuttavia, mentre A_{DFA} e A_{CFG} sono decidibili, A_{TM} non lo è. Sia

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ è una TM e } M \text{ accetta } w\}.$$

TEOREMA 4.11

A_{TM} è indecidibile.

Prima di vedere la dimostrazione, osserviamo che A_{TM} è Turing-riconoscibile. Quindi questo teorema mostra che i riconoscitori *sono* più potenti dei decisori. Richiedere che una TM si fermi su ogni input limita le tipologie di linguaggi che possono essere riconosciuti. La seguente macchina di Turing U riconosce A_{TM}.

U = "Su input $\langle M, w\rangle$, dove M è una TM e w è una stringa:

1. Simula M su input w.
2. Se durante la computazione M entra nello stato di accettazione, *accetta*; se M entra nello stato di rifiuto, *rifiuta*."

Si noti che questa macchina cicla su input $\langle M, w\rangle$ se M cicla su w, questo è il motivo per cui non decide A_{TM}. Se l'algoritmo avesse modo di determinare che M non si ferma su w, sarebbe in tal caso in grado di rifiutare w. Come dimostreremo, un algoritmo non ha modo di determinarlo.

La macchina di Turing U è, di per sè, interessante. È un esempio della prima *macchina universale di Turing* introdotta da Alan Turing nel 1936. Questa macchina è chiamata universale perché è in grado di simulare qualsiasi altra macchina di Turing a partire dalla descrizione di tale macchina. La macchina di Turing universale ha avuto un ruolo importante nello stimolare lo sviluppo di computer a programma memorizzato.

Il metodo della diagonalizzazione

La dimostrazione dell'indecidibilità di A_{TM} utilizza una tecnica chiamata *diagonalizzazione*, scoperta dal matematico Georg Cantor nel 1873. Cantor stava studiando il problema di misurare le dimensioni degli insiemi infiniti. Se abbiamo due insiemi infiniti, come possiamo dire se uno è più grande dell'altro o se hanno la stessa dimensione? Per insiemi finiti, ovviamente, rispondere a queste domande è facile. Dobbiamo semplicemente contare gli elementi di un insieme finito ed il numero risultante è la sua dimensione. Ma, se provassimo a contare gli elementi di un insieme infinito, non riusciremmo mai a terminare! Quindi non possiamo utilizzare il metodo del conteggio per determinare le dimensioni relative agli insiemi infiniti. Per esempio, prendiamo l'insieme degli interi pari e l'insieme di tutte le stringhe su {0,1}. Entrambi gli insiemi sono infiniti e quindi più grandi di qualsiasi insieme finito, ma uno dei due è più grande rispetto all'altro? Come possiamo confrontarne le dimensioni relative? Cantor propose una soluzione interessante a questo problema. Egli osservò che due insiemi finiti hanno la stessa dimensione se gli elementi di un insieme possono essere accoppiati agli elementi dell'altro gruppo. Questo metodo confronta le dimensioni senza ricorrere al conteggio. Possiamo estendere questa idea agli insiemi infiniti. Ora lo vediamo in maniera più precisa.

DEFINIZIONE **4.12**

Supponiamo di avere gli insiemi A e B ed una funzione f da A in B. Diciamo che f è ***iniettiva***, se essa non mappa mai due elementi diversi in uno stesso punto – cioè se $f(a) \neq f(b)$ ogniqualvolta $a \neq b$. Diciamo che f è ***suriettiva*** se tocca ogni elemento di B – cioè se per ogni $b \in B$ esiste un $a \in A$ tale che $f(a) = b$. Diciamo che A e B hanno la ***stessa cardinalità*** se esiste una funzione iniettiva e suriettiva $f: A \longrightarrow B$. Una funzione che è sia iniettiva che suriettiva è detta funzione ***biettiva*** (o biezione). In una funzione biettiva ogni elemento di A viene mappato in un unico elemento di B e per ogni elemento di B esiste un unico elemento di A che viene mappato in esso. Una biezione è un modo semplice per accoppiare elementi di A con elementi di B.

ESEMPIO 4.13

Sia $\mathcal{N}$ l'insieme dei numeri naturali $\{1, 2, 3, \ldots\}$ e sia $\mathcal{E}$ l'insieme dei numeri naturali pari $\{2, 4, 6, \ldots\}$. Utilizzando la definizione di cardinalità di Cantor possiamo vedere che $\mathcal{N}$ e $\mathcal{E}$ hanno la stessa cardinalità. La funzione f che crea una corrispondenza tra $\mathcal{N}$ e $\mathcal{E}$ è semplicemente $f(n) = 2n$. Possiamo visualizzare f più facilmente con l'aiuto di una tabella.

n	$f(n)$
1	2
2	4
3	6
$\vdots$	$\vdots$

Naturalmente, questo esempio sembra strano. Intuitivamente, $\mathcal{E}$ sembra più piccolo di $\mathcal{N}$ in quanto $\mathcal{E}$ è un sottoinsieme proprio di $\mathcal{N}$. Ma è possibile accoppiare ogni elemento di $\mathcal{N}$ con un elemento di $\mathcal{E}$, quindi dichiariamo che questi due insiemi hanno la stessa cardinalità.

DEFINIZIONE **4.14**

Un insieme A è ***numerabile*** se è finito oppure ha la stessa cardinalità di $\mathcal{N}$.

ESEMPIO 4.15

Passiamo ora ad un esempio ancora più strano. Se consideriamo l'insieme $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$ dei numeri razionali positivi, $\mathcal{Q}$ sembra essere molto più grande di $\mathcal{N}$. Tuttavia, questi due insiemi hanno la stessa cardinalità, secondo la nostra definizione. Forniamo una biezione con $\mathcal{N}$ per mostrare che $\mathcal{Q}$ è numerabile. Un modo semplice per farlo è quello di elencare tutti gli elementi di Q. Per farlo accoppiamo il primo elemento della lista con il numero 1 di $\mathcal{N}$ il secondo elemento della lista con il numero 2 di $\mathcal{N}$ e così via. Dobbiamo garantire che ogni elemento di $\mathcal{Q}$ sia presente una sola volta nella lista.

Per ottenere questa lista creiamo una matrice infinita contenente tutti i numeri razionali positivi, come mostrato nella Figura 4.16. La riga i-esima contiene tutti i numeri con numeratore i e la colonna j-esima ha tutti i numeri con denominatore j. In tal modo il numero $\frac{i}{j}$ occupa la i-esima riga e la j-esima colonna. Ora trasformiamo questa matrice in una lista. Un modo (errato) di farlo potrebbe essere quello di cominciare la lista con tutti gli elementi della prima riga. Questo non è un buon approccio perché la prima riga è infinita, quindi non raggiungeremmo mai la seconda riga. Elencheremo invece gli elementi sulle diagonali, sovrapposte al diagramma, a partire da un angolo. La prima diagonale include il singolo elemento $\frac{1}{1}$, e la seconda diagonale include i due elementi $\frac{2}{1}$ e $\frac{1}{2}$. Così i primi tre elementi della lista sono $\frac{1}{1}$, $\frac{2}{1}$ e $\frac{1}{2}$. Nella terza diagonale sorge una complicazione. Essa contiene $\frac{3}{1}$, $\frac{2}{2}$, e $\frac{1}{3}$. Se li aggiungessimo così come sono alla lista, ripeteremmo $\frac{1}{1} = \frac{2}{2}$. Risolviamo il problema omettendo un elemento quando esso dà origine ad una ripetizione. Così aggiungiamo solo i due nuovi elementi $\frac{3}{1}$ e $\frac{1}{3}$. Continuando in questo modo si ottiene una lista di tutti gli elementi di $\mathcal{Q}$.

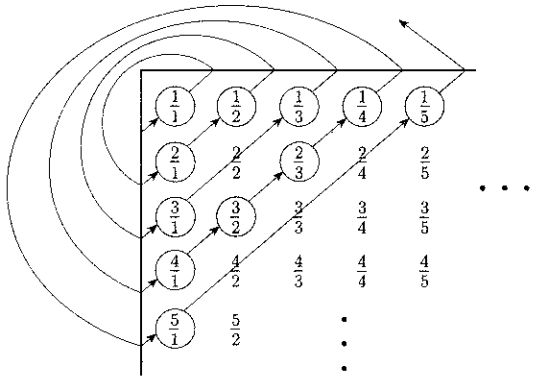


FIGURA 4.16
Una biezione da $\mathcal{N}$ a $\mathcal{Q}$.

Dopo aver visto la biezione da $\mathcal{N}$ a $\mathcal{Q}$, si potrebbe pensare di dimostrare che presi qualsiasi coppia di insiemi infiniti, questi hanno la stessa dimensione. Dopo tutto, dovete solo mostrare una biezione, e questo esempio mostra che esistono biezioni sorprendenti. Tuttavia, per alcuni insiemi infiniti non c'è alcuna biezione con $\mathcal{N}$. Questi insiemi sono semplicemente troppo grandi. Un tale insieme è detto ***non numerabile***.

L'insieme dei numeri reali è un esempio di insieme non numerabile. Un ***numero reale*** è un numero che ha una rappresentazione decimale. I numeri $\pi = 3.1415926\ldots$ e $\sqrt{2} = 1.4142135\ldots$ sono esempi di numeri reali. Sia $\mathcal{R}$ l'insieme dei numeri reali. Cantor dimostrò che $\mathcal{R}$ non è numerabile. Per far ciò introdusse il metodo della diagonalizzazione.

TEOREMA 4.17

$\mathcal{R}$ è non numerabile.

DIMOSTRAZIONE. Per dimostrare che $\mathcal{R}$ è non numerabile, mostriamo che non esiste una biezione tra $\mathcal{N}$ e $\mathcal{R}$. La dimostrazione è per assurdo. Supponiamo quindi che esista una biezione f tra $\mathcal{N}$ ed $\mathcal{R}$. Il nostro compito è quello di dimostrare che f non funziona come dovrebbe. Per essere una biezione, f deve associare tutti gli elementi di $\mathcal{N}$ con tutti gli elementi di $\mathcal{R}$. Tuttavia troveremo un numero x in $\mathcal{R}$ che non è accoppiato con alcun elemento in $\mathcal{N}$, il che rappresenterà la contraddizione cercata. Un metodo per trovare questo numero x è quello di costruirlo. Scegliamo ogni cifra di x in modo da renderlo differente da ogni numero reale accoppiato con un elemento di $\mathcal{N}$. Alla fine saremo sicuri che x risulta diverso da ogni numero reale accoppiato con un elemento di $\mathcal{N}$. Possiamo illustrare questa idea con un esempio. Supponiamo che la biezione f esista. Prendiamo $f(1) = 3.14159\ldots$, $f(2) = 55.55555\ldots$, $f(3) = \ldots$, e così via, tanto per dare alcuni valori per f. Allora f accoppia il numero 1 con $3.14159\ldots$, il numero 2 con $55.55555\ldots$,, e così via. La tabella seguente mostra alcuni valori di un'ipotetica biezione f tra $\mathcal{N}$ e $\mathcal{R}$.

n	$f(n)$
1	$3.14159\ldots$
2	$55.55555\ldots$
3	$0.12345\ldots$
4	$0.50000\ldots$
$\vdots$	$\vdots$

Costruiamo il numero x desiderato, dandone la rappresentazione decimale. Si tratta di un numero compreso tra 0 e 1, quindi tutte le sue cifre significative sono cifre frazionarie che seguono la virgola decimale. Il nostro obiettivo è quello di garantire che $x \neq f(n)$ per ogni n. Per garantire $x \neq f(1)$,

scegliamo la prima cifra di x come una qualsiasi cifra diversa dalla prima cifra decimale 1 di $f(1) = 3.\underline{1}4159\ldots$. Arbitrariamente, scegliamola pari a 4. Per assicurarci che $x \neq f(2)$, scegliamo la seconda cifra di x diversa dalla seconda cifra decimale 5 di $f(2) = 55.\underline{5}5555\ldots$. Arbitrariamente, scegliamola pari a 6. La terza cifra frazionaria di $f(3) = 0.12\underline{3}45\ldots$ è 3, quindi facciamo in modo che in x sia differente –, diciamo sia 4. Continuando in questo modo lungo la diagonale della tavola per f, otteniamo tutte le cifre di x, come mostrato nella tabella seguente. Sappiamo che x non è $f(n)$ per ogni n perché differisce da $f(n)$ nell'n-esima cifra frazionaria. (Un piccolo problema nasce dal fatto che alcuni numeri, come $0.1999\ldots$ e $0.2000\ldots$, sono uguali, anche se le loro rappresentazioni decimali sono diverse. Evitiamo questo problema semplicemente non selezionando mai le cifre 0 or 9 quando costruiamo x.)

n	$f(n)$
1	$3.\underline{1}4159\ldots$
2	$55.\underline{5}5555\ldots$
3	$0.12\underline{3}45\ldots$
4	$0.500\underline{0}0\ldots$
$\vdots$	$\vdots$

$x = 0.4641\ldots$

Il teorema precedente ha un'importante applicazione nella teoria della computazione. Esso dimostra che alcuni linguaggi non sono decidibili e neppure Turing riconoscibili, per la ragione che l'insieme dei linguaggi è non numerabile mentre l'insieme di tutte le macchine di Turing è numerabile. Poichè ogni macchina di Turing è in grado di riconoscere un solo linguaggio e ci sono più linguaggi che macchine di Turing, alcuni linguaggi non sono riconosciuti da una qualche macchina di Turing. Tali linguaggi non sono Turing-riconoscibili, come enunciato nel seguente corollario.

COROLLARIO 4.18

Alcuni linguaggi non sono Turing-riconoscibili.

DIMOSTRAZIONE. Per dimostrare che l'insieme di tutte le macchine di Turing è numerabile dobbiamo prima osservare che l'insieme di tutte le stringhe Σ^* è numerabile, per ogni alfabeto Σ. Avendo solo un numero finito di stringhe di ogni lunghezza, possiamo formare una lista di Σ^* scrivendo tutte le stringhe di lunghezza 0, lunghezza 1, lunghezza 2, e così via. L'insieme di tutte le macchine di Turing è numerabile perché ogni macchina di Turing M può essere codificata con una stringa $\langle M \rangle$. Se ci limitiamo a tralasciare quelle stringhe che non sono la codifica di una macchina di Turing, possiamo ottenere una lista di tutte le macchine di Turing. Per

mostrare che l'insieme di tutti i linguaggi è non numerabile, dobbiamo prima osservare che l'insieme delle sequenze binarie infinite è non-numerabile. Una *sequenza binaria infinita* è una sequenza senza fine di 0 e 1. Sia $\mathcal{B}$ l'insieme di tutte sequenze binarie infinite. Possiamo dimostrare che $\mathcal{B}$ è non numerabile utilizzando una dimostrazione mediante diagonalizzazione simile a quella che abbiamo utilizzato nel Teorema 4.17 per mostrare la non numerabilità di $\mathcal{R}$. Sia $\mathcal{L}$ l'insieme di tutti i linguaggi sull'alfabeto Σ. Mostriamo che $\mathcal{L}$ è non-numerabile dando una corrispondenza con $\mathcal{B}$, dimostrando così che i due insiemi hanno la stessa cardinalità. Sia $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Ogni linguaggio $A \in \mathcal{L}$ ha un'unica sequenza in $\mathcal{B}$. Il bit i-esimo della sequenza è un 1 se $s_i \in A$ ed è uno 0 se $s_i \notin A$; questa è chiamata la ***sequenza caratteristica*** di A. Per esempio, se A fosse il linguaggio di tutte le stringhe che iniziano con uno 0 sull'alfabeto $\{0,1\}$, la sua sequenza caratteristica χ_A sarebbe

$$\begin{aligned}
\Sigma^* &= \{\ \varepsilon,\ \ 0,\ \ 1,\ \ 00,\ 01,\ 10,\ 11,\ 000, 001,\ \cdots\ \};\\
A &= \{\ \ \ \ 0,\ \ \ \ \ 00,\ 01,\ \ \ \ \ \ \ 000, 001,\ \cdots\ \};\\
\chi_A &= \ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1\ \ \cdots\ \ .
\end{aligned}$$

La funzione $f\colon \mathcal{L} \longrightarrow \mathcal{B}$, dove $f(A)$ è uguale alla sequenza caratteristica di A, è sia iniettiva che suriettiva, quindi è una biezione. Pertanto, essendo $\mathcal{B}$ non numerabile, anche $\mathcal{L}$ è non numerabile.

Abbiamo così dimostrato che l'insieme di tutti i linguaggi non può essere messo in corrispondenza biunivoca con l'insieme di tutte le macchine di Turing. Concludiamo quindi che alcuni linguaggi non sono riconosciuti da una macchina di Turing

Un linguaggio indecidibile

Siamo ora pronti a dimostrare il Teorema 4.11, l'indecidibilità del linguaggio

$$A_{\mathsf{TM}} = \{\langle M, w\rangle |\ M \text{ è una TM ed } M \text{ accetta } w\}.$$

DIMOSTRAZIONE. Assumiamo che A_{TM} è decidibile, per poi ottenere una contraddizione. Supponiamo che H sia un decisore per A_{TM}. Sull'input $\langle M, w\rangle$, dove M è una TM e w è una stringa, H si ferma ed accetta se M accetta w. Inoltre, H si ferma e rifiuta se M non accetta w. In altri termini, assumiamo che H è una TM, dove

$$H\big(\langle M, w\rangle\big) = \begin{cases} \textit{accetta} & \text{se } M \text{ accetta } w \\ \textit{rifiuta} & \text{se } M \text{ non accetta } w. \end{cases}$$

Ora costruiamo una nuova macchina di Turing D avente H come sottoprocedura. Questa nuova TM chiama H per determinare cosa fa M quando

l'input di M è la sua stessa descrizione $\langle M\rangle$. Una volta che D ha determinato questa informazione, essa fa il contrario. Cioè, rifiuta se M accetta ed accetta se M non accetta. Diamo ora una descrizione di D.

D = "Su input $\langle M\rangle$, dove M è una TM:

1. Esegue H su input $\langle M, \langle M\rangle\rangle$.
2. Dà in output l'opposto di ciò che H dà in output. Cioè, se H accetta, *rifiuta*; e se H rifiuta, *accetta*."

Non lasciatevi confondere dall'idea di attivare una macchina sulla sua stessa descrizione! Ciò è simile all'esecuzione di un programma che chiama se stesso in input, qualcosa che si fa a volte nella pratica. Ad esempio, un compilatore è un programma che traduce altri programmi. Un compilatore per il linguaggio Python può essere esso stesso scritto in Python, quindi eseguire tale programma su se stesso avrebbe senso. Ricapitolando,

$$D\big(\langle M\rangle\big) = \begin{cases} \textit{accetta} & \text{se } M \text{ non accetta } \langle M\rangle \\ \textit{rifiuta} & \text{se } M \text{ accetta } \langle M\rangle. \end{cases}$$

Cosa succede quando eseguiamo D con la sua stessa descrizione $\langle D\rangle$ in input? In tal caso, otteniamo

$$D\big(\langle D\rangle\big) = \begin{cases} \textit{accetta} & \text{se } D \text{ non accetta } \langle D\rangle \\ \textit{rifiuta} & \text{se } D \text{ accetta } \langle D\rangle. \end{cases}$$

Indipendentemente da ciò che D fa, essa è costretta a fare il contrario, il che è ovviamente una contraddizione. Quindi, nè la TM D nè la TM H possono esistere.

Rivediamo i passi di questa prova. Supponiamo che la TM H decida A_{TM}. Quindi usiamo H per costruire una TM D che prende in input $\langle M\rangle$, tale che D accetta $\langle M\rangle$ esattamente quando M non accetta il suo input $\langle M\rangle$. Infine, eseguiamo D su se stessa. Quindi, le macchine eseguono le seguenti azioni, dove l'ultima riga fornisce la contraddizione.

- H accetta $\langle M, w\rangle$ esattamente quando M accetta w.
- D rifiuta $\langle M\rangle$ esattamente quando M accetta $\langle M\rangle$.
- D rifiuta $\langle D\rangle$ esattamente quando D accetta $\langle D\rangle$.

Dove si usa la diagonalizzazione nella dimostrazione del Teorema 4.11? Essa diviene evidente quando si esaminano le tavole del comportamento delle TM H e D. In queste tavole indicizziamo le righe con tutte le TM $M_1, M_2, \ldots,$ e indicizziamo le colonne con le descrizioni $\langle M_1\rangle, \langle M_2\rangle, \ldots$ di tali macchine. Le entrate dicono se la macchina di una determinata riga accetta l'input

di una data colonna. L'entrata è *accetta* se la macchina accetta l'input, ma è vuota, se rifiuta o entra in loop su quell'input. Nella seguente figura abbiamo creato alcune voci per illustrare l'idea.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accetta*		*accetta*		
M_2	*accetta*	*accetta*	*accetta*	*accetta*	
M_3					$\cdots$
M_4	*accetta*	*accetta*			
$\vdots$			$\vdots$		

FIGURA 4.19
L'entrata i, j è *accetta* se M_i accetta $\langle M_j \rangle$

Nella figura seguente le voci sono i risultati dell'esecuzione di H sugli input corrispondenti alla Figura 4.19. Quindi, se M_3 non accetta l'input $\langle M_2 \rangle$, l'entrata per la riga M_3 e la colonna $\langle M_2 \rangle$ è *rifiuta* perché H rifiuta gli input $\langle M_3, \langle M_2 \rangle \rangle$.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accetta*	*rifiuta*	*accetta*	*rifiuta*	
M_2	*accetta*	*accetta*	*accetta*	*accetta*	
M_3	*rifiuta*	*rifiuta*	*rifiuta*	*rifiuta*	$\cdots$
M_4	*accetta*	*accetta*	*rifiuta*	*rifiuta*	
$\vdots$			$\vdots$		

FIGURA 4.20
L'entrata i, j è il valore di H su input $\langle M_i, \langle M_j \rangle \rangle$

Nella figura seguente, abbiamo aggiunto D alla Figura 4.20. Secondo la nostra assunzione, H è una TM così come D. Quindi deve comparire nella lista M_1, M_2, ... di tutte le TM. Si noti che D calcola il contrario degli elementi sulla diagonale. La contraddizione si ottiene in corrispondenza del punto interrogativo, dove l'entrata dovrebbe essere l'opposto di se stessa.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accetta*	*rifiuta*	*accetta*	*rifiuta*		*accetta*	
M_2	*accetta*	*accetta*	*accetta*	*accetta*	$\cdots$	*accetta*	$\cdots$
M_3	*rifiuta*	*rifiuta*	*rifiuta*	*rifiuta*		*rifiuta*	
M_4	*accetta*	*accetta*	*rifiuta*	*rifiuta*		*accetta*	
$\vdots$			$\vdots$		$\ddots$		
D	*rifiuta*	*rifiuta*	*accetta*	*accetta*		?	
$\vdots$			$\vdots$				$\ddots$

FIGURA 4.21
Se D compare nella figura, si ha una contraddizione in corrispondenza di "?"

Un linguaggio non Turing-riconoscibile

Nella sezione precedente abbiamo dimostrato che un linguaggio – precisamente, A_{TM}– è non decidibile. Ora mostriamo un linguaggio che non è neppure Turing-riconoscibile. Si noti che A_{TM} non basta allo scopo, perché abbiamo dimostrato che A_{TM} è Turing-riconoscibile (pagina 212). Il teorema seguente mostra che, se un linguaggio ed il suo complemento sono entrambi Turing-riconoscibili, il linguaggio è decidibile. Quindi, per qualsiasi linguaggio non decidibile, o esso non è Turing-riconoscibile oppure il suo complemento non è Turing-riconoscibile. Ricordiamo che il complemento di un linguaggio è il linguaggio costituito da tutte le stringhe che non sono nel linguaggio. Diciamo che un linguaggio è ***coTuring riconoscibile*** se esso è il complemento di un linguaggio Turing-riconoscibile.

TEOREMA 4.22

Un linguaggio è decidibile se e solo se è Turing-riconoscibile e coTuring-riconoscibile.

In altre parole, un linguaggio è decidibile esattamente quando sia esso che il suo complemento sono Turing-riconoscibili.

DIMOSTRAZIONE. Abbiamo due direzioni da dimostrare. In primo luogo, se A è decidibile, possiamo facilmente vedere che sia A che il suo complemento $\overline{A}$ sono Turing-riconoscibili. Qualsiasi linguaggio decidibile è Turing-riconoscibile, e il complemento di un linguaggio decidibile è a sua volta decidibile. Per la direzione inversa, se entrambi A e $\overline{A}$ sono Turing-riconoscibili, indichiamo con M_1 il riconoscitore per A e con M_2 il riconoscitore per $\overline{A}$. La seguente macchina di Turing M è un decisore per A.

M = "Su input w:

1. Esegue sia M_1 che M_2 su input w in parallelo.
2. Se M_1 accetta, *accetta*; se M_2 accetta, *rifiuta*."

L'esecuzione di due macchine in parallelo significa che M ha due nastri, uno per simulare M_1 e l'altro per simulare M_2. In questo caso M alterna la simulazione di un passo di M_1 con un passo di M_2 e continua finché una delle due accetta. Ora dimostriamo che M decide A. Ogni stringa w è in A, oppure in $\overline{A}$. Pertanto una tra M_1 ed M_2 deve accettare w. Poichè M si ferma ogni volta che M_1 accetta oppure M_2 accetta, allora M si ferma sempre quindi è un decisore. Inoltre, accetta tutte le stringhe in A e respinge tutte le stringhe che non sono in A. Quindi M è un decisore per A, e pertanto A è decidibile.

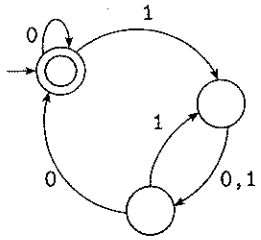
COROLLARIO 4.23

$\overline{A_{\mathsf{TM}}}$ non è Turing-reconoscibile.

DIMOSTRAZIONE. Sappiamo che A_{TM} è Turing-riconoscibile. Se anche $\overline{A_{\mathsf{TM}}}$ fosse Turing-riconoscibile, A_{TM} sarebbe decidibile. Il Teorema 4.11 ci dice che A_{TM} non è decidibile, quindi $\overline{A_{\mathsf{TM}}}$ non può essere Turing-riconoscibile.

ESERCIZI

[A]**4.1** Rispondere a tutte le domande che riguardano il seguente DFA M e giustificare le risposte.



a. Risulta $\langle M, 0100\rangle \in A_{\mathsf{DFA}}$?
b. Risulta $\langle M, 011\rangle \in A_{\mathsf{DFA}}$?
c. Risulta $\langle M\rangle \in A_{\mathsf{DFA}}$?
d. Risulta $\langle M, 0100\rangle \in A_{\mathsf{REX}}$?
e. Risulta $\langle M\rangle \in E_{\mathsf{DFA}}$?
f. Risulta $\langle M, M\rangle \in EQ_{\mathsf{DFA}}$?

4.2 Considerare il problema di determinare se un DFA ed un'espressione regolare sono equivalenti. Esprimere questo problema come un linguaggio e dimostrare che è decidibile.

4.3 Sia $ALL_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ è un DFA e } L(A) = \Sigma^*\}$. Mostrare che ALL_{DFA} è decidibile.

4.4 Sia $A\varepsilon_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ è una CFG che genera } \varepsilon\}$. Mostrare che $A\varepsilon_{\mathsf{CFG}}$ è decidibile.

[A]**4.5** Sia $E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ è una TM e } L(M) = \emptyset\}$. Mostrare che $\overline{E_{\mathsf{TM}}}$, il complemento di E_{TM}, è Turing-riconoscibile

4.6 Siano X l'insieme $\{1,2,3,4,5\}$ e Y l'insieme $\{6,7,8,9,10\}$. Le funzioni $f\colon X \longrightarrow Y$ e $g\colon X \longrightarrow Y$ sono descritte nelle tabelle seguenti. Rispondere ad ogni domanda e fornire una giustificazione per le risposte negative.

n	$f(n)$
1	6
2	7
3	6
4	7
5	6

n	$g(n)$
1	10
2	9
3	8
4	7
5	6

[A]**a.** La funzione f è iniettiva?
b. La funzione f è suriettiva?
c. La funzione f è biettiva?
[A]**d.** La funzione g è iniettiva?
e. La funzione g è suriettiva?
f. La funzione g è biettiva?

4.7 Sia $\mathcal{B}$ l'insieme di tutte le sequenze infinite su $\{0,1\}$. Utilizzare il metodo della diagonalizzazione per mostrare che $\mathcal{B}$ non è numerabile.

4.8 Sia $T = \{(i,j,k) \mid i,j,k \in \mathcal{N}\}$. Mostrare che T è numerabile.

4.9 Riguardare la definizione di insiemi aventi la stessa cardinalità nella Definizione 4.12 (pagina 214). Mostrare che "ha la stessa cardinalità" è una relazione di equivalenza.

PROBLEMI

4.10 La dimostrazione del Lemma 2.41 dice che (q,x) è una *situazione di loop* per un DPDA P se quando P parte nello stato q con $x \in \Gamma$ al top della pila, esso non farà mai il pop di un elemento al di sotto di x e non leggerà mai un simbolo di input. Mostrare che F è decidibile, dove $F = \{\langle P,q,x\rangle \mid (q,x) \text{ è una situazione di loop } P\}$.

4.11 Diciamo che una variabile A in un CFL G è ***utilizzabile*** se compare in una derivazione di una qualche stringa $w \in G$. Data una CFG G ed una variabile A, si consideri il problema di verificare se A è utilizzabile. Formulare tale problema come un linguaggio e mostrare che è decidibile.

4.12 Sia A un linguaggio Turing-riconoscibile consistente in descrizioni di macchine di Turing, $\{\langle M_1\rangle, \langle M_2\rangle, \ldots\}$, dove ogni M_i è un decisore. Mostrare che esiste un linguaggio decidibile D che non è deciso da un decisore M_i la cui descrizione appare in A. (Suggerimento: Potete trovare utile considerare un enumeratore per A.)

4.13 Sia $C_{\mathsf{CFG}} = \{\langle G,k\rangle \mid G \text{ è una CFG e } L(G) \text{ contiene asattamente } k \text{ stringhe dove } k \geq 0 \text{ o } k = \infty\}$. Mostrare che C_{CFG} è decidibile.

4.14 Sia $C = \{\langle G,x\rangle \mid G \text{ è una CFG e } x \text{ è una sottostringa di } y \in L(G)\}$. Mostrare che C è decidibile. (Suggerimento: una soluzione elegante per questo problema usa il decisore per E_{CFG}.)

*4.15 Sia $E = \{\langle M\rangle \mid M \text{ è un DFA che accetta alcune stringhe contenenti un numero di 1 maggiore del numero di 0}\}$. Mostrare che E è decidibile. (Suggerimento: possono essere utili i teoremi sui CFL.)

*4.16 Sia $PAL_{\mathsf{DFA}} = \{\langle M\rangle \mid M \text{ è un DFA che accetta alcune stringhe palindrome }\}$. Mostrare che PAL_{DFA} è decidibile. (Suggerimento: possono essere utili i teoremi sui CFL.)

A**4.17** Sia $BAL_{\text{DFA}} = \{\langle M\rangle |\ M$ è un DFA che accetta alcune stringhe contenenti un ugual numero di 0 e 1$\}$. Mostrare che BAL_{DFA} è decidibile. (Suggerimento: possono essere utili i teoremi sui CFL.)

4.18 Uno ***stato inutile*** in un automa a pila è uno stato in cui l'automa non entra mai, qualunque sia la stringa in input. Considerare il problema di determinare se un automa a pila ha stati inutili. Formulare tale problema come un linguaggio e mostrare che è decidibile.

A**4.19** Diciamo che un NFA è ***ambiguo*** se accetta qualche stringa su due diversi cammini di computazione. Sia $AMBIG_{\text{NFA}} = \{\langle N\rangle |\ N$ è un NFA ambiguo $\}$. Mostrare che $AMBIG_{\text{NFA}}$ è decidibile. (Suggerimento: Un modo elegante per risolvere questo problema è costruire un DFA opportuno e quindi eseguire E_{DFA} su di esso).

4.20 Sia $S = \{\langle M\rangle |\ M$ è un DFA che accetta $w^{\mathcal{R}}$ ogni volta che esso accetta $w\}$. Mostrare che S è decidibile.

4.21 Sia $PREFIX\text{-}FREE_{\text{REX}} = \{\langle R\rangle |\ R$ è un'espressione regolare, $L(R)$ è prefisso$\}$. Mostrare che $PREFIX\text{-}FREE_{\text{REX}}$ è decidibile. Perché un approccio simile fallisce nel dimostrare che $PREFIX\text{-}FREE_{\text{CFG}}$ è decidibile?

4.22 Siano A e B due linguaggi disgiunti. Diciamo che il linguaggio C ***separa*** A e B se $A \subseteq C$ e $B \subseteq \overline{C}$. Mostrare che per ogni coppia di linguaggi coTuring-riconoscibili disgiunti esiste un linguaggio decidibile che li separa.

*4.23 Mostrare che la classe dei linguaggi decidibili non è chiuso rispetto agli omomorfismi.

*4.24 Sia C un linguaggio. Dimostrare che C è Turing-riconoscibile sse esiste un linguaggio decidibile D tale che $C = \{x |\ \exists y\ (\langle x, y\rangle \in D)\}$.

4.25 Dimostrare che EQ_{DFA} è decidibile testando i due DFA su tutte le stringhe la cui lunghezza è limitata superiormente da un qualche valore. Calcolare un valore valido.

4.26 Sia $A = \{\langle R\rangle |\ R$ un'espressione regolare che descrive un linguaggio che contiene almeno una stringa w che ha 111 come sottostringa (cioè, $w = x111y$ per qualche x e $y)\}$. Mostrare che A è decidibile.

*4.27 Mostrare che il problema di determinare se una CFG genera tutte le stringhe in 1^* è decidibile. In altre parole, mostrare che $\{\langle G\rangle |\ G$ è una CFG su $\{0,1\}$ e $1^* \subseteq L(G)\}$ è un linguaggio decidibile.

A**4.28** Sia $\Sigma = \{0,1\}$. Mostrare che il problema di determinare se una CFG genera qualche stringa 1^* è decidibile. In altre parole, mostrare che

$$\{\langle G\rangle |\ G \text{ è una CFG su } \{0,1\} \text{ e } 1^* \cap L(G) \neq \emptyset\}$$

è un linguaggio decidibile.

4.29 Sia $A = \{\langle R, S\rangle |\ R$ e S sono espressioni regolari e $L(R) \subseteq L(S)\}$. Mostrare che A è decidibile.

A**4.30** Sia $A = \{\langle M\rangle |\ M$ è un DFA che non accetta alcuna stringa che contiene un numero dispari di 1$\}$. Mostrare che A è decidibile.

4.31 Sia $INFINITE_{\text{PDA}} = \{\langle M\rangle |\ M$ è un PDA e $L(M)$ è un linguaggio infinito$\}$. Mostrare che $INFINITE_{\text{PDA}}$ è decidibile.

A**4.32** Sia $INFINITE_{\text{DFA}} = \{\langle A\rangle |\ A$ è un DFA e $L(A)$ è un linguaggio infinito$\}$. Mostrare che $INFINITE_{\text{DFA}}$ è decidibile.

SOLUZIONI SELEZIONATE

4.1 (a) Sì. Il DFA M accetta 0100.

(b) No. M non accetta 011.

(c) No. Questo input ha una sola componente e quindi non è nella forma corretta.

(d) No. La prima componente non è un'espressione regolare e quindi l'input non è in forma corretta

(e) No. Il linguaggio di M non è vuoto.

(f) Sì. M accetta lo stesso linguaggio di se stessa.

4.5 Sia $s_1, s_2, \ldots$ la lista di tutte le stringhe in Σ^*. La seguente TM riconosce $\overline{E_{\text{TM}}}$.

"Su input $\langle M\rangle$, dove M è una TM:

1. Ripete per $i = 1, 2, 3, \ldots$.
2. Esegue M per i passi su ogni input, $s_1, s_2, \ldots, s_i$.
3. Se M ne ha accettato almeno uno, *accetta*. Altrimenti, continua."

4.6 (a) No, f non è biettiva poichè $f(1) = f(3)$.

(d) Sì, g è biettiva.

4.17 Il linguaggio di tutte le stringhe con un numero uguale di 0 e di 1 è un linguaggio context-free, generato dalla grammatica $S \to 1S0S \mid 0S1S \mid \varepsilon$. Sia P il PDA che riconosce questo linguaggio. Costruire una TM M per BAL_{DFA}, che opera come segue. Su input $\langle B\rangle$, dove B un DFA, usare B e P per costruire un nuovo PDA R che riconosce l'intersezione dei linguaggi B e P. Quindi verificare se il linguaggio di R è vuoto. Se il linguaggio è vuoto, *rifiuta*; altrimenti, *accetta*.

4.19 La procedura seguente decide $AMBIG_{\text{NFA}}$. Dato un NFA N, progettiamo un DFA D che simula N e accetta una stringa sse essa è accettata da N lungo due differenti cammini di computazione. Poi usiamo un decisore per E_{DFA} per determinare se D accetta almeno una stringa. La nostra strategia per la costruzione di D è simile alla conversione da NFA a DFA nel Teorema 1.39. Simuliamo N mantenendo un segno su ogni stato attivo. Iniziamo mettendo un segno rosso sullo stato iniziale e su ogni stato raggiungibile dallo stato iniziale, lungo le ε -transizioni. Ci muoviamo, aggiungiamo e rimuoviamo i segni in conformità con le transizioni di N, conservando il colore dei segni. Quando due o più segni vengono spostati nello stesso stato, sostituiamo i segni con uno di colore blu. Dopo aver letto l'input, accettiamo se vi è un segno blu su uno stato di accettazione di N oppure due differenti stati di accettazione di N hanno un segno rosso. Il DFA D ha uno stato corrispondente ad ogni possibile posizione dei segni. Per ogni stato di N, si possono verificare tre casi: esso può contenere un segno rosso, uno blu, o nessun segno. Così, se N ha n stati, D avrà 3^n stati. I suoi stati iniziale e di accettazione e la funzione di transizione sono definiti per effettuare la simulazione.

4.28 Avete dimostrato nel Problema 2.30 che se C è un linguaggio context-free e R è un linguaggio regolare, allora $C \cap R$ è context free. Quindi, $1^* \cap L(G)$ è context free. La seguente TM decide il linguaggio di tale problema.

"Su input $\langle G \rangle$:

1. Costruisce la CFG H tale che $L(H) = 1^* \cap L(G)$.
2. Testa se $L(H) = \emptyset$ usando il decisore R per la E_{CFG} del Teorema 4.8.
3. Se R accetta, *rifiuta*; if R rifiuta, *accetta*."

4.30 La seguente TM decide A.

"Su input $\langle M \rangle$:

1. Costruisce il DFA O che accetta ogni stringa contenente un numero dispari di 1.
2. Costruisce un DFA B tale che $L(B) = L(M) \cap L(O)$.
3. Testa se $L(B) = \emptyset$ usando il decisore T per E_{DFA} del Teorema 4.4.
4. Se T accetta, *accetta*; se T rifiuta, *rifiuta*."

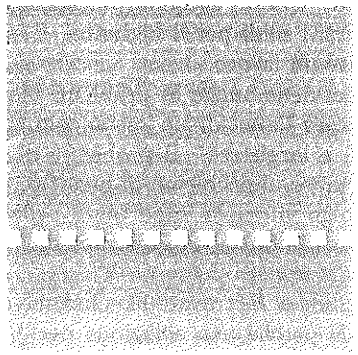
4.32 Il seguente TM I decide $INFINITE_{\mathsf{DFA}}$.

I = "Su input $\langle A \rangle$, dove A è un DFA:

1. Sia k il numero di stati di A.
2. Costruisce un DFA D che accetta tutte le stringhe di lunghezza k o più.
3. Costruisce un DFA M tale che $L(M) = L(A) \cap L(D)$.
4. Testa se $L(M) = \emptyset$, usando il decisore per E_{DFA} del Teorema 4.4.
5. Se T accetta, *rifiuta*; se T rifiuta, *accetta*."

Questo algoritmo funziona perché un DFA che accetta un numero infinito di stringhe deve accettare stringhe arbitrariamente lunghe. Quindi l'algoritmo accetta tali DFA. Viceversa, se l'algoritmo accetta un DFA, il DFA accetta almeno una stringa di lunghezza k o più, dove k è il numero di stati del DFA. Questa può essere iterata come nel pumping lemma per i linguaggi regolari per ottenere un numero infinito di stringhe accettate.

5



RIDUCIBILITÀ

Nel Capitolo 4 abbiamo adottato la macchina di Turing quale nostro modello di computer di uso generale. Abbiamo presentato vari esempi di problemi che sono risolvibili con una macchina di Turing e dato un esempio di un problema, A_{TM}, che è computazionalmente irrisolvibile. In questo capitolo esamineremo vari altri problemi irrisolvibili. Nel far ciò introdurremo il metodo principale per dimostrare che alcuni problemi sono computazionalmente irrisolvibili. Tale metodo si chiama ***riducibilità***.

Una ***riduzione*** è un modo di convertire un problema in un altro problema in modo tale che una soluzione al secondo problema può essere usata per risolvere il primo problema. Un qualcosa di simile accade spesso nella vita di tutti i giorni, anche se di solito non si fa riferimento ciò allo stesso modo. Per esempio, supponiamo di voler trovare la strada in una nuova città. Sappiamo che fare ciò sarebbe semplice se avessimo una mappa. In questo modo è possibile ridurre il problema di spostarsi nella città al problema di ottenere una mappa della città.

La riducibilità coinvolge sempre due problemi, che noi chiamiamo A e B. Se A si riduce a B, possiamo usare una soluzione per B per risolvere A. Quindi, nel nostro esempio, A è il problema di orientarvi nella città e B è il problema di ottenere una mappa. Notate che la riducibilità non dice nulla circa la soluzione dei problemi A o B individualmente, ma soltanto qualcosa circa la risolubilità di A quando abbiamo una soluzione per B. Di seguito sono riportati ulteriori esempi di riducibilità. Il problema del

viaggio da Boston a Parigi si riduce al problema dell'acquisto di un biglietto aereo tra le due città. Questo secondo problema a sua volta si riduce al problema di procurarsi i soldi per il biglietto. Il quale problema si riduce al problema di trovare un lavoro.

La riducibilità interviene anche in problemi matematici. Per esempio, il problema di misurare l'area di un rettangolo si riduce al problema di misurarne lunghezza e larghezza. Il problema di risolvere un sistema di equazioni lineari si riduce al problema di invertire una matrice.

La riducibilità svolge un ruolo importante nella classificazione dei problemi in base alla decidibilità e, successivamente, anche nella teoria della complessità.

Quando A è riducibile a B, trovare la soluzione di A non può essere più difficile di risolvere B perché una soluzione per B offre una soluzione ad A. In termini di teoria della computabilità, se A è riducibile a B e B è decidibile, allora anche A è decidibile. Equivalentemente, se A è indecidibile e riducibile a B, B è indecidibile. Questa ultima parte è la chiave per dimostrare che alcuni problemi sono indecidibili.

In breve, il nostro metodo per dimostrare che un problema è indecidibile sarà quello di mostrare che qualche altro problema già noto per essere indecidibile si riduce ad esso.

5.1
PROBLEMI INDECIDIBILI DALLA TEORIA DEI LINGUAGGI

Abbiamo già stabilito l'indecidibilità di A_{TM}, il problema di determinare se una macchina di Turing accetta un dato input. Consideriamo ora un problema affine, $HALT_{\mathsf{TM}}$, il problema di determinare se una macchina di Turing si ferma (accettando o rifiutando) su un dato input. Questo problema è generalmente noto come il ***problema della fermata***. Usiamo l'indecidibilità di A_{TM} per dimostrare l'indecidibilità del problema della fermata riducendo A_{TM} a $HALT_{\mathsf{TM}}$. Sia

$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle | \ M \text{ è una TM e } M \text{ si ferma su input } w\}.$$

TEOREMA 5.1

$HALT_{\mathsf{TM}}$ è indecidibile.

IDEA. Questa dimostrazione è per assurdo. Assumiamo che $HALT_{\mathsf{TM}}$ è decidibile ed utilizziamo questa assunzione per dimostrare che A_{TM} è decidibile, contraddicendo il Teorema 4.11. L'idea chiave è quella di mostrare che A_{TM} è riducibile ad $HALT_{\mathsf{TM}}$.

Supponiamo di avere una TM R che decide $HALT_{\mathsf{TM}}$. Usiamo quindi R per costruire S, una TM che decide A_{TM}. Per intuire come possiamo costruire S, immaginate di essere S. Il vostro compito è decidere A_{TM}. Ricevete un input della forma $\langle M, w\rangle$. Dovete dare in output *accetta* se M accetta w, e dovete fornire in output *rifiuta* se M cicla oppure rifiuta w. Provate a simulare M su w. Se accetta o rifiuta, fate lo stesso. Ma potreste non essere in grado di determinare se M è entrata in un ciclo, in questo caso la simulazione non terminerà. Questo non va bene perché voi siete un decisore e non vi è permesso ciclare. Quindi questa idea non funziona. Utilizziamo, invece, l'ipotesi che avete una TM R che decide $HALT_{\mathsf{TM}}$. Con R, potete verificare se M si ferma su w. Se R indica che M non si ferma su w, rifiutate perché $\langle M, w\rangle$ non è in A_{TM}. Tuttavia, se R indica che M si ferma su w, potete fare la simulazione senza alcun pericolo di ciclare. Così, se la TM R esistesse, potremmo decidere A_{TM}, ma sappiamo che A_{TM} è indecidibile. In virtù di questa contraddizione possiamo concludere che R non esiste. Pertanto $HALT_{\mathsf{TM}}$ è indecidibile.

DIMOSTRAZIONE. Assumiamo al fine di ottenere una contraddizione che la TM R decide $HALT_{\mathsf{TM}}$. Costruiamo la TM S per decidere A_{TM}, che opera come segue.

S = "Su input $\langle M; w\rangle$, una codifica di una TM M ed una stringa w:

1. Esegue la TM R su input $\langle M, w\rangle$.
2. Se R rifiuta, *rifiuta*.
3. Se R accetta, simula M su w finché non si ferma
4. Se M ha accettato, *accetta*; se M ha rifiutato, *rifiuta*."

Chiaramente, se R decide $HALT_{\mathsf{TM}}$, allora S decide A_{TM}. Poichè A_{TM} è indecidibile, allora anche $HALT_{\mathsf{TM}}$ deve essere indecidibile.

Il Teorema 5.1 illustra il nostro metodo per dimostrare che un problema è indecidibile. Questo metodo è comune alla maggior parte delle dimostrazioni di indecidibilità, tranne che per l'indecidibilità di A_{TM} stesso, che viene dimostrata direttamente utilizzando il metodo della diagonalizzazione.

Presentiamo ora vari altri teoremi e le loro dimostrazioni come ulteriori esempi dell'uso della riducibilità per dimostrare l'indecidibilità. Sia

$$E_{\mathsf{TM}} = \{\langle M\rangle | \ M \text{ è una TM e } L(M) = \emptyset\}.$$

TEOREMA 5.2

E_{TM} è indecidibile.

IDEA. Seguiamo lo schema adottato nel Teorema 5.1. Assumiamo che E_{TM} è decidibile e mostriamo che A_{TM} è decidibile – una contraddizione. Sia R una TM che decide E_{TM}. Utilizziamo R per costruire la TM S che decide A_{TM}. Come funzionerà S quando riceve in input$\langle M, w\rangle$?

Un'idea è che S esegue R su input $\langle M\rangle$ e vede se R accetta. Se lo fa, sappiamo che $L(M)$ è vuoto e quindi che M non accetta w. Ma, se R rifiuta $\langle M\rangle$, tutto quello che sappiamo è che $L(M)$ non è vuoto e, di conseguenza, che M accetta qualche stringa – ma ancora non sappiamo se M accetta la stringa w in particolare. Quindi abbiamo bisogno di utilizzare un'idea diversa. Invece di eseguire R su $\langle M\rangle$, eseguiamo R su una modifica di $\langle M\rangle$. Modifichiamo $\langle M\rangle$ così da garantire che M rifiuta tutte le stringhe tranne w, ma su input w, funziona come al solito. A questo punto, utilizziamo R per determinare se la macchina modificata riconosce il linguaggio vuoto. L'unica stringa che adesso la macchina può accettare è w, per cui il suo linguaggio sarà non vuoto sse accetta w. Se R accetta quando riceve in input la descrizione della macchina modificata, sappiamo che la macchina modificata non accetta nulla e che M non accetta w.

DIMOSTRAZIONE Descriviamo la macchina modificata già descritta nell'idea di dimostrazione usando la nostra notazione standard. Chiamiamola M_1.

M_1 = "Su input x:

1. Se $x \neq w$, *rifiuta*.
2. Se $x = w$, esegue M su input w e *accetta* se M accetta."

Questa macchina ha la stringa w come parte della sua descrizione. Essa verifica se $x = w$ nel modo ovvio, attraverso la scansione dell'input e confrontandolo carattere per carattere con w per determinare se coincidono. Mettendo insieme tutto questo, assumiamo che la TM R decide E_{TM} e costruiamo la TM S che decide A_{TM} come segue.

S = "Su input $\langle M, w\rangle$, una codifica di una TM M e una stringa w:

1. Usa la descrizione di M e w per costruire la TM M_1 descritta sopra.
2. Esegue R su input $\langle M_1\rangle$.
3. Se R accetta, *rifiuta*; se R rifiuta, *accetta*."

Notate che S deve essere effettivamente in grado di calcolare una descrizione di M_1 da una descrizione di M e w. Può farlo, perché ha bisogno solo di aggiungere ad M alcuni stati in più che svolgono il test $x = w$. Se R fosse un decisore per E_{TM}, S sarebbe un decisore per A_{TM}. Un decisore per A_{TM} non può esistere, quindi sappiamo che E_{TM} deve essere indecidibile.

Un altro interessante problema computazionale che concerne le macchine di Turing è quello di determinare se una data macchina di Turing riconosce un linguaggio che può essere riconosciuto anche da un modello di calcolo più semplice. Ad esempio, sia $REGULAR_{\mathsf{TM}}$ il problema di determinare se una data macchina di Turing ha un automa finito equivalente. Questo problema equivale a determinare se la macchina di Turing riconosce un linguaggio regolare. Sia

$$REGULAR_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ è una TM ed } L(M) \text{ è un linguaggio regolare}\}.$$

TEOREMA 5.3

$REGULAR_{\mathsf{TM}}$ è indecidibile.

IDEA. Come al solito per i teoremi sull'indecidibilità, questa dimostrazione consiste in una riduzione da A_{TM}. Assumiamo che $REGULAR_{\mathsf{TM}}$ sia deciso da una TM R e utilizziamo questa assunzione per costruire una TM S che decide A_{TM}. Questa volta risulta meno ovvio come utilizzare la capacità di R di assistere S nel suo compito. Tuttavia siamo in grado di farlo. L'idea è che S prenda il suo input $\langle M, w\rangle$ e modifichi M in modo che la risultante TM riconosca un linguaggio regolare se e solo se M accetta w. Chiamiamo M_2 la macchina così modificata. Progettiamo M_2 in modo che riconosca il linguaggio non regolare $\{0^n1^n \mid n \geq 0\}$ se M non accetta w, e riconosca il linguaggio regolare Σ^* se M accetta w. Dobbiamo specificare come S può costruire una tale M_2 da M e w. Qui, M_2 accetta automaticamente tutte le stringhe in $\{0^n1^n \mid n \geq 0\}$. Inoltre, se M accetta w, M_2 accetta tutte le altre stringhe.

Notate che la TM M_2 *non* è costruita con lo scopo di essere eseguita su qualche input – un malinteso comune. Costruiamo M_2 al solo scopo di dare in input la sua descrizione al decisore per $REGULAR_{\mathsf{TM}}$ che abbiamo assunto esistere. Quando tale decisore dà la sua risposta, possiamo usarla per rispondere se M accetta w o meno. Quindi possiamo decidere A_{TM}, una contraddizione.

DIMOSTRAZIONE Definiamo R come una TM che decide $REGULAR_{\mathsf{TM}}$ e costruiamo una TM S che decide A_{TM}. Allora S funziona come segue.

S = "Su input $\langle M, w\rangle$, dove M è una TM e w è una stringa:

1. Costruisce la seguente TM M_2.
   M_2 = "Su input x:
   1. Se x ha la forma 0^n1^n, *accetta*.
   2. Se x non ha tale forma, esegue M su input w e *accetta* se M accetta w."
2. Esegue R su input $\langle M_2\rangle$.
3. Se R accetta, *accetta*; se R rifiuta, *rifiuta*."

Con prove simili, si può dimostrare che i problemi di testare se il linguaggio di una macchina di Turing è un linguaggio context-free, un linguaggio decidibile o anche un linguaggio finito sono indecidibili. Infatti, un risultato generale, chiamato teorema di Rice, afferma che determinare una *qualsiasi proprietà* dei linguaggi riconosciuti da macchine di Turing è indecidibile. Diamo il teorema di Rice nel Problema 5.16.

Finora, la nostra strategia per dimostrare l'indecidibilità di linguaggi prevede una riduzione da A_{TM}. A volte la riduzione da qualche altro linguaggio indecidibile, come ad esmpio E_{TM}, risulta più conveniente per dimostrare che certi linguaggi sono indecidibili. Il Teorema 5.4 mostra che verificare l'equivalenza di due macchine di Turing è un problema indecidibile. Potremmo provarlo con una riduzione da A_{TM}, ma abbiamo l'occasione di dare un esempio di una prova di indecidibilità mediante una riduzione da E_{TM}. Sia

$$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ ed } M_2 \text{ sono TM ed } L(M_1) = L(M_2)\}.$$

TEOREMA 5.4

EQ_{TM} è indecidibile.

IDEA. Mostriamo che se EQ_{TM} fosse decidibile allora anche E_{TM} sarebbe decidibile eseguendo una riduzione da E_{TM} a EQ_{TM}. L'idea è semplice. E_{TM} è il problema di determinare se il linguaggio di una TM è vuoto. EQ_{TM} è il problema di determinare se i linguaggi delle due TM sono uguali. Se uno di questi linguaggi è $\emptyset$, ci ritroviamo con il problema di determinare se il linguaggio dell'altra macchina è vuoto, cioè il problema E_{TM}. Quindi, in un certo senso, il problema E_{TM} è un caso particolare del problema EQ_{TM} dove una delle macchine è fissata per riconoscere il linguaggio vuoto. Questa idea rende la riduzione più facile.

DIMOSTRAZIONE Consideriamo una TM R che decide EQ_{TM} e costruiamo la TM S che decide E_{TM} come segue.

S = "Su input $\langle M\rangle$, dove M è una TM:

1. Esegue R su input $\langle M, M_1\rangle$, dove M_1 è una TM che rifiuta ogni input.
2. Se R accetta, *accetta*; se R rifiuta, *rifiuta*."

Se R decide EQ_{TM}, S decide E_{TM}. Ma il Teorema 5.2 ci dice che E_{TM} è indecidibile, quindi anche EQ_{TM} deve essere indecidibile.

Riduzioni mediante storie di computazione

Il metodo mediante storie di computazione è una tecnica importante per dimostrare che A_{TM} è riducibile a certi linguaggi. Questo metodo è spesso utile quando il problema da mostrare indecidibile comporta la dimostrazione dell'esistenza di qualcosa. Per esempio, questo metodo viene utilizzato per mostrare l'indecidibilità del decimo problema di Hilbert, verificare l'esistenza di radici intere di un polinomio. La computazione per una macchina di Turing su un input è semplicemente la sequenza delle configurazioni attraverso cui la macchina passa mentre elabora l'input. Si tratta di una registrazione completa della computazione della macchina.

DEFINIZIONE 5.5

Sia M una macchina di Turing e w una stringa di input. Una ***storia di computazione accettante*** per M su w è una sequenza di configurazioni, $C_1, C_2, \ldots, C_l$, dove C_1 è la configurazione iniziale di M su w, C_l è una configurazione di accettazione per M, e ogni C_i segue da C_{i-1} in accordo alle regole di M. Una ***storia di computazione di rifiuto*** per M su w è definita in modo analogo, tranne per il fatto che C_l è una configurazione di rifiuto.

Le storie di computazione sono sequenze finite. Se M non si ferma su w, non esiste una storia di computazione accettante o di rifiuto per M su w. Le macchine deterministiche hanno al massimo una storia di computazione su ogni input. Le macchine non deterministiche possono avere molte storie di computazione su un singolo input, corrispondenti ai diversi rami di computazione. Per ora, continuiamo a concentrarci sulle macchine deterministiche. La nostra prima dimostrazione di indecidibilità utilizzando il metodo delle storie di computazione riguarda un tipo di macchina chiamato automa linearmente limitato.

DEFINIZIONE **5.6**

Un ***automa linearmente limitato*** è un tipo ristretto di macchina di Turing in cui alla testina del nastro non è permesso di spostarsi fuori dalla parte del nastro che contiene l'input. Se la macchina tenta di spostare la testina al di fuori dell'input, la testina rimane dove si trova – come accade in una normale macchina di Turing dove la testina non avanza oltre l'estremità sinistra del nastro.

Un automa linearmente limitato è una macchina di Turing con una quantità limitata di memoria, come mostrato schematicamente nella seguente figura. Può risolvere solo problemi che richiedono un quantità di memoria al più pari alla dimensione dell'input. L'uso di un alfabeto di nastro più grande dell'alfabeto dei input consente alla memoria disponibile di essere aumentata di un fattore costante. Quindi diciamo che per un input di lunghezza n, la quantità di memoria disponibile è lineare in n – da qui il nome di questo modello.

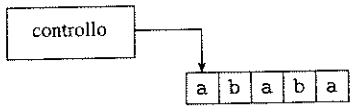


FIGURA 5.7
Schema di un automa linearmente limitato

Nonostante il vincolo sulla memoria, un automa linermente limitato (LBA) è abbastanza potente. Ad esempio, i decisori per A_{DFA}, A_{CFG}, E_{DFA}, e E_{CFG} sono tutti dei LBA. Ogni CFL può essere deciso da un LBA. Infatti, ottenere un linguaggio decidibile che non può essere deciso da un LBA non è semplice. Svilupperemo le tecniche per farlo nel Capitolo 9.

Qui, A_{LBA} è il problema di determinare se un LBA accetta il suo input. Anche se A_{LBA} coincide con il problema indecidibile A_{TM} quando la macchina di Turing è un LBA, possiamo dimostrare che A_{LBA} è decidibile. Sia

$$A_{\mathsf{LBA}} = \{\langle M, w\rangle |\ M \text{ è un LBA che accetta la stringa } w\}.$$

Prima di dimostrare la decidibilità di A_{LBA}, troviamo utile il seguente lemma. Esso dice che un LBA può avere solo un numero limitato di configurazioni quando ha in input una stringa di lunghezza n.

LEMMA 5.8

Sia M un LBA con q stati e g simboli nell'alfabeto di nastro. Esistono esattamente qng^n configurazioni distinte di M per un nastro di lunghezza n.

DIMOSTRAZIONE. Ricordiamo che una configurazione di M è come un'istantanea durante il suo calcolo. Una configurazione comprende lo stato del controllo, la posizione della testina ed il contenuto del nastro. Qui, M ha q stati. La lunghezza del suo nastro è n, per cui la testina può essere in una delle n posizioni, e g^n possibili stringhe di simboli del nastro compaiono sul nastro. Il prodotto di queste tre quantità è il numero totale di configurazioni differenti di M con un nastro di lunghezza n.

TEOREMA 5.9

A_{LBA} è decidibile.

IDEA. Per decidere se un LBA M accetta l'input w, simuliamo M su w. Nel corso della simulazione, se M si ferma e accetta o rifiuta, noi accettiamo o rifiutiamo di conseguenza. La difficoltà si verifica se M cicla su w. Dobbiamo essere in grado di rilevare tali situazioni in modo da poter fermare e rifiutare. L'idea per rilevare se M cicla è che, mentre M computa su w, essa va da configurazione a configurazione. Se M ripetesse una configurazione essa continuerebbe a ripetere tale configurazione pi volte, quindi ciclerebbe. Poichè M è un LBA, la quantità di nastro disponibile è limitata. Il Lemma 5.8, ci dice che M può essere solo in un numero limitato di configurazioni su questa quantità di nastro. Quindi è disponibile solo una quantità limitata di tempo per M prima che si trovi in una configurazione in cui è già stata precedentemente. Rilevare che è entrata in un ciclo è possibile simulando M per il numero di passi dati dal Lemma 5.8. Se M non si ferma entro tale numero di passi, allora deve essere entrata in un ciclo.

DIMOSTRAZIONE L'algoritmo che decide A_{LBA} funziona come segue.

L = "Su input $\langle M, w\rangle$, dove M è un LBA e w è una stringa:

1. Simula M su w per qng^n passi o finché non si ferma.
2. Se M si è fermata, *accetta* se ha accettato e *rifiuta* se ha rifiutato. Se non si è fermata, *rifiuta*."

Se M su w non si è fermata entro qng^n passi, essa deve ripetere una configurazione, in accordo al Lemma 5.8 e quindi entra in ciclo. Ecco perché il nostro algoritmo in questo caso rifiuta.

Il Teorema 5.9 mostra che LBA e TM differiscono in maniera essenziale: per i primi il problema dell'accettazione è decidibile, ma per le macchine di Turing non lo è. Tuttavia, alcuni problemi che coinvolgono gli LBA, restano indecidibili. Uno di questi è il problema del vuoto $E_{\mathsf{LBA}} = \{\langle M\rangle |\ M \text{ è un LBA dove } L(M) = \emptyset\}$. Per dimostrare che E_{LBA} è indecidibile, diamo una riduzione che utilizza il metodo delle storie di computazione.

TEOREMA 5.10

E_{LBA} è indecidibile.

IDEA. Questa dimostrazione utilizza una riduzione da A_{TM}. Si dimostra che se E_{LBA} fosse decidibile, lo sarebbe anche A_{TM}. Supponiamo che E_{LBA} sia decidibile. Come possiamo utilizzare questa assunzione per decidere A_{TM}? Per una TM M e un input w, possiamo determinare se M accetta w costruendo un LBA B e poi verificando se $L(B)$ è vuoto. Il linguaggio che riconosce B comprende tutte le storie di computazione accettanti di M su w. Se M accetta w, tale linguaggio contiene almeno una stringa per cui è diverso dal vuoto. Se M non accetta w, questo linguaggio è vuoto. Se siamo in grado di determinare se il linguaggio di B è vuoto o meno, chiaramente siamo in grado di determinare se M accetta w o meno. Adesso vediamo come costruire B da M e w. Si noti che abbiamo bisogno di mostrare più che la semplice esistenza di B. Dobbiamo mostrare come una macchina di Turing può ottenere una descrizione di B a partire dalle descrizioni di M e w. Come nelle riduzioni date in precedenza per dimostrare l'indecidibilità di un linguaggio dato, costruiamo B solo per dare in input la sua descrizione al presunto decisore per E_{LBA} non per eseguire B su qualche input.

Costruiamo B in modo che accetti il suo input x se x è una storia di computazione accettante di M su w. Ricordate che una storia di computazione accettante è una sequenza di configurazioni, $C_1, C_2, \ldots, C_l$ che M occupa quando accetta una qualche stringa w. Ai fini di questa prova si assume che la storia di computazione accettante viene data come una singola stringa con le configurazioni separate l'una dall'altra dal simbolo #, come illustrato nella Figura 5.11.

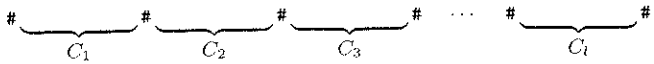


FIGURA 5.11
Un possibile input per B

L'automa linearmente limitato B funziona come segue. Quando riceve un input x, B deve accettare se x è una storia di computazione accettante per M su w. In primo luogo, B si ferma su x secondo i delimitatori contenuti nelle stringhe $C_1, C_2, \ldots, C_l$. Poi B determina se i vari C_i soddisfano le tre condizioni di una storia di computazione accettante.

1. C_1 è la configurazione iniziale per M su w.
2. Ogni C_{i+1} segue legalmente da C_i.
3. C_l è una configurazione di accettazione per M.

La configurazione iniziale C_1 per M su w è la stringa $q_0 w_1 w_2 \cdots w_n$, dove q_0 è lo stato iniziale di M. Qui, B ha questa stringa direttamente integrata, quindi è in grado di verificare la prima condizione. Una configurazione di accettazione è una che contiene lo stato q_{accept} quindi B può controllare la terza condizione scandendo C_l per q_{accept}. La seconda condizione è la più difficile da controllare. Per ogni coppia di configurazioni adiacenti, B controlla se C_{i+1} segue legalmente da C_i. Questa fase consiste nel verificare che C_i e C_{i+1} sono identiche tranne che per la posizioni sotto la testina e quelle adiacenti ad essa in C_i. Queste posizioni devono essere aggiornate in base alla funzione di transizione di M. Allora B verifica che l'aggiornamento è stato fatto correttamente zig-zagando tra le posizioni corrispondenti di C_i e C_{i+1}. Per tenere traccia delle posizioni attuali, durante la fase di zig-zag, B marca la posizione corrente con dei punti sul nastro. Infine, se le condizioni 1, 2 e 3 sono soddisfatte, B accetta il suo input. Invertendo la risposta del decisore, otteniamo la risposta alla domanda se M accetta w o meno. In questo modo possiamo decidere A_{TM}, ottenendo così una contraddizione.

DIMOSTRAZIONE Ora siamo pronti a formalizzare la riduzione da A_{TM} a E_{LBA}. Supponiamo che la TM R decide E_{LBA}. Costruiamo una TM S che decide A_{TM} come segue.

S = "Su input $\langle M, w\rangle$, dove M è una TM e w è una stringa:

1. Costruisce un LBA B da M e w come descritto nell'idea di dimostrazione.
2. Esegue R su input $\langle B\rangle$.
3. Se R rifiuta, *accetta*; se R accetta, *rifiuta*."

Se R accetta $\langle B\rangle$, allora $L(B) = \emptyset$. Quindi M non ha alcuna storia di computazione accettante su w e M non accetta w. Di conseguenza S rifiuta $\langle M, w\rangle$. Allo stesso modo, se R rifiuta $\langle B\rangle$, il linguaggio di B è diverso dal vuoto. L'unica stringa che B può accettare è una storia di computazione accettante di M su w. Quindi M deve accettare w. Di conseguenza S accetta $\langle M, w\rangle$. La Figura 5.12 illustra B.

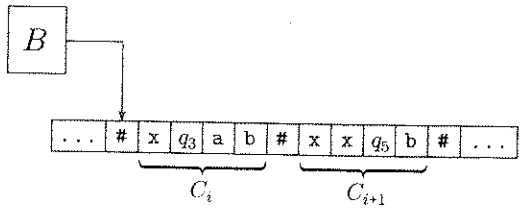


FIGURA 5.12
Un LBA B che controlla una storia di computazione di una TM

Possiamo anche usare la tecnica di riduzione mediante storie di computazione per stabilire l'indecidibilità di alcuni problemi legati alle grammatiche context-free ed agli automi a pila. Ricordiamo che nel Teorema 4.8 abbiamo presentato un algoritmo per decidere se una grammatica context-free non genera alcuna stringa, cioè se $L(G) = \emptyset$. Ora dimostriamo che un problema correlato è indecidibile. Si tratta di determinare se una grammatica context-free genera tutte le possibli stringhe. Dimostrare che questo problema è indecidibile è il passo principale per dimostrare che il problema dell'equivalenza per le grammatiche context-free è indecidibile. Sia

$$ALL_{\mathsf{CFG}} = \{\langle G \rangle | \ G \text{ è una CFG ed } L(G) = \Sigma^*\}.$$

TEOREMA 5.13

ALL_{CFG} è indecidibile.

DIMOSTRAZIONE. La dimostrazione procede per assurdo. Per ottenere la contraddizione assumiamo che ALL_{CFG} sia decidibile e usiamo questa assunzione per dimostrare che A_{TM} è decidibile. Questa dimostrazione è simile a quella del Teorema 5.10, ma con un tocco in più: è una riduzione da A_{TM} attraverso storie di computazione, ma modifichiamo leggermente la rappresentazione della storia di computazione per una ragione tecnica che spiegheremo in seguito. Ora descriviamo come utilizzare una procedura di decisione per ALL_{CFG} per decidere A_{TM}. Per una TM M ed un input w costruiamo una CFG G che genera tutte le stringhe se e solo se M non accetta w. Così, se M non accetta w, G *non* genera una particolare stringa. Questa stringa è – indovinate – la storia di computazione accettante per M su w. Cioè G è progettata in modo da generare tutte le stringhe che *non* sono una storia di computazione accentante per M su w. Per far siche la CFG G generi tutte le stringhe che non costituiscono una storia di computazione accettante per M su w, utilizziamo la seguente strategia. Una stringa può non essere una storia di computazione accettante per vari motivi. Una storia di computazione accettante per M su w è del tipo $\#C_1\#C_2\#\cdots\#C_l\#$, dove C_i è la configurazione di M al passo i-mo di computazione su w. Quindi, G genera tutte le stringhe

1. che *non* iniziano con C_1,
2. che *non* terminano con una configurazione di accettazione, o
3. in cui qualche C_i *non* produce C_{i+1} in accordo alle regole di M.

Se M non accetta w, non esiste alcuna storia di computazione accettante, quindi *tutte* le stringhe falliscono in un modo o nell'altro. Pertanto G genererebbe tutte le stringhe, come desiderato.

Ora arriviamo alla effettiva costruzione di G. Invece di costruire G, costruiamo un PDA D. Sappiamo di poter usare la costruzione data nel Teorema 2.20 (pagina 120) per convertire D in una CFG. Operiamo in questo modo perché, per i nostri scopi, la progettazione di un PDA risulta più semplice della progettazione di una CFG. In questa istanza, D partirà in maniera non deterministica per indovinare quale delle tre precedenti condizioni verificare. Una ramificazione controlla se l'inizio della stringa di input è C_1 e accetta se non lo è. Un'altra ramificazione controlla se la stringa di input termina con una configurazione che contiene lo stato di accettazione, q_{accept}, ed accetta, se non lo contiene.

La terza ramificazione dovrebbe accettare se qualche C_i non produce correttamente C_{i+1}. Funziona scandendo l'input fino a quando non decide in maniera non deterministica che è arrivata a C_i. Poi, inserisce C_i nella pila fino a quando non arriva alla fine come contrassegnato dal simbolo #. Quindi D estrae dalla pila per effettuare il confronto con C_{i+1}. Esse devono corrispondere, tranne che attorno alla posizione della testina dove la differenza è determinata dalla funzione di transizione di M. In conclusione, D accetta se individua una mancata corrispondenza o un aggiornamento improprio.

Il problema con questa idea è che, quando D estrae C_i dalla pila, è in ordine inverso e non adatto al confronto con C_{i+1}. A questo punto appare evidente come cambiare la dimostrazione: scriviamo le storie di computazione accettanti in modo diverso. Ogni seconda configurazione viene scritta al contrario. Le posizioni dispari rimangono scritte nell'ordine originale, ma le posizioni pari sono scritte in ordine inverso (all'indietro). Così una storia di computazione accettante appare come mostrato nella figura seguente.

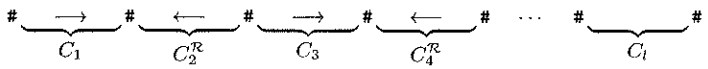


FIGURA 5.14
Ogni seconda configurazione è scritta in ordine inverso

In questa forma modificata, il PDA è in grado di inserire una configurazione in modo tale che, quando viene estratta, l'ordine è adatto per il confronto tra elementi successivi. Progettiamo D in modo tale da accettare qualsiasi stringa che non è una storia di computazione accettante nella forma modificata.

Nell'Esercizio 5.1 potete usare il Teorema 5.13 per mostrare che EQ_{CFG} è indecidibile.

5.2
UN SEMPLICE PROBLEMA INDECIDIBILE

In questa sezione mostriamo che il fenomeno dell'indecidibilità non è limitato a problemi connessi con gli automi. Diamo un esempio di un problema indecidibile che concerne semplici manipolazioni di stringhe. Esso è noto come il ***Problema della Corrispondenza di Post***, o ***PCP***.

Possiamo descrivere questo problema come un tipo di puzzle. Iniziamo con un insieme di tessere del domino, ognuna contenente due stringhe, una su ogni lato. Una tessera è del tipo

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]$$

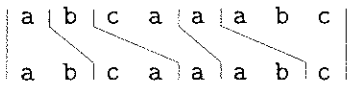
ed un insieme di tessere è del tipo

$$\left\{\left[\frac{\texttt{b}}{\texttt{ca}}\right], \left[\frac{\texttt{a}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{abc}}{\texttt{c}}\right]\right\}.$$

Lo scopo è quello di creare una lista di tessere (con eventuali ripetizioni) in modo tale che la stringa che otteniamo leggendo i simboli sulla metà superiore delle tessere è la stessa che si ottiene leggendo i simboli sulle metà inferiori. Una tale lista è chiamata ***match***. Per esempio la lista seguente è un match per il precedente puzzle.

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{b}}{\texttt{ca}}\right]\left[\frac{\texttt{ca}}{\texttt{a}}\right]\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{abc}}{\texttt{c}}\right]$$

Leggendo la stringa superiore otteniamo abcaaabc, che è uguale a quella inferiore. Possiamo anche rappresentare tale match deformando le tessere in modo tale da allineare i simboli di sopra e di sotto corrispondenti.



Alcuni insiemi di tessere non ammettono un match. Per esempio l'insieme

$$\left\{\left[\frac{\texttt{abc}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{acc}}{\texttt{ba}}\right]\right\}$$

non può ammettere un match perché ogni stringa superiore è più lunga della corrispondente stringa inferiore.

Il Problema della Corrispondenza di Post chiede di determinare se un insieme di tessere ammette un match. Tale problema non è risolvibile mediante un algoritmo.

Prima di dare la formulazione esatta di questo teorema, definiamo il problema in maniera precisa e poi lo esprimiamo come linguaggio. Un'istanza di PCP è un insieme P di tessere

$$P = \left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\},$$

ed un match è una sequenza $i_1, i_2, \ldots, i_l$, dove $t_{i_1} t_{i_2} \cdots t_{i_l} = b_{i_1} b_{i_2} \cdots b_{i_l}$. Il problema consiste nel determinare se P ammette un match. Sia

$$PCP = \{\langle P\rangle \mid P \text{ è un'istanza del Problema della Corrispondenza di Post che ammette un match}\}.$$

TEOREMA 5.15

PCP è indecidibile.

IDEA. Concettualmente è una dimostrazione semplice, anche se necessita di molti dettagli. La tecnica principale consiste in una riduzione da A_{TM} mediante storie di computazione accettanti. Mostriamo che per ogni TM M ed input w, possiamo costruire un'istanza P dove un match è una storia di computazione accettante per M su w. Se potessimo determinare se un'istanza ammette un match, saremmo in grado di determinare se M accetta w o meno.

Come possiamo costruire P in modo tale che un match sia una storia di computazione accettante per M su w? Scegliamo le tessere in P in modo tale che un match forza l'esecuzione di una simulazione di M. Nel match, ogni tessera collega una o più posizioni in una configurazione con la (le) corrispondente nella configurazione successiva.

Prima di passare alla costruzione, dobbiamo considerare tre dettagli tecnici. (Non preoccupatevene eccessivamente in una prima lettura della costruzione). Come prima cosa, per comodità nella costruzione di P, assumiamo che M su w non tenta mai di muovere la testina oltre l'estremità sinistra del nastro. Ciò richiede di modificare preliminarmente M per ottenere tale comportamento. In secondo luogo, se $w = \varepsilon$, usiamo la stringa $\sqcup$ invece di w nella costruzione. Infine, richiediamo che un match inizi con la prima tessera,

$$\left[\frac{t_1}{b_1}\right].$$

In seguito mostreremo come eliminare questa richiesta. Chiamiamo questo problema il Problema della Corrispondenza di Post Modificato (MPCP). Sia

$$MPCP = \{\langle P\rangle \mid P \text{ è un'istanza del Problema della Corrispondenza di Post che ammette un match che inizia con la prima tessera}\}.$$

Ora diamo i dettagli della dimostrazione e progettiamo P in modo da simulare M su w.

DIMOSTRAZIONE Assumiamo che la TM R decide PCP e costruiamo S che decide A_{TM}. Sia

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

dove Q, Σ, Γ, e δ sono l'insieme degli stati, l'alfabeto di input, l'alfabeto di nastro e la funzione di transizione di M, rispettivamente.

In tal caso, S costruisce un'istanza P di PCP che ammette un match sse M accetta w. Per fare ciò, S prima costruisce un'istanza P' di MPCP. Ne descriviamo la costruzione in sette parti, ognuna delle quali riguarda un aspetto della simulazione di M su w. Per spiegare quello che facciamo, alterniamo la costruzione con un esempio di costruzione.

Parte 1. La costruzione inizia come segue.

Poniamo $\left[\frac{\#}{\#q_0w_1w_2\cdots w_n\#}\right]$ in P' come prima tessera $\left[\frac{t_1}{b_1}\right]$.

Poichè P' è un'istanza di MPCP, il match deve iniziare con questa tessera. Quindi la stringa inferiore inizia correttamente con $C_1 = q_0w_1w_2\cdots w_n$, la prima configurazione nella storia di computazione accettante di M su w, come mostrato nella seguente figura.

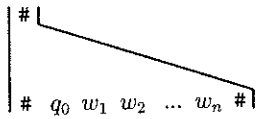


FIGURA 5.16
Inizio del match per MPCP

In questa descrizione del match parziale ottenuto finora, la stringa inferiore è $\#q_0w_1w_2\cdots w_n\#$ e quella superiore è semplicemente #. Per ottenere un match, dobbiamo estendere la stringa superiore in modo da coincidere con quella inferiore. A tal fine, forniamo altre tessere. Le ulteriori tessere fanno sì che la successiva configurazione di M appaia con l'estensione della stringa inferiore forzando la simulazione di un passo di M.

Nelle parti 2,3 e 4, aggiungiamo a P' delle tessere che realizzano la parte principale della simulazione. La parte 2 gestisce il movimento della testina a destra, la parte 3 gestisce il il movimento della testina a sinistra, la parte 4 gestisce le celle del nastro non adiacenti alla testina.

Parte 2. Per ogni $a, b \in \Gamma$ ed ogni $q, r \in Q$ dove $q \neq q_{\text{reject}}$,

$$\text{se } \delta(q, a) = (r, b, \text{R}), \text{ inserisci } \left[\frac{qa}{br}\right] \text{ in } P'.$$

Parte 3. Per ogni $a, b, c \in \Gamma$ ed ogni $q, r \in Q$ dove $q \neq q_{\text{reject}}$,

$$\text{se } \delta(q, a) = (r, b, \text{L}), \text{ inserisci } \left[\frac{cqa}{rcb}\right] \text{ in } P'.$$

Parte 4. Per ogni $a \in \Gamma$,

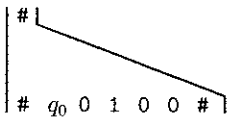
$$\text{inserisci } \left[\frac{a}{a}\right] \text{ in } P'.$$

Ora creiamo un esempio ipotetico per illustrare quello che abbiamo costruito finora. Sia $\Gamma = \{0, 1, 2, \sqcup\}$. Diciamo che w è la stringa 0100 e che lo stato iniziale di M è q_0. Nello stato q_0, leggendo 0, supponiamo che la funzione di transizione dice che M va nello stato q_7, scrive 2 e muove la testina a destra. In altre parole $\delta(q_0, 0) = (q_7, 2, \text{R})$.

La parte 1 inserisce la tessera

$$\left[\frac{\#}{\#q_00100\#}\right] = \left[\frac{t_1}{b_1}\right]$$

in P', e il match inizia.



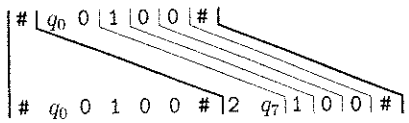
Poi la parte 2 inserisce la tessera

$$\left[\frac{q_00}{2q_7}\right]$$

poichè $\delta(q_0, 0) = (q_7, 2, \text{R})$ e la parte 4 inserisce le tessere

$$\left[\frac{0}{0}\right], \left[\frac{1}{1}\right], \left[\frac{2}{2}\right], \text{ e } \left[\frac{\sqcup}{\sqcup}\right]$$

in P', poichè 0, 1, 2 e $\sqcup$ sono elementi di Γ. Insieme alla parte 5, ciò ci permette di estendere il match a



Quindi le tessere delle parti 2,3 e 4 ci permettono di estendere il match aggiungendo la seconda configurazione dopo la prima. Vogliamo che questo processo continui, con l'aggiunta della terza configurazione, poi della quarta, e così via. Affinché ciò avvenga, abbiamo bisogno di aggiungere una o più tessere per copiare il simbolo #.

Parte 5.

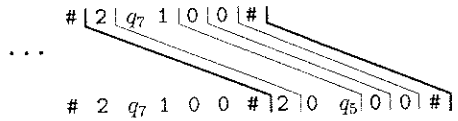
$$\text{Inserisci } \left[\frac{\#}{\#}\right] \text{ e } \left[\frac{\#}{\sqcup\#}\right] \text{ in } P'.$$

La prima di queste tessere ci permette di copiare il simbolo # che separa le configurazioni. Inoltre, la seconda tessera ci permette di aggiungere il simbolo $\sqcup$ alla fine della configurazione così da simulare gli infiniti blank sulla parte destra del nastro che sono soppressi quando scriviamo la configurazione.

Continuando con l'esempio, diciamo che nello stato q_7, leggendo 1, M va nello stato q_5, scrive 0 e muove la testina a destra. Cioè $\delta(q_7, 1) = (q_5, 0, \mathrm{R})$. Quindi abbiamo la tessera

$$\left[\frac{q_7 1}{0 q_5}\right] \text{ in } P'.$$

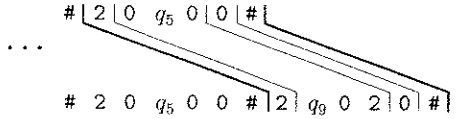
Di conseguenza, il match parziale viene esteso a



Ora, supponiamo che nello stato q_5, leggendo 0, M si sposta nello stato q_9, scrive 2, e muove la testina a sinistra. Cioè $\delta(q_5, 0) = (q_9, 2, \mathrm{L})$. Quindi abbiamo le tessere

$$\left[\frac{0 q_5 0}{q_9 0 2}\right], \left[\frac{1 q_5 0}{q_9 1 2}\right], \left[\frac{2 q_5 0}{q_9 2 2}\right], \text{ e } \left[\frac{\sqcup q_5 0}{q_9 \sqcup 2}\right].$$

La prima è importante perché il simbolo a sinistra della testina è 0. Il match viene esteso a

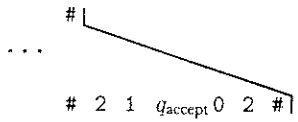


Notate che mentre costruiamo un match, siamo forzati a simulare M su input w. Questo processo continua fino a quando M raggiunge uno stato di arresto. Se tale stato è di accettazione, vogliamo che la stringa superiore venga a coincidere con quella inferiore così da completare il match. Possiamo far sì che ciò accada aggiungendo alcune tessere.

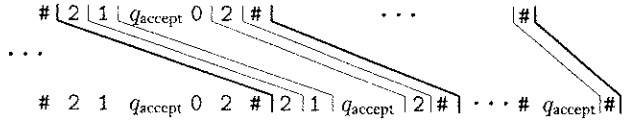
Parte 6. Per ogni $a \in \Gamma$,

$$\text{inserisci } \left[\frac{a\, q_{\text{accept}}}{q_{\text{accept}}}\right] \text{ e } \left[\frac{q_{\text{accept}}\, a}{q_{\text{accept}}}\right] \text{ in } P'.$$

Questo passo ha l'effetto di aggiungere "pseudo-passi" della macchina di Turing dopo che essa si è fermata, dove la testina "mangia" simboli adiacenti fino a che non ne rimane alcuno. Continuando con l'esempio, se il match parziale quando la macchina si ferma nello stato di accettazione è



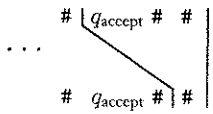
Le tessere appena aggiunte permettono di continuare il match:



Parte 7. Infine aggiungiamo la tessera

$$\left[\frac{q_{\text{accept}} \# \#}{\#}\right]$$

e completiamo il match:



Questo conclude la costruzione di P'. Ricordate che P' è un'istanza di MPCP dove il match simula la computazione di M su w. Per completare la dimostrazione, ricordiamo che MPCP differisce da PCP per la richiesta che il match inizi con la prima tessera della lista. Se guardiamo a P' come a un'istanza di PCP invece che di MPCP, esso ammette sicuramente un match indipendentemente dal fatto che M accetta w o meno. Vedete il perché? (Sugg. è molto breve.)

Ora mostriamo come trasformare P' in P, un'istanza di PCP che ancora simula M su w. Lo facciamo utilizzando uno stratagemma tecnico. L'idea è di utilizzare la richiesta che il match inizi con la prima tessera ed inglobarla direttamente nell'istanza del problema così che sia automaticamente soddisfatta. A tal punto la richiesta non è piu necessaria. Introduciamo la notazione necessaria all'implementazione di tale idea.

Sia $u = u_1u_2 \cdots u_n$ una stringa di lunghezza n. Definiamo $\star u$, $u\star$ e $\star u\star$ come le tre stringhe

$$\begin{aligned} \star u &= *u_1 * u_2 * u_3 * \cdots *u_n \\ u\star &= u_1 * u_2 * u_3 * \cdots *u_n * \\ \star u\star &= *u_1 * u_2 * u_3 * \cdots *u_n *. \end{aligned}$$

Qui $\star u$ aggiunge il simbolo $*$ prima di ogni carattere in u, $u\star$ ne aggiunge uno dopo ogni carattere in u e $\star u\star$ ne aggiunge uno sia prima che dopo ogni carattere in u.

Per trasformare P' in P, un'istanza di PCP, procediamo come segue. Se P' è l'insieme

$$\left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \left[\frac{t_3}{b_3}\right], \cdots, \left[\frac{t_k}{b_k}\right]\right\},$$

prendiamo P come l'insieme

$$\left\{\left[\frac{\star t_1}{\star b_1\star}\right], \left[\frac{\star t_1}{b_1\star}\right], \left[\frac{\star t_2}{b_2\star}\right], \left[\frac{\star t_3}{b_3\star}\right], \cdots, \left[\frac{\star t_k}{b_k\star}\right], \left[\frac{*\diamond}{\diamond}\right]\right\}.$$

Considerando P come un'istanza di PCP, vediamo che l'unica tessera che potrebbe eventualmente essere quella iniziale del match è proprio la prima

$$\left[\frac{\star t_1}{\star b_1\star}\right],$$

poichè è la sola in cui sia la parte superiore che quella inferiore iniziano con uno stesso simbolo – precisamente $*$. Oltre a far sì che il match inizi con la prima tessera, la presenza dei caratteri $*$ non influisce sull'esistenza di un match perché essi si limitano ad alternarsi con i simboli originali. I simboli originali ora si trovano nelle posizioni pari del match. La tessera

$$\left[\frac{*\diamond}{\diamond}\right]$$

è presente per permettere alla parte superiore di aggiungere l'ulteriore $*$ alla fine del match.

5.3
RIDUCIBILITÀ MEDIANTE FUNZIONE

Abbiamo mostrato come utilizzare la tecnica della riducibilità per dimostrare che alcuni problemi sono indecidibili. In questa sezione formalizziamo il concetto di riducibilità. Questo ci permette di usare la riducibilità in modo più raffinato, come per esempio per dimostrare che alcuni linguaggi non sono Turing-riconoscibili e per applicazioni in teoria della complessità. La nozione di ridurre un problema ad un altro può essere definita formalmente in vari modi. La scelta di quale usare dipende dall'applicazione. La nostra scelta ricade su un tipo semplice di riducibilità chiamato ***riducibilità mediante funzione***.[1]

Informalmente, essere in grado di ridurre il problema A al problema B utilizzando una riduzione mediante funzione significa che esiste una funzione calcolabile che trasforma istanze del problema A in istanze del problema B. Se abbiamo una tale funzione, detta *riduzione*, siamo in grado di risolvere A risolvendo istanze di B. Il motivo risiede nel fatto che una qualsiasi istanza di A può essere risolta utilizzando prima la riduzione per convertirla in un'istanza di B e poi risolvere tale istanza di B. Nel breve daremo una definizione precisa di riducibilità mediante funzione.

Funzioni calcolabili

Una macchina di Turing calcola una funzione iniziando con l'input della funzione sul nastro e terminando con l'output della funzione sul nastro.

DEFINIZIONE **5.17**

Una funzione $f: \Sigma^* \longrightarrow \Sigma^*$ è una ***funzione calcolabile*** se esiste una macchina di Turing M che, su qualsiasi input w, si ferma avendo solo $f(w)$ sul nastro.

ESEMPIO 5.18

Tutte le operazioni aritmetiche usuali su numeri interi sono funzioni calcolabili. Per esempio, possiamo progettare una macchina che prende in input

[1] È chiamata ***riducibilità molti-a-uno*** in altri libri di testo.

$\langle m, n \rangle$ e restituisce $m + n$, la somma di m ed n. Non diamo dettagli qui, li lasciamo come esercizio.

ESEMPIO 5.19

Le funzioni calcolabili possono essere trasformazioni di descrizioni di macchine. Per esempio, una funzione calcolabile f prende in input w e restituisce la descrizione di una macchina di Turing $\langle M' \rangle$ se $w = \langle M \rangle$ è la codifica di una macchina di Turing M. La macchina M' è una macchina che riconosce lo stesso linguaggio di M, ma non tenta mai di muovere la testina oltre il limite sinistro del suo nastro. La funzione f realizza questo compito con l'aggiunta di vari stati nella descrizione di M. La funzione restituisce ϵ se w non è una codifica legale di una macchina di Turing.

Definizione formale di riducibilità mediante funzione

Definiamo ora la riducibilità mediante funzione. Come al solito, rappresentiamo problemi computazionali mediante linguaggi.

DEFINIZIONE **5.20**

Un linguaggio A si dice ***riducibile mediante funzione*** al linguaggio B, e si denota con $A \leq_{\mathrm{m}} B$, se esiste una funzione calcolabile $f\colon \Sigma^* \longrightarrow \Sigma^*$, dove per ogni w,

$$w \in A \iff f(w) \in B.$$

La funzione f è chiamata ***riduzione*** da A a B.

La figura seguente illustra la riducibilità mediante funzione.

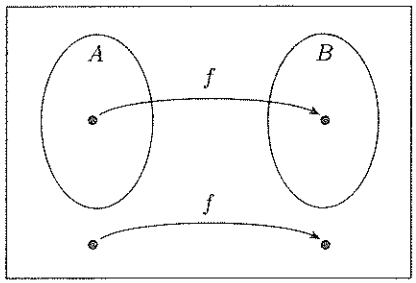


FIGURA 5.21
La funzione f riduce A a B

Una riduzione mediante funzione da A a B fornisce un modo per convertire problemi di appartenenza ad A in problemi di appartenenza a B. Per verificare se $w \in A$, usiamo la riduzione f per mappare w in $f(w)$ e verifichiamo se $f(w) \in B$ o meno. Il termine *riduzione mediante funzione* deriva dalla funzione che viene utilizzata per la riduzione.

Se un problema è riducibile mediante funzione ad un secondo problema, già precedentemente risolto, possiamo ottenere da questo una soluzione al problema originale. Catturiamo quest'idea nel Teorema 5.22.

TEOREMA 5.22

Se $A \leq_{\mathrm{m}} B$ e B è decidibile, allora A è decidibile.

DIMOSTRAZIONE. Siano M il decisore per B ed f la riduzione da A a B. Descriviamo un decisore N per A come segue.

N = "Su input w:

1. Computa $f(w)$.
2. Esegue M su input $f(w)$ e restituisce lo stesso output di M."

Ovviamente, se $w \in A$, allora $f(w) \in B$ perché f è una riduzione da A a B. Quindi, M accetta $f(w)$ ogni volta che $w \in A$. Quindi, N svolge il compito desiderato.

Il seguente corollario al Teorema 5.22 rappresenta il nostro strumento principale nelle dimostrazioni di indecidibilità.

COROLLARIO 5.23

Se $A \leq_{\mathrm{m}} B$ e A è indecidibile, allora B è indecidibile.

Ora rivediamo alcune delle nostre prove precedenti che hanno utilizzato il metodo della riducibilità per ottenere esempi di riduzioni mediante funzione.

ESEMPIO 5.24

Nel Teorema 5.1 abbiamo usato una riduzione da A_{TM} per dimostrare che $HALT_{\mathsf{TM}}$ è indecidibile. Questa riduzione ha mostrato come un decisore per $HALT_{\mathsf{TM}}$ potrebbe essere utilizzato per ottenere un decisore per A_{TM}. Possiamo fornire una riduzione mediante funzione da A_{TM} a $HALT_{\mathsf{TM}}$ come segue. Per farlo, dobbiamo fornire una funzione calcolabile f che prende in input $\langle M, w \rangle$ e restituisce in output $\langle M', w' \rangle$, dove

$$\langle M, w \rangle \in A_{\mathsf{TM}} \text{ se e solo se } \langle M', w' \rangle \in HALT_{\mathsf{TM}}.$$

La seguente macchina F calcola una riduzione f.

F = "Su input $\langle M, w\rangle$:

1. Costruisce la seguente macchina M'.
   M' = "Su input x:
   1. Esegue M su x.
   2. Se M accetta, *accetta*.
   3. Se M rifiuta, cicla."
2. Output $\langle M', w\rangle$."

Qui sorge un problema minore che riguarda le stringhe in input che non sono della forma corretta. Se la TM F determina che il suo input non è della forma corretta, in accordo a quanto specificato dalla linea dell'input "Su input $\langle M, w\rangle$:"e, quindi, che l'input non è in A_{TM}, la TM dà in output una stringa non in $HALT_{\mathsf{TM}}$. Può essere una qualsiasi stringa non in $HALT_{\mathsf{TM}}$. In generale, quando descriviamo una macchina di Turing che calcola una riduzione da A a B, assumiamo che gli input che non sono della forma corretta sono mappati in stringhe al di fuori di B.

ESEMPIO 5.25

La dimostrazione dell'indecidibilità del Problema della Corrispondenza di Post nel Teorema 5.15 contiene due riduzioni mediante funzione. Prima, dimostra che $A_{\mathsf{TM}} \leq_{\mathrm{m}} MPCP$ e poi mostra che $MPCP \leq_{\mathrm{m}} PCP$. In entrambi i casi siamo in grado di ottenere facilmente l'effettiva funzione di riduzione e mostrare che si tratta di una riduzione mediante funzione. Come mostra l'Esercizio 5.6, la riducibilità mediante funzione è transitiva, per cui queste due riduzioni insieme implicano $A_{\mathsf{TM}} \leq_{\mathrm{m}} PCP$.

ESEMPIO 5.26

Una riduzione mediante funzione da E_{TM} a EQ_{TM} si trova nella dimostrazione del Teorema 5.4. In questo caso la riduzione f mappa l'input $\langle M\rangle$ nell'output $\langle M, M_1\rangle$, dove M_1 è la macchina che rifiuta ogni input.

ESEMPIO 5.27

La dimostrazione del Teorema 5.2, mostrando che E_{TM} è indecidibile, illustra la differenza tra la nozione formale di riducibilità mediante funzione che abbiamo definito in questa sezione e la nozione informale di riducibilità che abbiamo usato in precedenza in questo capitolo. La dimostrazione prova che E_{TM} è indecidibile riducendo A_{TM} ad esso. Vediamo se siamo in grado di riscrivere questa riduzione come una riduzione mediante funzione.

Dalla riduzione originale possiamo facilmente costruire una funzione f che prende in input $\langle M, w\rangle$ e produce un output $\langle M_1\rangle$, dove M_1 è la macchina di Turing descritta in quella dimostrazione. Ma M accetta w sse

$L(M_1)$ è *non* vuoto, quindi f è una riduzione mediante funzione da A_{TM} a $\overline{E_{\mathsf{TM}}}$. Essa ancora dimostra che E_{TM} è indecidibile in quanto la decidibilità non è influenzata dalla complementazione, ma non fornisce una riduzione mediante funzione da A_{TM} a E_{TM}. Infatti, una tale riduzione non esiste, come vi viene chiesto di dimostrare nell'Esercizio 5.5.

La sensitività delle riduzioni mediante funzione alla complementazione è importante nell'uso della riducibilità per dimostrare la non-riconoscibilità di alcuni linguaggi. Possiamo anche utilizzare la riducibilità mediante funzione per dimostrare che alcuni problemi non sono Turing-riconoscibili. Il seguente teorema è analogo al Teorema 5.22.

TEOREMA 5.28

Se $A \leq_{\mathrm{m}} B$ e B è Turing-riconoscibile, allora A è Turing-riconoscibile.

La dimostrazione è uguale a quella che abbiamo dato per il Teorema 5.22, tranne per il fatto che M ed N sono riconoscitori invece di decisori.

COROLLARIO 5.29

Se $A \leq_{\mathrm{m}} B$ e A non è Turing-riconoscibile, allora B non è Turing-riconoscibile.

In una tipica applicazione di questo corollario, supponiamo che A sia $\overline{A_{\mathsf{TM}}}$, il complemento di A_{TM}. Dal Corollario 4.23, sappiamo che non è Turing-riconoscibile. La definizione di riduzione mediante funzione implica che $A \leq_{\mathrm{m}} B$ ha lo stesso significato di $\overline{A} \leq_{\mathrm{m}} \overline{B}$. Per dimostrare che B non è riconoscibile possiamo mostrare che $A_{\mathsf{TM}} \leq_{\mathrm{m}} \overline{B}$. Possiamo anche usare la riducibilità mediante funzione per dimostrare che alcuni problemi non sono nè Turing-riconoscibili nè co-Turing-riconoscibili, come nel seguente teorema.

TEOREMA 5.30

EQ_{TM} non è nè Turing-riconoscibile nè co-Turing-riconoscibile.

DIMOSTRAZIONE. Come primo passo, dimostriamo che EQ_{TM} non è Turing-riconoscibile. Lo facciamo dimostrando che A_{TM} è riducibile a $\overline{EQ_{\mathsf{TM}}}$. La funzione di riduzione f opera come segue.

F = "Su input $\langle M, w\rangle$, dove M è una TM e w è una stringa:

1. Costruisce le seguenti due macchine, M_1 e M_2.
   M_1 = "Su ogni input:
   1. *Rifiuta*."
   M_2 = "Su ogni input:
   1. Esegue M su w. Se accetta, *accetta*."
2. Restituisce $\langle M_1, M_2\rangle$."

Qui, M_1 non accetta alcuna stringa. Se M accetta w, M_2 accetta qualsiasi stringa, quindi le due macchine non sono equivalenti. Viceversa, se M non accetta w, M_2 non accetta alcuna stringa, ed esse sono equivalenti. Perciò f riduce A_{TM} a $\overline{EQ_{\mathsf{TM}}}$, come desiderato.

Per dimostrare che $\overline{EQ_{\mathsf{TM}}}$ non è Turing-riconoscibile diamo una riduzione da A_{TM} al complemento di $\overline{EQ_{\mathsf{TM}}}$, cioè, EQ_{TM}. In questo modo dimostriamo che $A_{\mathsf{TM}} \leq_{\mathrm{m}} EQ_{\mathsf{TM}}$. La seguente TM G calcola la funzione di riduzione g.

G = "Su input $\langle M, w\rangle$, dove M è una TM e w è una stringa:

1. Costruisce le seguenti due macchine, M_1 ed M_2.
   M_1 = "Su ogni input:
   1. *Accetta.*"
   M_2 = "Su ogni input:
   1. Esegue M su w.
   2. Se accetta, *accetta.*"
2. Restituisce $\langle M_1, M_2\rangle$."

La sola differenza tra f e g si trova nella macchina M_1. In f, la macchina M_1 rifiuta sempre, mentre in g accetta sempre. Sia in f che in g, M accetta w sse M_2 accetta ogni input. In g, M accetta w sse M_1 ed M_2 sono equivalenti. Questo è il motivo per cui g è una riduzione da A_{TM} a EQ_{TM}.

ESERCIZI

5.1 Mostrare che EQ_{CFG} è indecidibile

5.2 Mostrare che EQ_{CFG} è coTuring-riconiscibile.

5.3 Trovare un match nella seguente istanza del Problema della Corrispondenza di Post.

$$\left\{\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right], \left[\frac{\mathtt{b}}{\mathtt{a}}\right], \left[\frac{\mathtt{aba}}{\mathtt{b}}\right], \left[\frac{\mathtt{aa}}{\mathtt{a}}\right]\right\}$$

5.4 $A \leq_{\mathrm{m}} B$ e B è un linguaggio regolare, ciò implica che A è un linguaggio regolare? Perché sì o perché no?

A5.5 Mostrare che A_{TM} non è riducibile mediante funzione a E_{TM}. In altre parole, mostrare che nessuna funzione calcolabile riduce A_{TM} ad E_{TM}. (Suggerimento: utilizzare una dimostrazione per assurdo e fatti noti circa A_{TM} e E_{TM}.)

A5.6 Mostrare che $\leq_{\mathrm{m}}$ è una relazione transitiva.

A5.7 Mostrare che se A è Turing-riconoscibile e $A \leq_{\mathrm{m}} \overline{A}$, allora A è decidibile.

A5.8 Nella dimostrazione del Teorema 5.15, abbiamo modificato la macchina di Turing M in modo che non cerca mai di muovere la testina oltre il limite sinistro del nastro. Supponiamo di non aver fatto questa modifica a M. Modificare la costruzione per PCP per gestire questo caso.

PROBLEMI

5.9 Sia $AMBIG_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ è una CFG ambigua}\}$. Mostrare che $AMBIG_{\mathsf{CFG}}$ è indecidibile. (Suggerimento: usare una riduzione da *PCP*. Data un'istanza

$$P = \left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\}$$

del Problema della Corrispondenza di Post costruire una CFG G con le regole

$$\begin{aligned} S &\to T \mid B \\ T &\to t_1 T \mathtt{a}_1 \mid \cdots \mid t_k T \mathtt{a}_k \mid t_1 \mathtt{a}_1 \mid \cdots \mid t_k \mathtt{a}_k \\ B &\to b_1 B \mathtt{a}_1 \mid \cdots \mid b_k B \mathtt{a}_k \mid b_1 \mathtt{a}_1 \mid \cdots \mid b_k \mathtt{a}_k, \end{aligned}$$

dove $\mathtt{a}_1, \ldots, \mathtt{a}_k$ sono simboli terminali. Mostrare la correttezza di questa riduzione.)

5.10 Mostrare che A è Turing-riconoscibile sse $A \leq_{\mathrm{m}} A_{\mathsf{TM}}$.

5.11 Mostrare che A è decidibile sse $A \leq_{\mathrm{m}} \mathtt{0}^*\mathtt{1}^*$.

5.12 Sia $J = \{w \mid \text{o } w = \mathtt{0}x \text{ per qualche } x \in A_{\mathsf{TM}} \text{ oppure } w = \mathtt{1}y \text{ per qualche } y \in \overline{A_{\mathsf{TM}}}\}$. Mostrare che sia J sia $\overline{J}$ non sono Turing-riconoscibili.

5.13 Fornire un esempio di un linguaggio indecidibile B, tale che $B \leq_{\mathrm{m}} \overline{B}$.

5.14 Definiamo un ***automa finito a due testine*** (2DFA) come un automa finito deterministico a due nastri di sola lettura con testine che si muovono in entrambe le direzioni e che iniziano all'estremo sinistro del nastro di input e che possono essere controllate in modo indipendente per farle spostare in entrambe le direzioni. Il nastro di una 2DFA è finito ed è solo sufficiente a contenere l'input più due celle vuote addizionali, una sul lato sinistro e una sul lato destro, che fungono da delimitatori. Un 2DFA accetta il suo input occupando uno stato speciale di accettazione. Ad esempio, un 2DFA può riconoscere il linguaggio $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$.

a. Sia $A_{\mathsf{2DFA}} = \{\langle M, x\rangle \mid M \text{ è un 2DFA e } M \text{ accetta } x\}$. Mostrare che A_{2DFA} è decidibile.

b. Sia $E_{\mathsf{2DFA}} = \{\langle M\rangle \mid M \text{ è un 2DFA e } L(M) = \emptyset\}$. Mostrare che E_{2DFA} non è decidibile.

5.15 Un ***automa finito bidimensionale*** (2DIM-DFA) è definito come segue. L'input è un rettangolo $m \times n$, per qualche $m, n \geq 2$. I quadrati lungo il bordo del rettangolo contengono il simbolo # e quelli interni contengono simboli di input sull'alfabeto Σ. La funzione di transizione $\delta\colon Q \times (\Sigma \cup \{\#\}) \longrightarrow Q \times \{\mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{D}\}$ indica lo stato successivo e la nuova posizione della testina (Sinistra, Destra, Su, Giù). La macchina accetta quando entra in uno degli stati designati di accettazione. Rifiuta se cerca di spostarsi fuori dal rettangolo di input o se non si ferma mai. Due tali macchine sono equivalenti se accettano gli stessi rettangoli. Si consideri il problema di determinare se due di queste macchine sono equivalenti. Formulare il problema come un linguaggio e dimostrare che è indecidibile.

A*5.16 **Teorema di Rice**. Sia P una qualsiasi proprietà non banale del linguaggio di una macchina di Turing. Dimostrare che il problema di determinare se il linguaggio di una data macchina di Turing ha la proprietà P è indecidibile. In termini più formali, sia P un linguaggio di descrizioni di macchine di Turing che soddisfa due condizioni. Primo, P è non banale – contiene alcune, ma non tutte, le descrizioni delle TM. Secondo, P è una proprietà del linguaggio della TM– ogni volta che $L(M_1) = L(M_2)$, abbiamo $\langle M_1 \rangle \in P$ sse $\langle M_2 \rangle \in P$. Qui, M_1 ed M_2 sono macchine di Turing. Dimostrare che P è un linguaggio indecidibile.

5.17 Mostrare che entrambe le condizioni del Problema 5.16 sono necessarie per provare che P è indecidibile.

5.18 Usare il Teorema di Rice, descritto nel Problema 5.16, per dimostrare l'indecidibilità di ciascuno dei seguenti linguaggi.

A**a.** $INFINITE_{\mathsf{TM}} = \{\langle M \rangle | \ M$ è una TM ed $L(M)$ è un linguaggio infinito$\}$.

b. $\{\langle M \rangle | \ M$ è una TM e $1011 \in L(M)\}$.

c. $ALL_{\mathsf{TM}} = \{\langle M \rangle | \ M$ è una TM e $L(M) = \Sigma^*\}$.

5.19 Sia

$$f(x) = \begin{cases} 3x+1 & \text{se } x \text{ è dispari} \\ x/2 & \text{se } x \text{ è pari} \end{cases}$$

per ogni numero naturale x. Se partite con un intero x ed iterate f, ottenete una sequenza, $x, f(x), f(f(x)), \ldots$. Fermatevi se ad una qualche iterazione ottenete 1. Ad esempio, se $x = 17$, ottenete la sequnza 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Numerosi test al calcolatore hanno mostrato che ogni punto di partenza tra 1 ed un intero positivo grande produce una sequenza che termina in 1. Ma stabilire se tutti i numeri di partenza portano ad 1 è un problema aperto, chiamato il problema $3x + 1$.

Si supponga che A_{TM} sia decidibile da una TM H. Usare H per descrivere una TM che sia in grado di dare una risposta al problema $3x + 1$.

5.20 Dimostrare che i due linguaggi seguenti sono indecidibili.

a. $OVERLAP_{\mathsf{CFG}} = \{\langle G, H \rangle | \ G$ e H sono CFG dove $L(G) \cap L(H) \neq \emptyset\}$. (Suggerimento: adattare il suggerimento del Problema 5.9.)

b. $PREFIX\text{-}FREE_{\mathsf{CFG}} = \{\langle G \rangle | \ G$ è una CFG tale che $L(G)$ è prefisso$\}$.

5.21 Considerare il problema di determinare se un PDA accetta almena almeno una stringa della forma $\{ww | w \in \{0,1\}^*\}$. Utilizzare il metodo delle storie di computazione per mostrare che questo problema è indecidibile.

5.22 Sia $X = \{\langle M, w \rangle | \ M$ è una TM a nastro singolo che non modifica la porzione di nastro che contiene l'input $w\}$. X è decidibile? Dimostrare la vostra risposta.

5.23 Una variabile A in CFG G è detta ***necessaria*** se essa appare in ogni derivazione di una qualche stringa $w \in G$. Sia $NECESSARY_{\mathsf{CFG}} = \{\langle G, A \rangle | \ A$ è una variabile necessaria in $G\}$.

a. Mostrare che $NECESSARY_{\mathsf{CFG}}$ è Turing-riconoscibile.

b. Mostrare che $NECESSARY_{\mathsf{CFG}}$ è indecidibile.

*5.24 Una CFG è detta *minimale* se nessuna delle sue regole può essere eliminata senza modificare il linguaggio generato. Sia $MIN_{\mathsf{CFG}} = \{\langle G \rangle | \ G$ è una CFG minimale$\}$.

a. Mostrare che MIN_{CFG} è Turing-riconoscibile.

b. Mostrare che MIN_{CFG} è indecidibile.

5.25 Sia $T = \{\langle M \rangle | \ M$ è una TM che accetta $w^{\mathcal{R}}$ ogniqualvolta essa accetta $w\}$. Mostrare che T è indecidibile.

A**5.26** Considerare il problema di determinare se una macchina di Turing a due nastri eventualmente scrive un simbolo diverso dal blank sul suo secondo nastro quando funziona sull'input w. Formulare questo problema come un linguaggio e mostrare che esso è indecidibile.

A**5.27** Considerare il problema di determinare se una macchina di Turing a due nastri eventualmente scrive un simbolo diverso dal blank sul suo secondo nastro durante la computazione su un qualsiasi input. Formulare questo problema come un linguaggio e mostrare che esso è indecidibile.

5.28 Considerare il problema di determinare se una macchina di Turing a singolo nastro eventualmente scrive il simbolo blank su un simbolo diverso dal blank durante la computazione su un qualsiasi input. Formulare questo problema come un linguaggio e mostrare che esso è indecidibile.

5.29 Uno ***stato inutile*** in una macchina di Turing è uno stato in cui la macchina non entra mai, qualsiasi sia la stringa di input. Considerare il problema di determinare se una macchina di Turing ha uno stato inutile. Formulare questo problema come un linguaggio e mostrare che esso è indecidibile.

5.30 Considerare il problema di determinare se una macchina di Turing M su input w prova mai a muovere la testina a sinistra quando questa si trova all'estremità sinistra del nastro. Formulare questo problema come un linguaggio e mostrare che esso è indecidibile.

5.31 Considerare il problema di determinare se una macchina di Turing M prova mai a muovere la testina a sinistra durante una computazione su un dato input w. Formulare questo problema come un linguaggio e mostrare che esso è decidibile.

5.32 Sia $\Gamma = \{0, 1, \sqcup\}$ l'alfabeto del nastro di ogni TM in questo problema. Definire la ***funzione del castoro*** $BB : \mathcal{N} \longrightarrow \mathcal{N}$ come segue. Per ogni valore di k, considerare tutte le macchine a k stati che si fermano quando partono con il nastro vuoto. Sia $BB(k)$ il massimo numero di simboli 1 che rimangono sul nastro di una di queste macchine. Mostrare che BB non è una funzione computabile.

5.33 Mostrare che il Problema della Corrispondenza di Post è decidibile sull'alfabeto unario $\Sigma = \{1\}$.

5.34 Mostrare che il Problema della Corrispondenza di Post è indecidibile sull'alfabeto binario $\Sigma = \{0,1\}$.

5.35 Nel *Problema della Corrispondenza di Post sciocco*, *SPCP*, per ogni coppia la stringa superiore ha la stessa lunghezza di quella inferiore. Mostrare che *SPCP* è decidibile.

5.36 Mostrare che $\{1\}^*$ contiene un sottoinsieme indecidibile.

SOLUZIONI SELEZIONATE

5.5 Supponiamo per assurdo che $A_{\mathsf{TM}} \leq_{\mathrm{m}} E_{\mathsf{TM}}$ tramite la riduzione f. Dalla definizione di riducibilità mediante funzione segue che $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{E_{\mathsf{TM}}}$ tramite la stessa funzione di riduzione f. Tuttavia $\overline{E_{\mathsf{TM}}}$ è Turing-riconoscibile (vedere la

soluzione dell'Esercizio 4.5) e $\overline{A_{\mathsf{TM}}}$ non è Turing-riconoscibile, contraddicendo il Teorema 5.28.

5.6 Supponiamo che $A \leq_{\mathrm{m}} B$ e $B \leq_{\mathrm{m}} C$. Quindi esistono funzioni computabili f e g tali che $x \in A \iff f(x) \in B$ e $y \in B \iff g(y) \in C$. Si consideri la composizione di funzioni $h(x) = g(f(x))$. Possiamo costruire una TM che calcola h come segue. Come prima cosa, simuliamo una TM per f (essa esiste perché abbiamo supposto che f è computabile) su input x e chiamiamo l'output y. Poi simuliamo una TM per g su y. L'output è $h(x) = g(f(x))$. Quindi h è una funzione calcolabile. Inoltre, $x \in A \iff h(x) \in C$. Quindi $A \leq_{\mathrm{m}} C$ mediante la funzione di riduzione h.

5.7 Supponiamo che $A \leq_{\mathrm{m}} \overline{A}$. Allora $\overline{A} \leq_{\mathrm{m}} A$ tramite la stessa riduzione mediante funzione. Poichè A è Turing-riconoscibile, il Teorema 5.28 implica che $\overline{A}$ è Turing-riconoscibile, e quindi il Teorema 4.22 implica che A è decidibile.

5.8 Bisogna gestire il caso in cui la testina si trova nella cella all'estremità sinistra del nastro e cerca di spostarsi a sinistra. Per far questo aggiungiamo le tessere

$$\left[\begin{array}{c}\#qa\\ \hline \#rb\end{array}\right]$$

per ogni $q, r \in Q$ and $a, b \in \Gamma$, $\delta(q, a) = (r, b, \mathrm{L})$. Inoltre sostituiamo la prima tessera con

$$\left[\begin{array}{c}\#\\ \hline \#\#q_0 w_1 w_2 \cdots w_n\end{array}\right]$$

per gestire il caso di un tentativo di spostamento a sinistra alla prima mossa.

5.16 Assumiamo per assurdo che P sia un linguaggio decidibile che soddisfa le proprietà e sia R_P una TM che decide P. Mostriamo come decidere A_{TM} usando R_P costruendo una TM S. Sia $T_\emptyset$ una TM che rifiuta qualsiasi input, per cui $L(T_\emptyset) = \emptyset$. Possiamo assumere che $\langle T_\emptyset \rangle \notin P$, senza perdita di generalità, in quanto si potrebbe procedere con $\overline{P}$ invece di P nel caso $\langle T_\emptyset \rangle \in P$. Poichè P è non banale, esiste una TM T con $\langle T \rangle \in P$. Progettiamo S in modo tale da decidere A_{TM} utilizzando la capacità di R_P di distinguere tra $T_\emptyset$ e T.

S = "Su input $\langle M, w \rangle$:

1. Usa M e w per costruire la seguente TM M_w.
   M_w = "Su input x:
   1. Simula M su w. Se si ferma e rifiuta, *rifiuta*.
      Se accetta, va al passo 2.
   2. Simula T su x. Se accetta, *accetta*."
2. Usa la TM R_P per determinare se $\langle M_w \rangle \in P$. Se SI, *accetta*.
   Se NO, *rifiuta*."

La TM M_w simula T se M accetta w. Quindi $L(M_w)$ è uguale a $L(T)$ se M accetta w e $\emptyset$ altrimenti. Quindi, $\langle M_w \rangle \in P$ sse M accetta w.

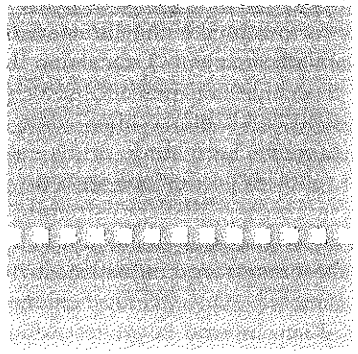
5.18 **(a)** $INFINITE_{\mathsf{TM}}$ è un linguaggio i cui elementi sono descrizioni di macchine di Turing. Esso soddisfa le due condizioni del teorema di Rice. In primo luogo, è non banale perché il linguaggio di alcune macchine è infinito e di altre no. In secondo luogo, esso dipende solo dal linguaggio. Se due macchine riconoscono lo stesso linguaggio, o entrambe hanno descrizioni in $INFINITE_{\mathsf{TM}}$ oppure nessuna delle due. Di conseguenza, il teorema di Rice implica che $INFINITE_{\mathsf{TM}}$ è indecidibile.

5.26 Sia $B = \{\langle M, w \rangle |$ M una TM a due nastri che scrive un simbolo diverso dal blank sul suo secondo nastro quando funziona su input $w\}$. Mostriamo che esiste una riduzione da A_{TM} a B. Supponiamo per assurdo che TM R decida B. Possiamo costruire una TM S che usa R per decidere A_{TM}.

S = "Su input $\langle M, w \rangle$:

1. Usa M per costruire la seguente TM T a due nastri.
   T = "Su input x:
   1. Simula M su x usando il primo nastro.
   2. Se la simulazione mostra che M accetta, scrive un simbolo diverso dal blank sul secondo nastro."
2. Esegue R su $\langle T, w \rangle$ per determinare se T su input w scrive un simbolo diverso dal blank sul secondo nastro.
3. Se R accetta, M accetta w, quindi *accetta*. Altrimenti, *rifiuta*."

6



ARGOMENTI AVANZATI NELLA TEORIA DELLA COMPUTAZIONE

In questo capitolo approfondiremo quattro aspetti particolarmente significativi della teoria della computazione: (1) il teorema di ricorsione, (2) le teorie logiche, (3) la Turing riducibilità e (4) la complessità descrittiva. L'argomento affrontato in ogni sezione è per lo più indipendente dagli altri, tranne per un'applicazione del teorema di ricorsione alla fine della sezione sulle teorie logiche. La Parte Tre di questo libro non dipende dal contenuto di questo capitolo.

6.1
IL TEOREMA DI RICORSIONE

Il teorema di ricorsione è un risultato matematico che gioca un ruolo importante nello studio avanzato nella teoria della computazione. Esso ha collegamenti con la logica matematica, con la teoria dei sistemi che si autoriproducono e persino con i virus informatici.

Per introdurre il teorema di ricorsione, consideriamo un paradosso che sorge nello studio della vita. Riguarda la possibilità di creare macchine in grado di costruire copie di sé stesse. Il paradosso può essere sintetizzato come segue.

1. Le forme di vita sono macchine.

2. Le forme di vita sono in grado di autoriprodursi.

3. Le macchine non sono in grado di autoriprodursi.

L'affermazione 1 è un principio cardine della biologia moderna. Crediamo che gli organismi funzionino in modo meccanicistico. L'affermazione 2 è ovvia. La capacità di autoriprodursi è una caratteristica essenziale di tutte le specie biologiche. Per l'affermazione 3, facciamo il ragionamento seguente per sostenere che le macchine non possono autoriprodursi. Consideriamo una macchina che costruisce altre macchine, come un'industria automatizzata che produce automobili. Le materie prime entrano a un'estremità del ciclo di produzione, i robot di produzione seguono un insieme di istruzioni, e poi i veicoli completati escono all'altra estremità del ciclo.

Affermiamo che l'industria deve essere più complessa delle automobili prodotte, nel senso che progettare l'industria dovrebbe essere più difficile che progettare un'automobile. Questa affermazione deve essere vera perché l'industria stessa ha al suo interno la progettazione dell'automobile, oltre al progetto di tutti i robot di produzione. Lo stesso ragionamento si applica a ogni macchina A che costruisce una macchina B: A deve essere *più* complessa di B. Ma una macchina non può essere più complessa di sé stessa. Di conseguenza, nessuna macchina può costruire sé stessa, e quindi l'autoriproduzione è impossibile.

Come risolvere questo paradosso? La risposta è semplice: l'affermazione 3 è errata. Creare macchine che riproducono sé stesse è possibile. Il teorema di ricorsione mostra come.

Autoreferenzialità

Iniziamo con il costruire una macchina di Turing che ignora il suo input e stampa una copia della sua stessa descrizione. Chiamiamo questa macchina *SELF*. Per semplificare la descrizione di *SELF*, abbiamo bisogno del lemma seguente.

LEMMA 6.1

Esiste una funzione calcolabile $q: \Sigma^* \longrightarrow \Sigma^*$, dove se w è una stringa, $q(w)$ è la descrizione di una macchina di Turing P_w che stampa w e poi si arresta.

DIMOSTRAZIONE. Una volta che abbiamo capito l'enunciato di questo lemma, la prova è facile. Ovviamente, possiamo prendere una stringa qualsiasi w e costruire da essa una macchina di Turing che ha w incorporata in una tabella cosicché la macchina può semplicemente dare in output w quando avviata. La seguente TM Q calcola $q(w)$.

Q = "Sulla stringa di input w:

1. Costruisce la seguente macchina di Turing P_w.
   P_w = "Su un input qualsiasi:
   1. Cancella l'input.
   2. Scrive w sul nastro.
   3. Si arresta."
2. Dà in output $\langle P_w \rangle$."

La macchina di Turing *SELF* è costituita da due parti: A e B. Pensiamo ad A e B come se fossero due procedure separate che insieme formano *SELF*. Vogliamo che *SELF* stampi $\langle SELF \rangle = \langle AB \rangle$.

La parte A va in esecuzione per prima e non appena ha completato passa il controllo a B. Il compito di A è stampare una descrizione di B e, viceversa il compito di B è stampare una descrizione di A. Il risultato è la descrizione voluta di *SELF*. I compiti sono simili, ma essi sono eseguiti diversamente. Mostriamo prima come ottenere la parte A.

Per A usiamo la macchina $P_{\langle B \rangle}$, descritta da $q(\langle B \rangle)$, che è il risultato di applicare la funzione q a $\langle B \rangle$. Quindi, la parte A è una macchina di Turing che stampa $\langle B \rangle$. La nostra descrizione di A dipende dall'avere una descrizione di B. Quindi non possiamo completare la descrizione di A finché non costruiamo B.

Ora la parte B. Saremmo tentati di definire B con $q(\langle A \rangle)$, ma questo non ha senso! Così facendo definiremmo B in termini di A, che a sua volta è definito in termini di B. Questa sarebbe una definizione *circolare* di un oggetto in termini di sé stesso, un errore logico. Definiamo invece B in modo che esso stampi A usando una diversa strategia: B *calcola* A dall'output che A produce.

Abbiamo definito $\langle A \rangle$ in modo che sia $q(\langle B \rangle)$. Ora viene la parte complicata: se B può ottenere $\langle B \rangle$, può applicare q a essa e ottenere $\langle A \rangle$. Ma B come ottiene $\langle B \rangle$? Esso è sul nastro quando A termina! Quindi B deve solo guardare il nastro per ottenere $\langle B \rangle$. Poi dopo che B ha calcolato $q(\langle B \rangle) = \langle A \rangle$, compone A e B in una singola macchina e scrive la sua descrizione $\langle AB \rangle = \langle SELF \rangle$ sul nastro. Riassumendo, abbiamo:

$A = P_{\langle B \rangle}$, e

B = "Sull'input $\langle M \rangle$, dove M è una parte di una TM:

1. Calcola $q(\langle M \rangle)$.
2. Compone il risultato con $\langle M \rangle$ per realizzare una TM completa.
3. Stampa la descrizione di questa TM e si arresta."

Questo completa la costruzione di *SELF*, per la quale una rappresentazione schematica è presentata nella figura seguente.

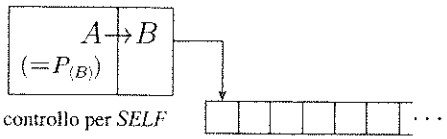


FIGURA 6.2
Rappresentazione schematica di *SELF*, una TM che stampa la sua descrizione

Se ora eseguiamo *SELF*, osserviamo il comportamento seguente.

1. In primo luogo A va in esecuzione. Essa stampa $\langle B \rangle$ sul nastro.
2. B inizia. Guarda il nastro e trova il suo input, $\langle B \rangle$.
3. B calcola $q(\langle B \rangle) = \langle A \rangle$ e lo compone con $\langle B \rangle$ in una descrizione di una TM, $\langle SELF \rangle$.
4. B stampa questa descrizione e si arresta.

Possiamo facilmente implementare questa costruzione in un qualsiasi linguaggio di programmazione per ottenere un programma che fornisca in output una copia di sé stesso. Possiamo perfino farlo con il linguaggio corrente. Supponiamo di voler fornire una frase che ordini al lettore di stampare una copia della frase stessa. Un modo di farlo è dire:

Stampa questa frase.

Questa frase ha il significato voluto perché dà istruzione al lettore di stampare una copia della frase stessa. Tuttavia non ha una traduzione evidente in un linguaggio di programmazione perché di solito la parola autoreferenziale "questa" nella frase non ha alcuna controparte. Ma non è necessaria alcuna autoreferenza per creare una tale frase. Considera l'alternativa seguente.

Stampa due copie di ciò che segue, la seconda tra virgolette:
"Stampa due copie di ciò che segue, la seconda tra virgolette:"

In questa frase, l'autoreferenza è sostituita con la stessa costruzione usata per costruire la TM *SELF*. La parte B della costruzione è la proposizione:

Stampa due copie di ciò che segue, la seconda tra virgolette:

La parte A è la stessa proposizione tra virgolette. A fornisce una copia di B a B in modo che B possa elaborare questa copia come fa la TM.

Il teorema di ricorsione fornisce la capacità di implementare l'autoreferenziale *questo* in un qualsiasi linguaggio di programmazione. Con esso, un qualsiasi programma ha la capacità di riferirsi alla sua propria descrizione, il che ha alcune applicazioni, come vedremo. Prima di arrivare a questo, enunciamo il teorema di ricorsione. Il teorema di ricorsione estende la tecnica che abbiamo usato nel costruire *SELF*, in modo che un programma possa acquisire la sua propria descrizione e poi continuare computando con essa, invece di stamparla semplicemente.

TEOREMA 6.3

Teorema di ricorsione Sia T una macchina di Turing che computa una funzione $t\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. Esiste una macchina di Turing R che calcola una funzione $r\colon \Sigma^* \longrightarrow \Sigma^*$, dove per ogni w,

$$r(w) = t(\langle R \rangle, w).$$

L'enunciato di questo teorema sembra un pó tecnico, ma in realtà esso rappresenta qualcosa di abbastanza semplice. Per costruire una macchina di Turing che può acquisire la sua propria descrizione e poi computare con essa, abbiamo bisogno solo di costruire una macchina, chiamata T nell'enunciato, che riceve la descrizione della macchina come input supplementare. Poi il teorema di ricorsione produce una nuova macchina R, che opera esattamente come fa T ma con la descrizione di R inserita automaticamente.

DIMOSTRAZIONE. La prova è simile alla costruzione di *SELF*. Costruiamo una TM R in tre parti, A, B e T, dove T è data dall'enunciato del teorema; una rappresentazione schematica è presentata nella figura seguente.

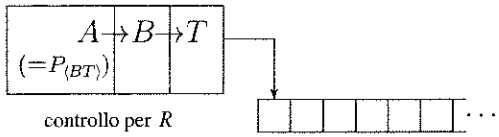


FIGURA 6.4
Rappresentazione schematica di R

In questo caso A è la macchina di Turing $P_{\langle BT \rangle}$ descritta da $q(\langle BT \rangle)$. Per conservare l'input w, riprogettiamo q in modo che $P_{\langle BT \rangle}$ scriva il suo output dopo ogni stringa preesistente sul nastro. Dopo la computazione di A, il nastro contiene $w\langle BT \rangle$.

Di nuovo B è una procedura che esamina il suo nastro e applica q al suo contenuto. Il risultato è $\langle A \rangle$. Poi B compone A, B e T in una singola macchina e ottiene la sua descrizione $\langle ABT \rangle = \langle R \rangle$. Infine, codifica questa descrizione insieme a w, scrive la stringa risultante $\langle R, w \rangle$ sul nastro e passa il controllo a T.

Terminologia per il teorema di ricorsione

Il teorema di ricorsione afferma che le macchine di Turing possono acquisire la loro propria descrizione e poi continuare computando con essa. A prima vista, questa capacità può sembrare utile solo per compiti insignificanti come costruire una macchina che stampa una copia di sé stessa. Ma, come

dimostreremo, il teorema di ricorsione è uno strumento utile per risolvere alcuni problemi riguardanti la teoria degli algoritmi.

Puoi usare il teorema di ricorsione nel modo seguente quando progetti algoritmi di macchine di Turing. Se stai progettando una macchina M, puoi includere la frase "ottiene" la propria descrizione $\langle M \rangle$" nella descrizione informale dell'algoritmo di M. Dopo aver ottenuto la sua descrizione, M può continuare usandola come userebbe ogni altro valore calcolato. Per esempio, M potrebbe semplicemente stampare $\langle M \rangle$, come accade nella TM *SELF*, o potrebbe contare il numero degli stati in $\langle M \rangle$, o forse perfino simulare $\langle M \rangle$. Per illustrare questo metodo, usiamo il teorema di ricorsione per descrivere la macchina *SELF*.

SELF = "Su un input qualsiasi:

1. Ottiene, attraverso il teorema di ricorsione, la propria descrizione $\langle SELF \rangle$.
2. Stampa $\langle SELF \rangle$."

Il teorema di ricorsione mostra come implementare il costrutto "ottiene la propria descrizione". Per realizzare la macchina *SELF*, scriviamo prima la seguente macchina T.

T = "Sull'input $\langle M, w \rangle$:

1. Stampa $\langle M \rangle$ e si arresta."

La TM T riceve una descrizione di una TM M e di una stringa w come input, e stampa la descrizione di M. Poi il teorema di ricorsione mostra come ottenere una TM R, che sull'input w agisce come T sull'input $\langle R, w \rangle$. Quindi, R stampa la descrizione di R – che è esattamente ciò che è richiesto alla macchina *SELF*.

Applicazioni

Un ***virus (informatico)*** è un programma che è progettato per diffondersi tra computer. Appropriatamente chiamato, esso ha molto in comune con un virus biologico. I virus informatici sono inattivi quando eseguiti indipendentemente come frammento di codice. Ma quando sono posti in modo appropriato in un computer in rete, in tal modo "infettandolo", possono attivarsi e trasmettere copie di sé stessi alle altre macchine accessibili. Vari supporti possono trasmettere i virus, tra cui Internet e i floppy disk. Per eseguire il suo compito fondamentale di auto-replicazione, un virus può contenere la costruzione descritta nella prova del teorema di ricorsione.

Consideriamo ora tre teoremi le cui dimostrazioni usano il teorema di ricorsione. Un'ulteriore applicazione appare nella prova del Teorema 6.17 nella Sezione 6.2.

In primo luogo torniamo alla prova dell'indecidibilità di A_{TM}. Ricordiamo che l'abbiamo provata in precedenza nel Teorema 4.11, usando il

metodo della diagonalizzazione di Cantor. Il teorema di ricorsione ci dà una nuova e più semplice prova.

TEOREMA 6.5

A_{TM} è indecidibile.

DIMOSTRAZIONE. Assumiamo che la macchina di Turing H decida A_{TM}, allo scopo di ottenere una contraddizione. Costruiamo la seguente macchina B.

B = "Sull'input w:

1. Ottiene, attraverso il teorema di ricorsione, la propria descrizione $\langle B \rangle$.
2. Esegue H sull'input $\langle B, w \rangle$.
3. Fa l'opposto di ciò che H fa. Cioè, *accetta* se H rifiuta e *rifiuta* se H accetta."

B sull'input w fa il contrario di quello che, secondo quanto dichiarato da H, dovrebbe fare. Quindi H non può decidere A_{TM}. Prova conclusa!

Il teorema seguente, che riguarda le macchine di Turing minimali, è un'altra applicazione del teorema di ricorsione.

DEFINIZIONE 6.6

Se M è una macchina di Turing, la *lunghezza* della descrizione $\langle M \rangle$ di M è il numero di simboli nella stringa che descrive M. Diciamo che M è ***minimale*** se non esiste nessuna macchina di Turing equivalente a M che ha una descrizione più corta. Sia

$$MIN_{\text{TM}} = \{\langle M \rangle \mid M \text{ è una TM minimale}\}.$$

TEOREMA 6.7

MIN_{TM} non è Turing-riconoscibile.

DIMOSTRAZIONE. Assumiamo che esista una TM E che enumeri MIN_{TM} e otteniamo una contraddizione. Costruiamo la seguente TM C.

C = "Sull'input w:

1. Ottiene, tramite il teorema di ricorsione, la sua descrizione $\langle C \rangle$.
2. Esegue l'enumeratore E fino a quando compare una macchina D con una descrizione più lunga di quella di C.
3. Simula D sull'input w."

Poiché MIN_{TM} è infinito, la lista di E deve contenere una TM con una descrizione più lunga della descrizione di C. Pertanto, il passo 2 di C alla fine termina con qualche TM D che è più lunga di C. Successivamente C simula D e quindi è equivalente a essa. Poiché C è più corta di D ed è equivalente a essa, D non può essere minimale. Ma D compare nella lista che E produce. Quindi, abbiamo una contraddizione.

La nostra applicazione finale del teorema di ricorsione è un tipo di teorema del punto fisso. Un ***punto fisso*** di una funzione è un valore che non è modificato dall'applicazione della funzione. In questo caso, consideriamo funzioni che sono trasformazioni calcolabili di descrizioni di macchine di Turing. Mostriamo che per una qualsiasi tale trasformazione, esiste una macchina di Turing il cui comportamento non è modificato dalla trasformazione. Questo teorema è chiamato la "versione punto fisso" del teorema di ricorsione.

TEOREMA 6.8

Sia $t\colon \Sigma^* \longrightarrow \Sigma^*$ una funzione calcolabile. Allora esiste una macchina di Turing F tale che $t(\langle F \rangle)$ descrive una macchina di Turing equivalente a F. Qui assumeremo che se una stringa non è una codifica corretta di una macchina di Turing, essa descrive una macchina di Turing che rifiuta sempre immediatamente.

In questo teorema, t gioca il ruolo della trasformazione ed F è il punto fisso.

DIMOSTRAZIONE. Sia F la seguente macchina di Turing.

F = "Sull'input w:

1. Ottiene, tramite il teorema di ricorsione, la sua descrizione $\langle F \rangle$.
2. Calcola $t(\langle F \rangle)$ per ottenere la descrizione di una TM G.
3. Simula G su w."

Evidentemente, $\langle F \rangle$ e $t(\langle F \rangle) = \langle G \rangle$ descrivono macchine di Turing equivalenti perché F simula G.

6.2
DECIDIBILITÀ DELLE TEORIE LOGICHE

La logica matematica è un settore della matematica che studia la matematica stessa. Essa affronta domande quali: Cos'è un teorema? Cos'è una dimostrazione? Cos'è la verità? Può un algoritmo decidere quali enunciati sono veri? Gli enunciati veri sono tutti dimostrabili? Accenneremo ad alcuni di questi argomenti nella nostra breve introduzione a questo ricco e attraente tema.

Concentreremo la nostra attenzione sul problema di stabilire se un'affermazione matematica sia vera o falsa e studieremo la decidabilità di questo problema. La risposta dipende dal settore della matematica in cui gli enunciati sono descritti. Esamineremo due settori: uno per il quale possiamo dare un algoritmo per decidere la verità, e un altro per il quale questo problema è indecidibile.

In primo luogo, abbiamo bisogno di stabilire un linguaggio preciso per formulare questi problemi. Il nostro scopo è poter considerare asserzioni matematiche quali

1. $\forall q\, \exists p\, \forall x,y\ \left[p{>}q \wedge (x,y{>}1 \rightarrow xy{\neq}p)\right]$,
2. $\forall a,b,c,n\ \left[(a,b,c{>}0 \wedge n{>}2) \rightarrow a^n{+}b^n{\neq}c^n\right]$, e
3. $\forall q\, \exists p\, \forall x,y\ \left[p{>}q \wedge (x,y{>}1 \rightarrow (xy{\neq}p \wedge xy{\neq}p{+}2))\right]$.

L'enunciato 1 afferma che esistono infiniti numeri primi, noto per essere vero dall'epoca di Euclide, circa 2300 anni fa. L'enunciato 2 è l'*ultimo teorema di Fermat*, noto per essere vero solo da quando Andrew Wiles lo ha dimostrato nel 1994. Infine, l'enunciato 3 afferma che esistono infinite coppie di numeri primi gemelli.[1] Nota come la *congettura dei numeri primi gemelli*, essa resta irrisolta.

Per esaminare se sia possibile automatizzare il processo di stabilire quali di questi enunciati siano veri, trattiamo gli enunciati semplicemente come stringhe e definiamo un linguaggio che consiste degli enunciati che sono veri. Quindi ci chiediamo se tale linguaggio sia decidibile.

Per rendere questo un pò più preciso, descriviamo la forma dell'alfabeto di questo linguaggio:

$$\{\wedge, \vee, \neg, (,), \forall, \exists, x, R_1, \ldots, R_k\}.$$

I simboli $\wedge$, $\vee$ e $\neg$ sono chiamati ***operazioni booleane***; "(" e ")" sono le ***parentesi***; i simboli $\forall$ e $\exists$ sono chiamati ***quantificatori***; il simbolo x è usato per denotare ***variabili***;[2] e i simboli $R_1, \ldots, R_k$ sono chiamati ***relazioni***.

[1] ***Coppie di primi gemelli*** sono primi che differiscono di 2.

[2] Se dobbiamo scrivere diverse variabili in una formula, usiamo i simboli w, y, z, o x_1, x_2, x_3, e così via. Non elenchiamo tutte le possibili infinite variabili nell'alfabeto per

Una ***formula*** è una stringa ben formata su questo alfabeto. Per completezza, a questo punto schematizzeremo la definizione tecnica ma ovvia di ***formula ben formata***, ma sentiti libero di saltare questa parte e passare al paragrafo seguente. Una stringa della forma $R_i(x_1, \ldots, x_k)$ è una ***formula atomica***. Il valore j è l'***arietà*** del simbolo di relazione R_i. Tutte le occorrenze dello stesso simbolo di relazione in una formula ben formata devono avere la stessa arietà. Una stringa ϕ, che verifica questo requisito, è una formula se

1. è una formula atomica,
2. ha la forma $\phi_1 \wedge \phi_2$ o $\phi_1 \vee \phi_2$ o $\neg\phi_1$, dove ϕ_1 e ϕ_2 sono formule più piccole, oppure
3. ha la forma $\exists x_i \, [\, \phi_1 \,]$ o $\forall x_i \, [\, \phi_1 \,]$, dove ϕ_1 è una formula più piccola.

Un quantificatore può essere ovunque in un'asserzione matematica. Il suo ***scope*** (o ambito) è il frammento dell'enunciato che compare all'interno della coppia di parentesi tonde o parentesi quadre che seguono la variabile quantificata. Assumiamo che tutte le formule sono in ***forma normale prenex***, dove tutti i quantificatori sono alla sinistra della formula. Una variabile che non è vincolata dallo scope di un quantificatore è chiamata una ***variabile libera***. Una formula senza variabili libere è chiamata una ***formula chiusa*** (sentence) o ***enunciato***.

ESEMPIO 6.9

Tra i seguenti esempi di formule, solo l'ultima è una formula chiusa.

1. $R_1(x_1) \wedge R_2(x_1, x_2, x_3)$
2. $\forall x_1 \, \left[R_1(x_1) \wedge R_2(x_1, x_2, x_3) \right]$
3. $\forall x_1 \, \exists x_2 \, \exists x_3 \, \left[R_1(x_1) \wedge R_2(x_1, x_2, x_3) \right]$

Avendo stabilito la sintassi delle formule, discutiamo il loro significato. Le operazioni booleane e i quantificatori hanno il loro significato abituale. Ma per stabilire il significato delle variabili e dei simboli di relazione, dobbiamo specificare due oggetti. Uno è l'***universo*** dei possibili valori delle variabili. L'altro è un assegnamento di specifiche relazioni ai simboli di relazione. Come abbiamo descritto nella Sezione 0.2 (pagina 9), una relazione è una funzione da k-tuple di elementi dell'universo a {VERO, FALSO}. L'arietà di un simbolo di relazione deve corrispondere a quello della relazione che gli è stata assegnata.

Un universo con un assegnamento di relazioni ai simboli di relazione è chiamato un ***modello***.[3] Formalmente, diciamo che un modello $\mathcal{M}$ è una tupla $(U, P_1, \ldots, P_k)$, dove U è l'universo e da P_1 a P_k sono le relazioni

mantenere l'alfabeto finito. Invece elenchiamo solo il simbolo di variabile x, e usiamo stringhe di x per indicare altre variabili, come xx per x_2, xxx per x_3, e così via.

[3]Un modello è anche alternativamente chiamato una ***interpretazione*** o una ***struttura***.

assegnate ai simboli da R_1 a R_k. A volte ci riferiamo al ***linguaggio di un modello*** come alla collezione di formule che usano solo i simboli di relazione a cui il modello assegna una relazione e che usano ciascun simbolo di relazione con l'arietà corretta. Se ϕ è una formula chiusa nel linguaggio di un modello, ϕ è vera o falsa in quel modello. Se ϕ è vera in un modello $\mathcal{M}$, diciamo che $\mathcal{M}$ è un modello per ϕ.

Se ti senti sopraffatto da queste definizioni, concentrati sul nostro obiettivo di illustrarle. Vogliamo definire formalmente un linguaggio di enunciati matematici in modo che possiamo chiederci se un algoritmo può determinare quali sono veri e quali sono falsi. I due esempi seguenti dovrebbero essere utili.

ESEMPIO 6.10

Sia ϕ la formula chiusa $\forall x \forall y \, \left[R_1(x, y) \vee R_1(y, x) \right]$. Sia $\mathcal{M}_1 = (\mathcal{N}, \leq)$ il modello il cui universo è l'insieme dei numeri naturali e che assegna la relazione "minore o uguale" al simbolo R_1. Ovviamente, ϕ è vera nel modello $\mathcal{M}_1$ perché $a \leq b$ o $b \leq a$ per due qualsiasi numeri naturali a e b. Però, se $\mathcal{M}_1$ assegna "minore" invece che "minore o uguale" a R_1, allora ϕ non sarebbe vera perché non lo è quando x e y sono uguali.

Se conosciamo prima quale relazione sarà assegnata a R_i, possiamo usare il simbolo abituale per quella relazione a posto di R_i, con notazione infissa al posto della notazione prefissa se la prima è abituale per quel simbolo. Quindi, con riferimento al modello $\mathcal{M}_1$, potremmo scrivere ϕ come $\forall x \forall y \, \left[x \leq y \vee y \leq x \right]$.

ESEMPIO 6.11

Ora sia $\mathcal{M}_2$ il modello il cui universo è l'insieme dei numeri reali $\mathcal{R}$ e che assegna la relazione $PLUS$ a R_1, dove $PLUS(a, b, c) =$ VERO quando $a + b = c$. Allora $\mathcal{M}_2$ è un modello per $\psi = \forall y \, \exists x \, \left[R_1(x, x, y) \right]$. Però, se al posto di $\mathcal{R}$ in $\mathcal{M}_2$, venisse usato $\mathcal{N}$ per l'universo la formula chiusa sarebbe falsa.

Come nell'Esempio 6.10, possiamo scrivere ψ come $\forall y \, \exists x \, \left[x + x = y \right]$ invece di $\forall y \, \exists x \, \left[R_1(x, x, y) \right]$ quando sappiamo prima che assegneremo la relazione di addizione a R_1.

Come illustrato dalll'Esempio 6.11, possiamo rappresentare funzioni come la funzione addizione mediante relazioni. Analogamente, possiamo rappresentare costanti come 0 e 1 mediante relazioni.

Diamo ora un'ultima definizione in previsione della prossima sezione. Se $\mathcal{M}$ è un modello, definiremo ***teoria di*** $\mathcal{M}$, denotata con $\mathrm{Th}(\mathcal{M})$, la collezione delle formule chiuse vere nel linguaggio di quel modello.

Una teoria decidibile

La teoria dei numeri è uno dei settori più antichi della matematica e anche uno dei più difficili. Molti enunciati apparentemente semplici relativi ai numeri naturali con le operazioni di somma e prodotto hanno turbato matematici per secoli, come la summenzionata congettura dei numeri primi gemelli.

In uno dei celebri risultati di avanzamento della logica matematica, Alonzo Church, basandosi sul lavoro di Kurt Gödel, mostrò che nessun algoritmo può decidere, in generale, se enunciati in teoria dei numeri siano veri o falsi. Formalmente, denotiamo con $(\mathcal{N}, +, \times)$ il modello il cui universo è l'insieme dei numeri naturali[4] con le usuali relazioni $+$ e $\times$. Church mostrò che $\mathrm{Th}(\mathcal{N}, +, \times)$, la teoria di questo modello, è indecidibile.

Prima di dare uno sguardo a questa teoria indecidibile, esaminiamone una che è decidibile. Sia $(\mathcal{N}, +)$ lo stesso modello, senza la relazione $\times$. La sua teoria è $\mathrm{Th}(\mathcal{N}, +)$. Per esempio, la formula $\forall x \, \exists y \, [x + x = y]$ è vera e quindi è un elemento di $\mathrm{Th}(\mathcal{N}, +)$, ma la formula $\exists y \forall x \, [x + x = y]$ è falsa e quindi non ne è un elemento.

TEOREMA 6.12

$\mathrm{Th}(\mathcal{N}, +)$ è decidibile.

IDEA. Questa prova è un'applicazione interessante e non banale della teoria degli automi finiti che abbiamo presentato nel Capitolo 1. Usiamo un fatto relativo agli automi finiti che appare nel Problema 1.37 (pagina 93) dove viene chiesto di mostrare che essi sono in grado di eseguire l'addizione se l'input è fornito in una forma speciale. L'input descrive tre numeri in parallelo rappresentando un bit di ciascun numero in un solo simbolo di un alfabeto a otto simboli. Qui usiamo una generalizzazione di questo metodo per presentare tuple di i numeri in parallelo usando un alfabeto con 2^i simboli.

Diamo un algoritmo in grado di determinare se il suo input, una formula chiusa ϕ nel linguaggio di $(\mathcal{N}, +)$, sia vera in quel modello. Sia

$$\phi = \mathsf{Q}_1 x_1 \, \mathsf{Q}_2 x_2 \cdots \mathsf{Q}_l x_l \, [\psi],$$

dove ciascun $\mathsf{Q}_1, \ldots, \mathsf{Q}_l$ rappresenta $\exists$ o $\forall$ e ψ è una formula senza quantificatori che ha variabili $x_1, \ldots, x_l$. Per ogni i da 0 a l, definiamo la formula ϕ_i come

$$\phi_i = \mathsf{Q}_{i+1} x_{i+1} \, \mathsf{Q}_{i+2} x_{i+2} \cdots \mathsf{Q}_l x_l \, [\psi].$$

Quindi $\phi_0 = \phi$ e $\phi_l = \psi$.

[4]Per convenienza in questo capitolo, cambiamo la nostra usuale definizione di $\mathcal{N}$ in modo che sia $\{0, 1, 2, \ldots\}$.

La formula ϕ_i ha i variabili libere. Per $a_1, \ldots, a_i \in \mathcal{N}$, denotiamo con $\phi_i(a_1, \ldots, a_i)$ la formula chiusa ottenuta sostituendo le costanti $a_1, \ldots, a_i$ alle variabili $x_1, \ldots, x_i$ in ϕ_i.

Per ogni i da 0 a l, l'algoritmo costruisce un automa finito A_i che riconosce l'insieme delle stringhe che rappresentano tuple di i numeri che rendono ϕ_i vera. L'algoritmo inizia costruendo A_l direttamente, usando una generalizzazione del metodo nella soluzione al Problema 1.37. Poi, per ogni i da l fino a 1, usa A_i per costruire A_{i-1}. Infine, quando l'algoritmo ha ottenuto A_0, verifica se A_0 accetta la stringa vuota. Se lo fa, ϕ è vera e l'algoritmo accetta.

DIMOSTRAZIONE. Per $i > 0$, definiamo l'alfabeto

$$\Sigma_i = \left\{ \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 1 \end{bmatrix}, \ldots, \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Quindi Σ_i contiene tutte le colonne di caratteri 0 e 1 di dimensione i. Una stringa su Σ_i rappresenta i interi in binario (corrispondenti alle righe). Definiamo anche $\Sigma_0 = \{[]\}$, dove $[]$ è un simbolo.

Ora presentiamo un algoritmo che decide $\mathrm{Th}(\mathcal{N}, +)$. Sull'input ϕ, dove ϕ è una formula chiusa, l'algoritmo opera come segue. Scriviamo ϕ e definiamo ϕ_i per ogni i da 0 a l, come nell'"idea della prova". Per ognuno di tali i, costruiamo un automa finito A_i da ϕ_i che accetta le stringhe su Σ_i corrispondenti alle tuple $a_1, \ldots, a_i$ di i elementi che rendono $\phi_i(a_1, \ldots, a_i)$ vera, come segue.

Per costruire la prima macchina A_l, osserviamo che $\phi_l = \psi$ è una combinazione Booleana di formule atomiche. Una formula atomica nel linguaggio di $\mathrm{Th}(\mathcal{N}, +)$ è un'unica addizione. È possibile costruire automi finiti per calcolare ognuna di queste relazioni individuali corrispondenti a un'unica addizione e poi comporli per fornire l'automa A_l. Fare questo comporta l'uso delle costruzioni della chiusura dei linguaggi regolari rispetto all'unione, intersezione e complemento per calcolare le combinazioni booleane delle formule atomiche.

Successivamente, mostriamo come costruire A_i da A_{i+1}. Se $\phi_i = \exists x_{i+1} \; \phi_{i+1}$, costruiamo A_i in modo che operi come opera A_{i+1}, tranne che sceglie non deterministicamente il valore di a_{i+1} invece che riceverlo come parte dell'input.

Più precisamente, A_i contiene uno stato per ogni stato di A_{i+1} e un nuovo stato iniziale. Ogni volta che A_i legge un simbolo

$$\begin{bmatrix} b_1 \\ \vdots \\ b_{i-1} \\ b_i \end{bmatrix},$$

dove ogni $b_j \in \{0,1\}$ è un bit del numero a_j, esso non deterministicamente prova a indovinare $z \in \{0,1\}$ e simula A_{i+1} sul simbolo di input

$$\begin{bmatrix} b_1 \\ \vdots \\ b_{i-1} \\ b_i \\ z \end{bmatrix}.$$

Inizialmente, A_i ipotizza non deterministicamente i bit iniziali di a_{i+1}, corrispondenti a degli 0 iniziali soppressi a partire da a_1 fino ad a_i diramandosi non deterministicamente, usando ε-transizioni, dal suo nuovo stato iniziale, a tutti gli stati che A_{i+1} potrebbe raggiungere dal suo stato iniziale con stringhe input di simboli

$$\left\{ \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \right\}$$

in Σ_{i+1}. Chiaramente, A_i accetta il suo input $(a_1, \ldots, a_i)$ se esiste qualche a_{i+1} tale che A_{i+1} accetta $(a_1, \ldots, a_{i+1})$.

Se $\phi_i = \forall x_{i+1}\, \phi_{i+1}$, essa è equivalente a $\neg\exists x_{i+1} \neg\, \phi_{i+1}$. Quindi, possiamo costruire l'automa finito che riconosce il complemento del linguaggio di A_{i+1}, poi applicare la costruzione precedente per il quantificatore $\exists$ e infine applicare nuovamente la complementazione per ottenere A_i.

L'automa finito A_0 accetta qualche input se e solo se ϕ_0 è vera. Quindi il passo finale dell'algoritmo controlla se A_0 accetta ε. Se lo fa, ϕ è vera e l'algoritmo accetta; altrimenti, rifiuta.

Una teoria indecidibile

Come menzionato in precedenza, $\mathrm{Th}(\mathcal{N}, +, \times)$ è una teoria indecidibile. Non esiste alcun algoritmo per decidere la veridicità o falsità di asserzioni matematiche, perfino quando sono limitate al linguaggio di $(\mathcal{N}, +, \times)$. Filosoficamente questo teorema ha una grande importanza perché dimostra che la matematica non può essere automatizzata. Enunciamo questo teorema, ma diamo solo un breve accenno della sua prova.

TEOREMA 6.13

$\mathrm{Th}(\mathcal{N}, +, \times)$ è indecidibile.

Sebbene essa contenga molti dettagli, la prova di questo teorema non è concettualmente difficile. Essa segue lo schema delle altre prove di indecidibilità presentate nel Capitolo 4. Mostriamo che $\mathrm{Th}(\mathcal{N}, +, \times)$ è indecidibile riducendo A_{TM} a esso, usando il metodo delle storie di computazione, descritto precedentemente (pagina 233). L'esistenza della riduzione dipende dal lemma seguente.

LEMMA 6.14

Sia M una macchina di Turing e w una stringa. Possiamo costruire da M e w una formula $\phi_{M,w}$ nel linguaggio di $(\mathcal{N}, +, \times)$ che contiene una sola variabile libera x, tale che la formula chiusa $\exists x\, \phi_{M,w}$ è vera se e solo se M accetta w.

IDEA. La formula $\phi_{M,w}$ "afferma" che x è una storia di computazione accettante (opportunamente codificata) di M su w. Ovviamente, x è, in effetti, solo un intero piuttosto grande, ma esso rappresenta una storia di computazione in una forma che può essere controllata usando le operazioni $+$ e $\times$.

La costruzione effettiva di $\phi_{M,w}$ è troppo complicata per essere presentata qui. Essa estrae i simboli individuali nella storia di computazione con le operazioni $+$ e $\times$ per controllare che la configurazione iniziale per M su w è corretta, che ciascuna configurazione segue legittimamente da quella che la precede, e che l'ultima configurazione sia accettante.

DIMOSTRAZIONE DEL TEOREMA 6.13 Diamo una riduzione mediante funzione da A_{TM} a $\mathrm{Th}(\mathcal{N}, +, \times)$. La riduzione costruisce la formula $\phi_{M,w}$ dall'input $\langle M, w\rangle$ usando il Lemma 6.14. Poi fornisce in output la formula chiusa $\exists x\, \phi_{M,w}$.

In seguito schematizziamo la prova del celebre *teorema di incompletezza* di Kurt Gödel. Informalmente, questo teorema afferma che in ogni sistema accettabile di formalizzazione della nozione di dimostrabilità in teoria dei numeri, alcuni enunciati veri non sono dimostrabili.

Grosso modo, la ***prova formale*** π di un enunciato ϕ è una sequenza di enunciati $S_1, S_2, \ldots, S_l$, dove $S_l = \phi$. Ciascun S_i segue dai precedenti enunciati e da alcuni assiomi di base sui numeri, usando semplici e precise regole di implicazione. Non abbiamo spazio per definire il concetto di prova; ma per i nostri scopi, sarà sufficiente assumere le seguenti due proprietà ragionevoli delle prove.

1. La correttezza di una prova di un enunciato può essere verificata da una macchina. Formalmente, $\{\langle \phi, \pi\rangle \mid \pi$ è una prova di $\phi\}$ è decidibile.
2. Il sistema delle prove è *robusto*. Ovvero, se un enunciato è dimostrabile (cioè ha una prova), esso è vero.

Se un sistema di deduzioni soddisfa queste due condizioni, valgono i tre teoremi seguenti.

TEOREMA 6.15

La classe degli enunciati dimostrabili in $\mathrm{Th}(\mathcal{N}, +, \times)$ è Turing-riconoscibile.

DIMOSTRAZIONE. L'algoritmo seguente P accetta il suo input ϕ se ϕ è dimostrabile. L'algoritmo P testa ogni stringa come candidata per una prova π di ϕ, usando il verificatore della prova (proof checker) la cui esistenza è garantita dalla proprietà delle prove 1. Se esso scopre che una di queste candidate è una prova, accetta.

Ora possiamo usare il teorema precedente per provare la nostra versione del teorema di incompletezza.

TEOREMA 6.16

Alcuni enunciati veri in $\mathrm{Th}(\mathcal{N}, +, \times)$ non sono dimostrabili.

DIMOSTRAZIONE. Diamo una prova per assurdo. Al contrario, assumiamo che tutti gli enunciati veri siano dimostrabili. Usando questa assunzione, descriviamo un algoritmo D che decide se un enunciato è vero, in contraddizione con il Teorema 6.13.

Sull'input ϕ, l'algoritmo D opera eseguendo l'algoritmo P fornito nella prova del Teorema 6.15 in parallelo sugli input ϕ e $\neg\phi$. Uno di questi due enunciati è vero e quindi, in base alla nostra assunzione, è dimostrabile. Pertanto, P si deve arrestare su uno dei due input. Per la proprietà delle prove 2, se ϕ è dimostrabile, allora ϕ è vero; e se $\neg\phi$ è dimostrabile, allora ϕ è falso. Quindi l'algoritmo D può decidere se ϕ è vero o falso.

Nel teorema finale di questa sezione, usiamo il teorema di ricorsione per fornire una formula chiusa esplicita nel linguaggio di $(\mathcal{N}, +, \times)$ che è vera ma non dimostrabile. Nel Teorema 6.16 abbiamo dimostrato l'esistenza di una tale formula chiusa ma in effetti non l'abbiamo descritta, come facciamo ora.

TEOREMA 6.17

La formula chiusa $\psi_{\text{unprovable}}$, come descritta nella prova, è non dimostrabile.

IDEA. Costruire una formula chiusa che afferma "Questa formula chiusa non è dimostrabile," usando il teorema di ricorsione per ottenere l'autoreferenza.

DIMOSTRAZIONE. Sia S una TM che opera come segue.

S = "Su un input qualsiasi:

1. Ottiene la sua descrizione $\langle S \rangle$ mediante il teorema di ricorsione.
2. Costruisce la formula chiusa $\psi = \neg\exists c \left[\phi_{S,0}\right]$, usando il Lemma 6.14.
3. Esegue l'algoritmo P della prova del Teorema 6.15 sull'input ψ.
4. Se il passo 3 accetta, *accetta*."

Sia $\psi_{\text{unprovable}}$ la formula chiusa ψ descritta nel passo 2 dell'algoritmo S. Questa formula chiusa è vera se e solo se S non accetta 0 (la stringa 0 è stata scelta arbitrariamente).

Se S trova una prova di $\psi_{\text{unprovable}}$, S accetta 0, e la formula chiusa sarebbe quindi falsa. Una formula chiusa falsa non può essere dimostrabile, pertanto questa situazione non può verificarsi. La sola possibilità che resta è che S non trovi una prova di $\psi_{\text{unprovable}}$ e quindi S non accetta 0. Ma allora $\psi_{\text{unprovable}}$ è vera, come affermato.

6.3 TURING RIDUCIBILITÀ

Abbiamo introdotto il concetto di riducibilità nel Capitolo 5 come modo di usare una soluzione a un problema per risolvere altri problemi. Quindi, se A è riducibile a B, e troviamo una soluzione a B, possiamo ottenere una soluzione ad A. In seguito, abbiamo descritto la *riducibilità mediante funzione*, una forma specifica di riducibilità. Ma la riducibilità mediante funzione cattura il nostro concetto intuitivo di riducibilità nel modo più generale? Non lo fa.

Per esempio, consideriamo i due linguaggi A_{TM} e $\overline{A_{\mathsf{TM}}}$. Intuitivamente, essi sono riducibili l'uno all'altro perché una soluzione all'uno potrebbe essere usata per risolvere l'altro semplicemente invertendo la risposta. Tuttavia, sappiamo che $\overline{A_{\mathsf{TM}}}$ *non* è riducibile mediante funzione ad A_{TM} perché A_{TM} è Turing riconoscibile ma $\overline{A_{\mathsf{TM}}}$ non lo è. A questo punto presentiamo una forma molto generale di riducibilità, chiamata ***Turing riducibilità***, che cattura più minuziosamente il nostro concetto intuitivo di riducibilità.

DEFINIZIONE **6.18**

Un ***oracolo*** per un linguaggio B è un dispositivo esterno che è in grado di riferire se una stringa w è un elemento di B. Una ***macchina di Turing con oracolo*** è una macchina di Turing modificata che ha l'ulteriore possibilità di interrogare un oracolo. Usiamo la notazione M^B per descrivere una macchina di Turing che ha un oracolo per il linguaggio B.

Non ci occupiamo del modo con cui l'oracolo determina le sue risposte. Usiamo il termine oracolo per sottintendere un'abilità magica e consideriamo oracoli per linguaggi che non sono decidabili mediante gli usuali algoritmi, come mostra l'esempio seguente.

ESEMPIO 6.19

Consideriamo un oracolo per A_{TM}. Una macchina di Turing con oracolo che ha un oracolo per A_{TM} può decidere più linguaggi di quanti possa una comune macchina di Turing. Una tale macchina può (ovviamente) decidere A_{TM} stesso, interrogando l'oracolo sull'input. Può anche decidere E_{TM}, il problema di verificare se il linguaggio riconosciuto è vuoto per le TM con la seguente procedura chiamata $T^{A_{\mathsf{TM}}}$.

$T^{A_{\mathsf{TM}}}$ = "Sull'input $\langle M \rangle$, dove M è una TM:

1. Costruisce la seguente TM N.
   N = "Su un qualsiasi input:
   1. Esegue M in parallelo su tutte le stringhe in Σ^*.
   2. Se M accetta qualcuna di queste stringhe, *accetta*."
2. Interroga l'oracolo per stabilire se $\langle N, 0 \rangle \in A_{\mathsf{TM}}$.
3. Se l'oracolo risponde NO, *accetta*; se (risponde) SI, *rifiuta*."

Se il linguaggio di M non è vuoto, N accetterà ogni input e, in particolare, l'input 0. Quindi l'oracolo risponderà SI, e $T^{A_{\mathsf{TM}}}$ rifiuterà. Al contrario, se il linguaggio di M è vuoto, $T^{A_{\mathsf{TM}}}$ accetterà. Pertanto $T^{A_{\mathsf{TM}}}$ decide E_{TM}. Diciamo che E_{TM} è ***decidibile rispetto ad*** A_{TM}. Questo ci porta alla definizione di Turing riducibilità.

DEFINIZIONE **6.20**

Un linguaggio A è ***Turing-riducibile*** a un linguaggio B, denotato con $A \leq_{\mathrm{T}} B$, se A è decidibile rispetto a B.

L'Esempio 6.19 mostra che E_{TM} è Turing riducibile ad A_{TM}. La Turing riducibilità risponde al nostro concetto intuitivo di riducibilità come mostra il teorema seguente.

TEOREMA 6.21

Se $A \leq_{\mathrm{T}} B$ e B è decidibile, allora A è decidibile.

DIMOSTRAZIONE. Se B è decidibile, allora possiamo sostituire l'oracolo per B con una procedura effettiva che decide B. Quindi, possiamo sostituire la macchina di Turing con oracolo che decide A con una comune macchina di Turing che decide A.

La Turing riducibilità è una generalizzazione della riducibilità mediante funzione. Se $A \leq_{\mathrm{m}} B$, allora $A \leq_{\mathrm{T}} B$ perché la riduzione mediante funzione può essere usata per fornire una macchina di Turing con oracolo che decide A rispetto a B.

Una macchina di Turing con un oracolo per A_{TM} è molto potente. Può risolvere molti problemi che non sono solubili mediante macchine di Turing comuni. Ma perfino una macchina così non può decidere tutti i linguaggi (vedi Esercizio 6.4).

6.4 UNA DEFINIZIONE DI INFORMAZIONE

Le nozioni di *algoritmo* e *informazione* sono fondamentali in Informatica. Mentre la tesi di Church-Turing fornisce una definizione di algoritmo universalmente applicabile, non è nota una definizione di informazione ugualmente completa. Invece di una definizione sola e universale di informazione, sono usate diverse definizioni – a seconda dell'applicazione. In questa sezione presentiamo un modo di definire l'informazione, usando la teoria della computazione.

Iniziamo con un esempio. Consideriamo il contenuto di informazione delle due sequenze binarie seguenti.

$$A = 0101010101010101010101010101010101010101$$

$$B = 1110010110100011101010000111010011010111$$

Intuitivamente, la sequenza A contiene poca informazione perché essa è semplicemente una ripetizione del pattern 01 venti volte. Al contrario, la sequenza B sembra contenere più informazione.

Possiamo usare questo semplice esempio per illustrare l'idea dietro la definizione di informazione che presentiamo. Definiamo la quantità di informazione contenuta in un oggetto come la dimensione della più piccola rappresentazione o descrizione di quell'oggetto. Per descrizione di un oggetto, intendiamo una caratterizzazione precisa e non ambigua dell'oggetto tale che possiamo riprodurlo dalla sola descrizione. Quindi, la sequenza A contiene poca informazione perché ha una piccola descrizione, invece la sequenza B apparentemente contiene più informazione perché sembra non avere alcuna descrizione succinta.

Perché consideriamo solo la descrizione *più corta* quando determiniamo la quantità di informazione di un oggetto? Possiamo sempre descrivere un oggetto, come una stringa, mettendo una copia dell'oggetto direttamente nella descrizione. Quindi, ovviamente possiamo descrivere la stringa precedente B con una tabella lunga 40 bit, contenente una copia di B. Questo tipo di descrizione non è mai più corta dell'oggetto stesso e non ci dice nulla riguardo alla quantità di informazione. Tuttavia, una descrizione che sia sensibilmente più corta dell'oggetto implica che l'informazione in esso contenuta può essere compressa in un piccolo volume e quindi, la quantità di informazione non può essere molto grande. Perciò la dimensione della descrizione più corta determina la quantità di informazione.

Ora formalizziamo questa idea intuitiva. Farlo non è difficile, ma abbiamo bisogno di fare del lavoro preliminare. In primo luogo, limitiamo la nostra attenzione a oggetti che sono stringhe binarie. Altri oggetti possono essere rappresentati come stringhe binarie, quindi questa restrizione non limita la portata della teoria. In secondo luogo, consideriamo solo descrizioni che sono esse stesse stringhe binarie. Imponendo questo requisito, possiamo facilmente confrontare la lunghezza di un oggetto con la lunghezza della sua descrizione. Nella sezione seguente, consideriamo il tipo di descrizione che permettiamo.

Descrizioni di lunghezza minimale

Molti tipi di linguaggi per le descrizioni possono essere usati per definire l'informazione. Scegliere quale linguaggio usare ha un effetto sulle caratteristiche della definizione. Il nostro linguaggio per le descrizioni è basato sugli algoritmi.

Un modo per usare gli algoritmi per descrivere stringhe è costruire una macchina di Turing che stampi la stringa quando inizia con un nastro contenente solo caratteri blank e poi rappresentare questa stessa macchina di Turing come una stringa. Quindi, la stringa che rappresenta la macchina di Turing è una descrizione della stringa originaria. Un inconveniente di questo approccio è che una macchina di Turing non può rappresentare una tabella di informazioni succintamente con la sua

funzione di transizione. Per rappresentare una stringa di n bit, dovresti usare n stati ed n righe nella tabella della funzione di transizione. Ciò darebbe luogo ad una descrizione che è troppo lunga per il nostro scopo. Invece, usiamo il seguente linguaggio più succinto per le descrizioni.

Descriviamo una stringa binaria x con una macchina di Turing M e un input binario w a M. La lunghezza della descrizione è la lunghezza complessiva della rappresentazione di M e w. Denotiamo questa descrizione con la nostra abituale notazione per codificare diversi oggetti in una singola stringa binaria $\langle M, w\rangle$. Ma qui dobbiamo prestare ulteriore attenzione all'operazione di codifica $\langle \cdot, \cdot \rangle$ perché dobbiamo produrre un risultato conciso. Definiamo la stringa $\langle M, w\rangle$ come $\langle M\rangle w$, in cui semplicemente concateniamo la stringa binaria w alla fine della codifica binaria di M. La codifica $\langle M\rangle$ di M può essere fatta in un qualsiasi modo standard, tranne che per la sottigliezza che descriviamo nel paragrafo seguente. (Non preoccuparti per questo dettaglio alla tua prima lettura di questo argomento. Per adesso, salta dopo il prossimo paragrafo e la figura seguente.)

Quando concateniamo w alla fine di $\langle M\rangle$ per ottenere una descrizione di x, potresti rischiare un problema se il punto in cui $\langle M\rangle$ finisce e w inizia non è distinguibile dalla descrizione stessa. Altrimenti, possono esserci diversi modi di dividere la descrizione $\langle M\rangle w$ in una TM sintatticamente corretta e un input, e allora la descrizione sarebbe ambigua e quindi non valida. Evitiamo questo problema assicurandoci che possiamo individuare la separazione tra $\langle M\rangle$ e w in $\langle M\rangle w$. Un modo per farlo è scrivere ciascun bit di $\langle M\rangle$ due volte, scrivendo `0` come `00` e `1` come `11`, e poi facendo seguire `01` per evidenziare il punto di separazione. Illustriamo questa idea nella figura seguente, che raffigura la descrizione $\langle M, w\rangle$ di una qualche stringa x.

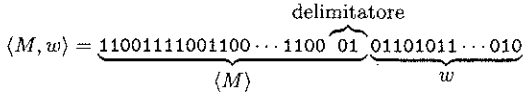


FIGURA 6.22
Esempio della struttura della descrizione $\langle M, w\rangle$ di una stringa x

Ora che abbiamo fissato il nostro linguaggio per le descrizioni, siamo pronti per definire la nostra misura della quantità di informazione in una stringa.

DEFINIZIONE **6.23**

Sia x una stringa binaria. La ***descrizione minimale*** di x, denotata con $d(x)$, è la stringa più corta $\langle M, w\rangle$ tale che la TM M sull'input w si arresta con x sul suo nastro. Se esistono diverse siffatte stringhe, scegliamo la prima lessicograficamente tra esse. La ***complessità descrittiva***[5] di x, denotata con $\mathrm{K}(x)$, è

$$\mathrm{K}(x) = |d(x)|.$$

In altre parole $\mathrm{K}(x)$ è la lunghezza della descrizione minimale di x. La definizione di $\mathrm{K}(x)$ intende catturare la nostra intuizione della quantità di informazione nella stringa x. Di seguito dimostriamo alcuni semplici risultati riguardanti la complessità descrittiva.

TEOREMA 6.24

$\exists c \forall x \left[\mathrm{K}(x) \leq |x| + c\right]$

Questo teorema afferma che la complessità descrittiva di una stringa è pari alla sua lunghezza più una costante fissata. La costante è una costante universale, non dipendente dalla stringa.

DIMOSTRAZIONE. Per provare un limite superiore per $\mathrm{K}(x)$ come afferma il teorema, dobbiamo solo mostrare una qualche descrizione di x che non è più lunga del limite dichiarato. Allora la descrizione minimale di x può essere più corta della descrizione mostrata, ma non più lunga.

Consideriamo la seguente descrizione della stringa x. Sia M una macchina di Turing che si arresta non appena inizia la computazione. Questa macchina calcola la funzione identità – il suo output è uguale al suo input. Una descrizione di x è semplicemente $\langle M\rangle x$. Porre c uguale alla lunghezza di $\langle M\rangle$ completa la prova.

Il Teorema 6.24 illustra come usiamo l'input alla macchina di Turing per rappresentare informazione che richiederebbe una descrizione sensibilmente più grande se memorizzata usando invece la funzione di transizione della macchina. Essa è conforme alla nostra intuizione che la quantità di informazione contenuta in una stringa non può essere (in sostanza) più della sua

[5] La ***complessità descrittiva*** è chiamata ***complessità di Kolmogorov*** o ***complessità di Kolmogorov–Chaitin*** in alcuni testi.

lunghezza. Analogamente, l'intuizione suggerisce che l'informazione contenuta nella stringa xx non è sensibilmente più dell'informazione contenuta in x. Il teorema seguente conferma questo fatto.

TEOREMA 6.25

$\exists c \forall x \left[\mathrm{K}(xx) \leq \mathrm{K}(x) + c\right]$

DIMOSTRAZIONE. Consideriamo la seguente macchina di Turing M, che si aspetta un input della forma $\langle N, w\rangle$, dove N è una macchina di Turing e w è un input per essa.

M = "Sull'input $\langle N, w\rangle$, dove N è una TM e w è una stringa:

1. Esegue N su w finché si arresta e produce una stringa di output s.
2. Fornisce in output la stringa ss."

Una descrizione di xx è $\langle M\rangle d(x)$. Ricorda che $d(x)$ è una descrizione minimale di x. La lunghezza di questa descrizione è $|\langle M\rangle| + |d(x)|$, che è $c + \mathrm{K}(x)$ dove c è la lunghezza di $\langle M\rangle$.

Di seguito esaminiamo come la complessità descrittiva della concatenazione xy di due stringhe x e y è connessa alle loro complessità individuali. Il Teorema 6.24 potrebbe indurre a credere che la complessità della concatenazione è al più la somma delle complessità individuali (più una costante fissata), ma il costo di combinare due descrizioni conduce a un limite più grande, come descritto nel teorema seguente.

TEOREMA 6.26

$\exists c \forall x,y \left[\mathrm{K}(xy) \leq 2\mathrm{K}(x) + \mathrm{K}(y) + c\right]$

DIMOSTRAZIONE. Costruiamo una TM M che divida il suo input w in due descrizioni separate. I bit della prima descrizione $d(x)$ sono tutti raddoppiati e terminano con la stringa 01 prima che appaia la seconda descrizione $d(y)$, come descritto nel testo che precede la Figura 6.22. Non appena sono state ottenute entrambe le descrizioni, esse vengono eseguite per ottenere le stringhe x e y e l'output xy viene prodotto.

La lunghezza di questa descrizione di xy è chiaramente due volte la complessità di x più la complessità di y più una costante fissata per descrivere M. Questa somma è

$$2\mathrm{K}(x) + \mathrm{K}(y) + c,$$

e la prova è completa.

Possiamo leggermente migliorare questo teorema usando un metodo più efficiente di indicare la separazione tra le due descrizioni. Un modo evita di raddoppiare i bit di $d(x)$. Invece antepone la lunghezza di $d(x)$ rappresentata come un intero binario che è stato raddoppiato per distinguerlo da $d(x)$. Questa descrizione contiene ancora abbastanza informazione per poterla decodificare nelle due descrizioni di x e y, ed essa ora ha lunghezza al più

$$2\log_2(\mathrm{K}(x)) + \mathrm{K}(x) + \mathrm{K}(y) + c.$$

Altri piccoli miglioramenti sono possibili. Tuttavia, come il Problema 6.19 chiede di mostrare, non possiamo raggiungere il limite $\mathrm{K}(x) + \mathrm{K}(y) + c$.

Ottimalità della definizione

Ora che abbiamo dimostrato alcune delle proprietà elementari della complessità descrittiva e che hai avuto la possibilità di sviluppare una qualche intuizione, discutiamo alcune caratteristiche delle definizioni.

La nostra definizione di $\mathrm{K}(x)$ ha una proprietà di ottimalità tra tutti i possibili modi di definire la complessità descrittiva con gli algoritmi. Supponiamo di definire un ***linguaggio per le descrizioni*** generale come una funzione calcolabile $p\colon \Sigma^* \longrightarrow \Sigma^*$ e definire la descrizione minimale di x rispetto a p, denotata con $d_p(x)$, come la prima stringa s tale che $p(s) = x$, nell'ordine standard delle stringhe. Quindi, s è la prima lessicograficamente tra le più corte descrizioni di x. Definiamo $\mathrm{K}_p(x) = |d_p(x)|$.

Per esempio, consideriamo un linguaggio di programmazione come Python (codificato in binario) come il linguaggio per le descrizioni. Allora $d_{\mathrm{Python}}(x)$ sarebbe il programma minimale in Python che fornisce in output x, e $\mathrm{K}_{\mathrm{Python}}(x)$ sarebbe la lunghezza del programma minimale.

Il teorema seguente mostra che ogni linguaggio per le descrizioni di questo tipo non è significativamente più conciso del linguaggio di macchine di Turing e input che abbiamo definito inizialmente.

TEOREMA 6.27

Per ogni linguaggio per le descrizioni p, esiste una costante fissata c che dipende solo da p, tale

$$\forall x \left[\mathrm{K}(x) \leq \mathrm{K}_p(x) + c\right].$$

IDEA. Illustriamo l'idea di questa prova usando l'esempio di Python. Supponiamo che x abbia una descrizione corta w in Python. Sia M una TM che può interpretare Python e usiamo il programma Python per x come l'input w di M. Allora $\langle M, w\rangle$ è una descrizione di x che è più grande della descrizione Python di x solo per una quantità fissata. La lunghezza in più è per l'interprete M per Python.

DIMOSTRAZIONE. Si prenda un linguaggio qualsiasi per le descrizioni p e si consideri la seguente macchina di Turing M.

M = "Sull'input w:

1. Fornisce in output $p(w)$."

Allora $\langle M\rangle d_p(x)$ è una descrizione di x la cui lunghezza è più grande di $\mathrm{K}_p(x)$ per al più una costante fissata. La costante è la lunghezza di $\langle M\rangle$.

Stringhe incompressibili e casualità

Il Teorema 6.24 mostra che la descrizione minimale di una stringa non è mai molto più lunga che la stringa stessa. Naturalmente per alcune stringhe, la descrizione minimale può essere molto più corta se l'informazione nella stringa appare in modo sparso o ridondante. Esistono stringhe che non hanno descrizioni corte? In altre parole, la descrizione minimale di alcune stringhe è effettivamente lunga quanto la stringa stessa? Mostriamo che tali stringhe esistono. Queste stringhe non possono essere descritte più concisamente che semplicemente trascrivendole.

DEFINIZIONE **6.28**

Sia x una stringa. Diciamo che x è ***c-compressibile*** se

$$\mathrm{K}(x) \leq |x| - c.$$

Se x non è c-compressibile, diciamo che x è ***incompressibile con c***. Se x è incompressibile con 1, diciamo che x è ***incompressibile***.

In altre parole, se x ha una descrizione che è c bit più corta della sua lunghezza, x è c-compressibile. Altrimenti, x è incompressibile con c. Infine, se x non ha alcuna descrizione più corta che sé stessa, x è incompressibile. In primo luogo mostriamo che esistono stringhe incompressibili, e poi discutiamo le loro proprietà interessanti. In particolare, mostriamo che le stringhe incompressibili assomigliano alle stringhe che sono ottenute da lanci casuali di monete.

TEOREMA 6.29

Esistono stringhe incompressibili di qualsiasi lunghezza.

IDEA. Il numero di stringhe di lunghezza n è più grande del numero delle descrizioni di lunghezza minore di n. Ogni descrizione descrive al più una

stringa. Quindi, qualche stringa di lunghezza n non è descritta da alcuna descrizione di lunghezza minore di n. Questa stringa è incompressibile.

DIMOSTRAZIONE. Il numero delle stringhe binarie di lunghezza n è 2^n. Ogni descrizione è una stringa binaria, quindi il numero di descrizioni di lunghezza minore di n è al più la somma del numero di stringhe di ogni lunghezza fino a $n-1$, ovvero

$$\sum_{0 \le i \le n-1} 2^i = 1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1.$$

Il numero delle descrizioni corte è minore del numero di stringhe di lunghezza n. Quindi, almeno una stringa di lunghezza n è incompressibile.

COROLLARIO 6.30

Almeno $2^n - 2^{n-c+1} + 1$ stringhe di lunghezza n sono incompressibili mediante c.

DIMOSTRAZIONE. Estendiamo la prova del Teorema 6.29. Ogni stringa c-compressibile ha una descrizione di lunghezza al più $n-c$. Non possono esserci più di $2^{n-c+1} - 1$ di tali descrizioni. Quindi, al più $2^{n-c+1} - 1$ delle 2^n stringhe di lunghezza n possono avere tali descrizioni. Le restanti stringhe, che ammontano almeno a $2^n - (2^{n-c+1} - 1)$, sono incompressibili mediante c.

Le stringhe incompressibili hanno molte proprietà che ci aspetteremmo di trovare in stringhe scelte in modo casuale. Per esempio, possiamo mostrare che una stringa incompressibile di lunghezza n ha all'incirca un ugual numero di `0` e `1`, e che la lunghezza della sua più lunga sequenza (run) di `0` consecutivi è approssimativamente $\log_2 n$, come ci aspetteremmo di trovare in una stringa casuale di quella lunghezza. Dimostrare questi enunciati ci porterebbe troppo lontano nella combinatorica e probabilità, ma dimostreremo un teorema che forma la base per questi enunciati.

Questo teorema mostra che ogni proprietà computabile che vale per "quasi tutte" le stringhe vale anche per tutte le stringhe incompressibili sufficientemente lunghe. Come menzionato nella Sezione 0.2, una ***proprietà*** delle stringhe è semplicemente una funzione f che associa alle stringhe valori in $\{\text{VERO}, \text{FALSO}\}$. Diciamo che una proprietà ***vale per quasi tutte le stringhe*** se la frazione delle stringhe di lunghezza n su cui è FALSO tende a 0 al crescere di n. È presumibile che una stringa lunga scelta casualmente soddisfi una proprietà calcolabile che vale per quasi tutte le stringhe. Quindi, stringhe casuali e stringhe incompressibili condividono tali proprietà.

TEOREMA 6.31

Sia f una proprietà calcolabile che vale per quasi tutte le stringhe. Allora, per ogni $b > 0$, la proprietà f è FALSO solo su un numero finito di stringhe che sono incompressibili mediante b.

DIMOSTRAZIONE. Sia M il seguente algoritmo.

M = "Sull'input i, un intero binario:

1. Trova l'i-esima stringa s tale che $f(s) = \text{FALSO}$, nell'ordine standard per le stringhe.
2. Fornisce in output la stringa s."

Possiamo usare M per ottenere descrizioni corte di stringhe che non hanno la proprietà f come segue. Per ogni tale stringa x, sia i_x la posizione o ***indice*** di x nella lista di tutte le stringhe che non hanno la proprietà f, nell'ordine standard per le stringhe (cioè, per lunghezza e lessicograficamente all'interno di ciascuna lunghezza). Allora $\langle M, i_x \rangle$ è una descrizione di x. La lunghezza di questa descrizione è $|i_x| + c$, dove c è la lunghezza di $\langle M \rangle$. Poiché poche stringhe non hanno la proprietà f, l'indice di x è piccolo e la sua descrizione è proporzionalmente corta.

Fissa un qualsiasi numero $b > 0$. Scegli n tale che al più una frazione $1/2^{b+c+1}$ delle stringhe di lunghezza minore o uguale a n non abbia la proprietà f. Ogni n sufficientemente grande soddisfa questa condizione perché f vale per quasi tutte le stringhe. Sia x una stringa di lunghezza n che non abbia la proprietà f. Abbiamo $2^{n+1} - 1$ stringhe di lunghezza minore o uguale a n, quindi

$$i_x \le \frac{2^{n+1} - 1}{2^{b+c+1}} \le 2^{n-b-c}.$$

Allora, $|i_x| \le n-b-c$, perciò la lunghezza di $\langle M, i_x \rangle$ è al più $(n-b-c)+c = n-b$, il che implica che

$$\mathrm{K}(x) \le n - b.$$

Quindi ogni x sufficientemente lunga che non abbia la proprietà f è compressibile mediante b. Allora solo un numero finito di stringhe che non hanno la proprietà f sono incompressibili mediante b, e il teorema è provato.

A questo punto, sarebbe appropriato esibire qualche esempio di stringa incompressibile. Tuttavia, come chiede di mostrare il Problema 6.16, la misura di complessità K non è calcolabile. Inoltre, nessun algoritmo può decidere in generale se una stringa è incompressibile, in accordo al Problema 6.17. Infatti, in accordo al Problema 6.18, nessun sottoinsieme infinito di esse è Turing riconoscibile. Quindi non abbiamo alcun mezzo per ottenere

lunghe stringhe incompressibili e non avremmo alcun modo per determinare se una stringa è incompressibile anche se ne avessimo una. Il teorema seguente descrive alcune stringhe che sono quasi incompressibili, sebbene esso non fornisca un modo per esibirle esplicitamente.

TEOREMA 6.32

Per qualche costante b, per ogni stringa x, la descrizione minimale $d(x)$ di x è incompressibile mediante b.

DIMOSTRAZIONE. Consideriamo la seguente TM M:

M = "Sull'input $\langle R, y\rangle$, dove R è una TM e y è una stringa:

1. Esegue R su y e *rifiuta* se il suo output non è della forma $\langle S, z\rangle$.
2. Esegue S su z e si arresta con il suo output sul nastro."

Sia b uguale a $|\langle M\rangle|+1$. Mostriamo che b soddisfa il teorema. Supponiamo al contrario che $d(x)$ sia b-compressibile per qualche stringa x. Quindi

$$|d(d(x))| \leq |d(x)| - b.$$

Ma allora $\langle M\rangle d(d(x))$ è una descrizione di x la cui lunghezza è al più

$$|\langle M\rangle| + |d(d(x))| \leq (b-1) + (|d(x)| - b) = |d(x)| - 1.$$

Questa descrizione di x è più corta di $d(x)$, contraddicendo la minimalità di quest'ultima.

ESERCIZI

6.1 Fornire un esempio nello spirito del teorema di ricorsione di un programma in un linguaggio di programmazione reale (o una ragionevole approssimazione di esso) che stampa sé stesso.

6.2 Mostrare che ogni sottoinsieme infinito di MIN_{TM} non è Turing-riconoscibile.

A6.3 Mostrare che se $A \leq_{\mathrm{T}} B$ e $B \leq_{\mathrm{T}} C$, allora $A \leq_{\mathrm{T}} C$.

6.4 Sia $A_{\mathsf{TM}}{}' = \{\langle M, w\rangle |\ M$ è una TM con oracolo e $M^{A_{\mathsf{TM}}}$ accetta $w\}$. Mostrare che $A_{\mathsf{TM}}{}'$ è indecidibile rispetto ad A_{TM}.

A6.5 L'enunciato $\exists x \forall y\ [x{+}y{=}y]$ è un elemento di $\mathrm{Th}(\mathcal{N}, +)$? Perché o perché no? E per quanto riguarda l'enunciato $\exists x \forall y\ [x{+}y{=}x]$?

PROBLEMI

6.6 Per ogni $m > 1$ sia $\mathcal{Z}_m = \{0, 1, 2, \ldots, m-1\}$ e sia $\mathcal{F}_m = (\mathcal{Z}_m, +, \times)$ il modello il cui universo è $\mathcal{Z}_m$ e che ha relazioni corrispondenti alle relazioni $+$ e $\times$ calcolate modulo m. Mostrare che per ogni m, la teoria $\mathrm{Th}(\mathcal{F}_m)$ è decidibile.

6.7 Mostrare che dati due qualsiasi linguaggi A e B, esiste un linguaggio J tale che $A \leq_{\mathrm{T}} J$ e $B \leq_{\mathrm{T}} J$.

6.8 Mostrare che per ogni linguaggio A, esiste un linguaggio B tale che $A \leq_{\mathrm{T}} B$ e $B \not\leq_{\mathrm{T}} A$.

***6.9** Provare che esistono due linguaggi A e B che non sono Turing confrontabili – cioè tali che $A \not\leq_{\mathrm{T}} B$ e $B \not\leq_{\mathrm{T}} A$.

***6.10** Siano A e B due linguaggi disgiunti. Diciamo che il linguaggio C *separa* A e B se $A \subseteq C$ e $B \subseteq \overline{C}$. Descrivere due linguaggi disgiunti Turing riconoscibili che non sono separabili mediante alcun linguaggio decidibile.

6.11 Mostrare che $\overline{EQ_{\mathsf{TM}}}$ è riconosciuto da una macchina di Turing con un oracolo per A_{TM}.

6.12 Nel Corollario 4.18, mostrammo che l'insieme di tutti i linguaggi non è numerabile. Usare questo risultato per provare che esistono linguaggi che non sono riconosciuti da una macchina di Turing con oracolo, con un oracolo per A_{TM}.

6.13 Ricorda il Problema della Corrispondenza di Post che definimmo nella Sezione 5.2 e il suo linguaggio associato PCP. Mostrare che PCP è decidibile rispetto ad A_{TM}.

6.14 Mostrare come calcolare la complessità descrittiva delle stringhe $\mathrm{K}(x)$ con un oracolo per A_{TM}.

6.15 Usare il risultato del Problema 6.14 per fornire una funzione f che è calcolabile con un oracolo per A_{TM}, dove per ogni n, $f(n)$ è una stringa incompressibile di lunghezza n.

6.16 Mostrare che la funzione $\mathrm{K}(x)$ non è una funzione calcolabile.

6.17 Mostrare che l'insieme delle stringhe incompressibili è indecidibile.

6.18 Mostrare che l'insieme delle stringhe incompressibili non contiene alcun sottoinsieme infinito che sia Turing riconoscibile.

***6.19** Mostrare che per ogni c, esistono stringhe x e y tali che $\mathrm{K}(xy) > \mathrm{K}(x) + \mathrm{K}(y) + c$.

6.20 Sia $S = \{\langle M\rangle |\ M$ una TM ed $L(M) = \{\langle M\rangle\}\}$. Mostrare che né S né $\overline{S}$ è Turing riconoscibile.

6.21 Sia $R \subseteq \mathcal{N}^k$ una relazione k-aria. Diciamo che R è ***definibile*** in $\mathrm{Th}(\mathcal{N}, +)$ se possiamo fornire una formula ϕ con k variabili libere $x_1, \ldots, x_k$ tale che per ogni $a_1, \ldots, a_k \in \mathcal{N}$, $\phi(a_1, \ldots, a_k)$ è vera esattamente quando $a_1, \ldots, a_k \in R$. Mostrare che ciascuna delle seguenti relazioni è definibile in $\mathrm{Th}(\mathcal{N}, +)$.

Aa. $R_0 = \{0\}$

b. $R_1 = \{1\}$

c. $R_= = \{(a, a) |\ a \in \mathcal{N}\}$

d. $R_< = \{(a, b) |\ a, b \in \mathcal{N}$ ed $a < b\}$

6.22 Descrivere due diverse macchine di Turing, M ed N, tali che, su un qualsiasi input, M fornisce in output $\langle N \rangle$ ed N fornisce in output $\langle M \rangle$.

6.23 Nella versione del punto fisso del teorema di ricorsione (Teorema 6.8), sia la trasformazione t una funzione che scambia gli stati q_{accept} e q_{reject} nelle descrizioni di una macchina di Turing. Fornire un esempio di un punto fisso per t.

*6.24 Mostrare che $EQ_{\text{TM}} \not\leq_{\text{m}} \overline{EQ_{\text{TM}}}$.

A6.25 Usare il teorema di ricorsione per dare una prova alternativa del teorema di Rice nel Problema 5.16.

A6.26 Fornire un modello per la formula chiusa

$$\begin{aligned}\phi_{\text{eq}} \;=\; & \forall x \left[R_1(x,x) \right] \\ & \wedge \forall x,y \left[R_1(x,y) \leftrightarrow R_1(y,x) \right] \\ & \wedge \forall x,y,z \left[(R_1(x,y) \wedge R_1(y,z)) \rightarrow R_1(x,z) \right].\end{aligned}$$

*6.27 Sia ϕ_{eq} definita come nel Problema 6.26. Fornire un modello per la formula chiusa

$$\begin{aligned}\phi_{\text{lt}} \;=\; & \phi_{\text{eq}} \\ & \wedge \forall x,y \left[R_1(x,y) \rightarrow \neg R_2(x,y) \right] \\ & \wedge \forall x,y \left[\neg R_1(x,y) \rightarrow (R_2(x,y) \oplus R_2(y,x)) \right] \\ & \wedge \forall x,y,z \left[(R_2(x,y) \wedge R_2(y,z)) \rightarrow R_2(x,z) \right] \\ & \wedge \forall x \exists y \left[R_2(x,y) \right].\end{aligned}$$

A6.28 Sia $(\mathcal{N}, <)$ il modello con universo $\mathcal{N}$ e la relazione "minore di". Mostrare che $\text{Th}(\mathcal{N}, <)$ è decidibile.

SOLUZIONI SELEZIONATE

6.3 Supponiamo che M_1^B decida A e M_2^C decida B. Usiamo una TM con oracolo M_3, tale che M_3^C decida A. La macchina M_3 simula M_1. Ogni volta che M_1 interroga il suo oracolo su una qualche stringa x, la macchina M_3 controlla se $x \in B$ e fornisce la risposta a M_1. Poiché la macchina M_3 non ha un oracolo per B e non può compiere questo controllo direttamente, essa simula M_2 sull'input x per ottenere questa informazione. La macchina M_3 può ottenere la risposta alle domande di M_2 direttamente perché queste due macchine usano lo stesso oracolo, C.

6.5 L'enunciato $\exists x \forall y \left[x{+}y{=}y \right]$ è un elemento di $\text{Th}(\mathcal{N}, +)$ perché questo enunciato è vero per l'interpretazione comune di $+$ sull'universo $\mathcal{N}$. Ricorda che utilizziamo $\mathcal{N} = \{0, 1, 2, \ldots\}$ in questo capitolo e quindi possiamo usare $x = 0$. L'enunciato $\exists x \forall y \left[x{+}y{=}x \right]$ non è un elemento di $\text{Th}(\mathcal{N}, +)$ perché questo enunciato non è vero in questo modello. Per ogni valore di x, porre $y = 1$ fa sì che non sia $x{+}y{=}x$.

6.21 (a) R_0 è definibile in $\text{Th}(\mathcal{N}, +)$ mediante $\phi_0(x) = \forall y[x + y = y]$.
(c) $R_=$ è definibile in $\text{Th}(\mathcal{N}, +)$ mediante $\phi_=(u, v) = \forall x[\phi_0(x) \rightarrow x{+}u{=}v]$.

6.25 Supponiamo per assurdo che una qualche TM X decida una proprietà P, e P soddisfi le condizioni del teorema di Rice. Una di queste condizioni dice che esistono TM A e B tali che $\langle A \rangle \in P$ e $\langle B \rangle \notin P$. Usiamo A e B per costruire la TM R:

R = "Sull'input w:

1. Ottiene la sua descrizione $\langle R \rangle$ usando il teorema di ricorsione.
2. Esegue X su $\langle R \rangle$.
3. Se X accetta $\langle R \rangle$, simula B su w.
   Se X rifiuta $\langle R \rangle$, simula A su w."

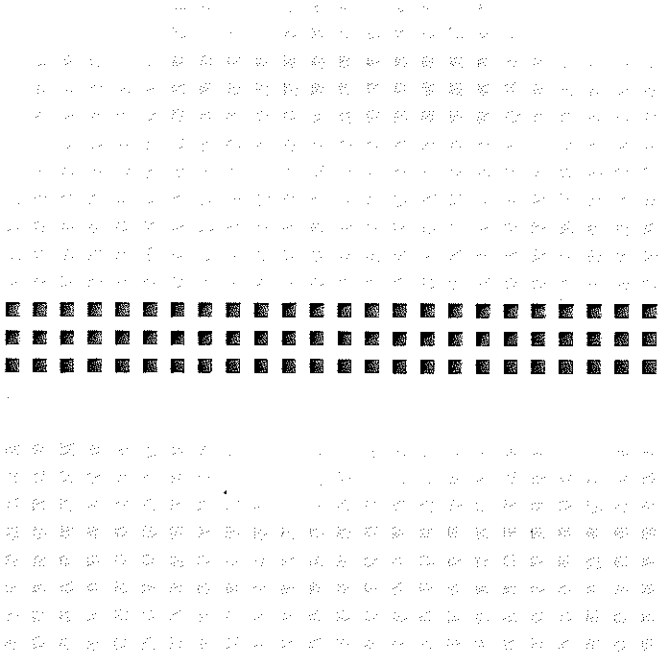
Se $\langle R \rangle \in P$, allora X accetta $\langle R \rangle$ ed $L(R) = L(B)$. Ma $\langle B \rangle \notin P$, in contraddizione con $\langle R \rangle \in P$, poiché P coincide sulle TM che hanno lo stesso linguaggio. Otteniamo una contraddizione simile se $\langle R \rangle \notin P$. Quindi, la nostra iniziale supposizione è falsa. Ogni proprietà che soddisfa le condizioni del teorema di Rice è indecidibile.

6.26 L'enunciato ϕ_{eq} fornisce le tre condizioni di una relazione di equivalenza. Un modello (A, R_1), dove A è un qualsiasi universo ed R_1 è una qualsiasi relazione di equivalenza su A, è un modello per ϕ_{eq}. Per esempio, sia A l'insieme degli interi $\mathcal{Z}$ e sia $R_1 = \{(i, i) | \, i \in \mathcal{Z}\}$.

6.28 Ridurre $\text{Th}(\mathcal{N}, <)$ a $\text{Th}(\mathcal{N}, +)$, che è decidibile, come abbiamo già dimostrato. Mostrare come trasformare una formula chiusa ϕ_1 sul linguaggio di $(\mathcal{N}, <)$ in una formula chiusa ϕ_2 sul linguaggio di $(\mathcal{N}, +)$ preservando la verità o falsità nei rispettivi modelli. Sostituire ogni occorrenza di $i < j$ in ϕ_1 con la formula $\exists k \left[(i{+}k{=}j) \wedge (k{+}k{\neq}k) \right]$ in ϕ_2, dove k è una diversa nuova variabile ogni volta.

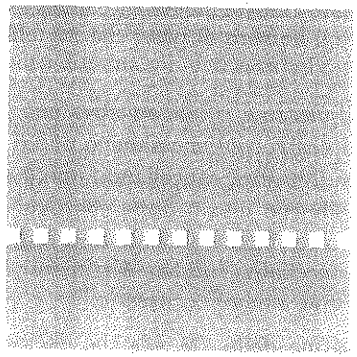
La formula chiusa ϕ_2 è equivalente a ϕ_1 poiché "i è minore di j" significa che possiamo addizionare un valore diverso da zero a i e ottenere j. Porre ϕ_2 in forma normale prenex, come richiesto dall'algoritmo per decidere $\text{Th}(\mathcal{N}, +)$, richiede un pó di lavoro supplementare. I nuovi quantificatori esistenziali sono portati alla sinistra della formula chiusa. Per farlo, questi quantificatori devono passare attraverso le operazioni booleane che sono presenti nella formula chiusa. I quantificatori possono essere portati attraverso le operazioni di $\wedge$ e $\vee$ senza cambiamenti. Attraversare $\neg$ cambia $\exists$ in $\forall$ e viceversa. Quindi, $\neg \exists k \, \psi$ diventa l'espressione equivalente $\forall k \neg \psi$, e $\neg \forall k \, \psi$ diventa $\exists k \neg \psi$.

PARTE TERZA



TEORIA DELLA COMPLESSITÀ

7



COMPLESSITÀ DI TEMPO

Anche quando un problema è decidibile, e quindi in linea di principio computazionalmente risolvibile, può non essere risolvibile in pratica se la soluzione richiede una quantità eccessiva di tempo o di memoria. In questa parte finale del libro introduciamo la teoria della complessità computazionale – uno studio del tempo, della memoria o di altre risorse, necessarie per risolvere problemi computazionali. Iniziamo dal tempo.

Il nostro obiettivo in questo capitolo è presentare le basi della teoria della complessità di tempo. Come primo passo, introduciamo un metodo per misurare il tempo usato per risolvere un problema. Poi mostriamo come classificare i problemi in base alla quantità di tempo che essi richiedono. Infine, consideriamo la possibilità che determinati problemi decidibili richiedano enormi quantità di tempo e di come sia possibile determinare quando ci troviamo di fronte a un problema di questo tipo.

7.1
MISURE DI COMPLESSITÀ

Iniziamo con un esempio. Consideriamo il linguaggio $A = \{0^k1^k \mid k \geq 0\}$. Ovviamente, A è un linguaggio decidibile. Di quanto tempo necessita una una macchina di Turing a nastro singolo per decidere A? Esaminiamo la

seguente TM M_1 per A. Diamo la descrizione della macchina di Turing a basso livello, includendo l'esatto movimento della testina sul nastro in modo che possiamo contare il numero di passi che M_1 effettua quando lavora.

M_1 = "Su input w:

1. Scandisce il nastro e *rifiuta* se trova uno 0 a destra di un 1.
2. Ripete se il nastri contiene almeno uno 0 ed almeno un 1:
3. Scandisce il nastro, cancellando uno 0 ed un 1.
4. Se rimane almeno uno 0 dopo che ogni simbolo 1 è stato cancellato, o se rimane almeno un 1 dopo che ogni simbolo 0 è stato cancellato, *rifiuta*. Altrimenti, se non rimangono né simboli 0 né simboli 1, *accetta*."

Analizzeremo l'algoritmo della TM M_1 che decide A per determinare la quantità di tempo che impiega. Come prima cosa, introduciamo la terminologia e la notazione necessaria. Il numero di passi che utilizza un algoritmo su un particolare input può dipendere da diversi parametri. Ad esempio, se l'input è un grafo, il numero di passi può dipendere dal numero di nodi, dal numero di archi e dal grado massimo del grafo, o da una combinazione di questi e/o altri parametri. Per semplicità si calcola il tempo di esecuzione di un algoritmo semplicemente in funzione della lunghezza della stringa che rappresenta l'input e non si considerano eventuali altri parametri. Nell'analisi del ***caso peggiore***, che noi considereremo, si valuta il tempo di esecuzione massimo tra tutti gli input di una determinata lunghezza. Nell'analisi del ***caso medio***, si considera la media dei tempi di esecuzione su tutti gli input di una determinata lunghezza.

DEFINIZIONE **7.1**

Sia M una macchina di Turing deterministica che si ferma su tutti gli input. Il ***tempo di esecuzione*** o la ***complessità di tempo*** di M è la funzione $f: \mathcal{N} \longrightarrow \mathcal{N}$, dove $f(n)$ è il numero massimo di passi che M utilizza su un qualsiasi input di lunghezza n. Se $f(n)$ è il tempo di esecuzione di M, diciamo che M ha tempo di esecuzione $f(n)$ e che M è una macchina di Turing di tempo $f(n)$. Abitualmente usiamo n per rappresentare la lunghezza dell'input.

Notazione O-grande ed o-piccola

Poichè il tempo esatto di esecuzione di un algoritmo è spesso un'espressione complessa, solitamente ci limitiamo ad ottenerne una stima. L'utile metodo di stima, detto ***analisi asintotica***, permette di valutare il tempo di esecuzione dell'algoritmo quando viene eseguito su grandi input. Lo

facciamo prendendo in considerazione solo il termine di ordine maggiore dell'espressione del tempo di esecuzione dell'algoritmo, trascurando sia il coefficiente di tale termine, che tutti i termini di ordine inferiore, perché il termine di ordine più alto domina gli altri termini quando l'input è grande.

Ad esempio, la funzione $f(n) = 6n^3 + 2n^2 + 20n + 45$ ha quattro termini ed il termine di ordine maggiore è $6n^3$. Trascurando il coefficiente 6, diciamo che f è asintoticamente al più n^3. La ***notazione asintotica*** o notazione ***O-grande*** per descrivere questo rapporto è $f(n) = O(n^3)$.

Formalizziamo tale nozione nella seguente definizione. Sia $\mathcal{R}^+$ l'insieme dei numeri reali non negativi.

DEFINIZIONE **7.2**

Siano f e g funzioni $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Si dice che $\boldsymbol{f(n) = O(g(n))}$ se esistono interi positivi c e n_0 tali che per ogni $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

Quando $f(n) = O(g(n))$, diciamo che $g(n)$ è un ***limite superiore*** per $f(n)$, o più precisamente, che $g(n)$ è un ***limite superiore asintotico*** per $f(n)$, per sottolineare che stiamo ignorando le costanti.

Intuitivamente, $f(n) = O(g(n))$ significa che f è minore o uguale a g se trascuriamo differenze fino ad un fattore costante. Potete pensare ad O come rappresentazione implicita di una costante. In pratica, la maggior parte delle funzioni f che potete incontrate hanno un termine di ordine più alto h. chiaramente individuabile. In tal caso si scrive $f(n) = O(g(n))$, dove g è h senza il suo coefficiente.

ESEMPIO 7.3

Sia $f_1(n)$ la funzione $5n^3 + 2n^2 + 22n + 6$. Quindi, selezionando il termine di ordine maggiore $5n^3$ e trascurando il suo coefficiente 5, si ottiene $f_1(n) = O(n^3)$. Verifichiamo che questo risultato soddisfa la definizione formale. Lo facciamo ponendo c pari a 6 e n_0 a 10. Quindi $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ per ogni $n \geq 10$. Inoltre, $f_1(n) = O(n^4)$ perché n^4 è più grande di n^3 e quindi è un limite superiore asintotico per f_1. Tuttavia, $f_1(n)$ non è $O(n^2)$. Indipendentemente dai valori che assegniamo a c e n_0, la definizione non viene soddisfatta in questo caso.

ESEMPIO 7.4

L'O-grande interagisce con i logaritmi in una maniera particolare. Di solito quando si usano i logaritmi dobbiamo specificarne la base, come per $x =$

$\log_2 n$. La base 2 qui indica che questa uguaglianza equivale all'uguaglianza $2^x = n$. Cambiando il valore della base b cambia il valore di $\log_b n$ di un fattore costante, dovuto all'uguaglianza $\log_b n = \log_2 n / \log_2 b$. Quindi, quando scriviamo $f(n) = O(\log n)$, non è più necessario specificare la base perché stiamo comunque eliminando i fattori costanti. Sia $f_2(n)$ la funzione $3n \log_2 n + 5n \log_2 \log_2 n + 2$. In questo caso abbiamo $f_2(n) = O(n \log n)$ perché $\log n$ domina $\log \log n$.

La notazione O-grande appare anche in espressioni aritmetiche, come l'espressione $f(n) = O(n^2) + O(n)$. In tal caso ciascuna occorrenza del simbolo O rappresenta un diverso fattore costante nascosto. Poichè il termine $O(n^2)$ domina il termine $O(n)$, l'espressione è equivalente a $f(n) = O(n^2)$. Quando il simbolo O si presenta all'esponente, come nell'espressione $f(n) = 2^{O(n)}$, vale la stessa idea. Questa espressione rappresenta un limite superiore per 2^{cn} per qualche costante c. A volte si trova l'espressione $f(n) = 2^{O(\log n)}$. Utilizzando l'uguaglianza $n = 2^{\log_2 n}$ e quindi $n^c = 2^{c \log_2 n}$, otteniamo che $2^{O(\log n)}$ rappresenta un limite superiore per n^c per qualche c. L'espressione $n^{O(1)}$ rappresenta la stessa limitazione in maniera diversa, in quanto l'espressione $O(1)$ rappresenta un valore che non è mai maggiore di una costante fissata.

Spesso si ottengono dei limiti della forma n^c per c maggiore di 0. Tali limiti sono chiamati ***limiti polinomiali***. Limiti del tipo $2^{(n^\delta)}$ sono chiamati ***limiti esponenziali*** quando δ è un numero reale maggiore di 0.

La notazione O-grande ha una controparte chiamata ***notatione o-piccolo***. La notazione O-grande dice che una funzione è asintoticamente *non più grande* di un'altra. Per dire che una funzione è asintoticamente *più piccola* di un'altra, usiamo la notazione o-piccolo. La differenza tra le notazioni O-grande e o-piccolo è analoga alla differenza tra $\leq$ e $<$.

DEFINIZIONE **7.5**

Siano f e g funzioni $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Diciamo che $\boldsymbol{f(n) = o(g(n))}$ se

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In altri termini, $f(n) = o(g(n))$ significa che per ogni numero reale $c > 0$, esiste un numero n_0, tale che $f(n) < c\,g(n)$ per ogni $n \geq n_0$.

ESEMPIO 7.6

Le seguenti uguaglianze sono semplici da verificare.

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.

Tuttavia, $f(n)$ non è mai $o(f(n))$.

Analisi degli algoritmi

Analizziamo l'algoritmo M_1 dato per il linguaggio $A = \{0^k 1^k \mid k \geq 0\}$. Riscriviamo qui l'algoritmo per comodità.

M_1 = "Su input w:

1. Scandisce il nastro e *rifiuta* se trova uno 0 a destra di un 1.
2. Ripete se il nastro contiene almeno uno 0 ed almeno un 1:
3. Scandisce il nastro, cancellando uno 0 ed un 1.
4. Se rimane almeno uno 0 dopo che ogni simbolo 1 è stato cancellato, o se rimane almeno un 1 dopo che ogni simbolo 0 è stato cancellato, *rifiuta*. Altrimenti, se non rimangono né simboli 0 né simboli 1, *accetta*."

Per analizzare M_1, consideriamo ciascuna delle sue quattro fasi separatamente. Nella fase 1, la macchina scansiona il nastro per verificare che l'input è del tipo 0^*1^*. Tale operazione di scansione usa n passi. Come abbiamo accennato in precedenza, di solito utilizziamo n per rappresentare la lunghezza dell'input. Per riposizionare la testina all'estremità sinistra del nastro utilizza ulteriori n passi. Per cui il totale di passi utilizzati in questa fase è $2n$ passi. Nella notazione O-grande diciamo che questa fase usa $O(n)$ passi. Si noti che non abbiamo fatto menzione del riposizionamento della testina del nastro nella descrizione della macchina. L'utilizzo della notazione asintotica ci permette di omettere quei dettagli della descrizione della macchina che influenzano il tempo di esecuzione al più di un fattore costante.

Nelle fasi 2 e 3, la macchina esegue ripetutamente la scansione del nastro e cancella uno 0 e un 1 ad ogni scansione. Ogni scansione utilizza $O(n)$ passi. Poichè ogni scansione elimina due simboli, possono verificarsi al più $n/2$ scansioni. Così il tempo totale impiegato dalle fasi 2 e 3 è di $(n/2)O(n) = O(n^2)$ passi.

Nella fase 4 la macchina fa una singola scansione per decidere se accettare o rifiutare. Il tempo impiegato in questa fase è al più $O(n)$.

Quindi il tempo totale di M_1 su un input di lunghezza n è $O(n)+O(n^2)+O(n)$, ovvero $O(n^2)$. In altre parole, il tempo di esecuzione è $O(n^2)$, il che completa l'analisi del tempo di esecuzione di questa macchina.

Vogliamo ora stabilire una notazione per classificare i linguaggi in base alla loro necessità di tempo.

DEFINIZIONE **7.7**

Sia $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ una funzione. La ***classe di complessità di tempo***, **TIME($t(n)$)**, è definita come l'insieme di tutti i linguaggi che sono decisi da una macchina di Turing in tempo $O(t(n))$.

Ricordiamo il linguaggio $A = \{0^k1^k \mid k \geq 0\}$. L'analisi precedente mostra che $A \in \text{TIME}(n^2)$ perché M_1 decide A in tempo $O(n^2)$ e TIME(n^2) contiene tutti i linguaggi che possono essere decisi in tempo $O(n^2)$.

Esiste una macchina che decide A in modo asintoticamente più veloce? In altri termini, risulta A in TIME($t(n)$) per $t(n) = o(n^2)$? Possiamo migliorare il tempo di esecuzione, cancellando due simboli 0 e due simboli 1 ad ogni scansione invece di uno solamente, perché questo riduce il numero di scansioni della metà. Ma ciò migliora il tempo di esecuzione solo per un fattore 2 e non influenza il tempo di esecuzione asintotico. La seguente macchina M_2, utilizza un metodo differente per decidere A asintoticamente più velocemente. Essa mostra che $A \in \text{TIME}(n \log n)$.

M_2 = "Su input w:

1. Scandisce il nastro e *rifiuta* se trova uno 0 a destra di un 1.
2. Ripete finché il nastro contiene almeno uno 0 ed un 1:
3. Scandisce il nastro, controllando se il numero totale di simboli 0 e di 1 rimasti è pari o dispari. Se è dispari, *rifiuta*.
4. Scandisce nuovamente il nastro, cancellando prima ogni secondo 0 a partire dal primo 0, poi cancellando ogni secondo 1 a partire dal primo 1.
5. Se nessuno 0 e nessun 1 rimangono sul nastro, *accetta*. Altrimenti, *rifiuta*."

Prima di analizzare M_2, cerchiamo di verificare se effettivamente decide A. In ogni scansione eseguita nella fase 4, il numero totale di 0 rimanenti è ridotto della metà e ogni eventuale resto viene scartato. Quindi, se abbiamo iniziato con 13 simboli 0, dopo una prima esecuzione della fase 4 rimangono solo 6 simboli 0. Dopo le successive esecuzioni di questa fase ne restano

prima 3, poi 1, e poi 0. Questa fase ha lo stesso effetto sul numero di simboli 1. Esaminiamo ora la parità (pari/dispari) del numero di simboli 0 e 1 ad ogni esecuzione della fase 3. Si consideri ancora l'inizio con 13 simboli 0 e 13 simboli 1. La prima esecuzione della fase 3 trova un numero dispari di simboli 0 (perché 13 è un numero dispari) ed un numero dispari di simboli 1. Nelle successive esecuzioni si ha un numero pari (6), poi un numero dispari (3), e un numero dispari (1). Non eseguiamo questa fase su 0 simboli 0 o 0 simboli 1 in accordo alla condizione specificata nella fase 2. Per la sequenza di parità trovata (dispari, pari, dispari, dispari), se sostituiamo pari con 0 e dispari 1 e poi invertiamo la sequenza, otteniamo 1101, la rappresentazione binaria di 13, ossia il numero di simboli 0 e 1 iniziale. La sequenza delle parità fornisce sempre l'inverso della rappresentazione binaria.

Quando la fase 3 controlla che il numero totale di 0 e 1 rimanenti è pari, in realtà sta controllando la coerenza della parità del numero di 0 con la parità del numero di 1. Se le parità corrispondono, le rappresentazioni binarie dei numeri di 0 e di 1 corrispondono, e quindi i due numeri sono uguali.

Per analizzare il tempo di esecuzione di M_2, per prima cosa osserviamo che ogni fase impiega un tempo $O(n)$. Determiniamo poi il numero di volte in cui ognuna viene eseguita. Le fasi 1 e 5 vengono eseguite una volta, impiegando un tempo totale $O(n)$. La fase 4 scarta almeno metà dei simboli 0 e 1 ogni volta che viene eseguita, quindi si verificano al massimo $1+\log_2 n$ iterazioni del ciclo prima di averli cancellati tutti. Così il tempo totale delle fasi 2,3, e 4 è $(1+\log_2 n)O(n)$, o $O(n \log n)$. Il tempo di esecuzione di M_2 è $O(n) + O(n \log n) = O(n \log n)$.

In precedenza abbiamo dimostrato che $A \in \text{TIME}(n^2)$, ma ora abbiamo ottenuto un limite migliore – precisamente, $A \in \text{TIME}(n \log n)$. Questo risultato non può essere ulteriormente migliorato su macchine di Turing a singolo nastro. Infatti, ogni linguaggio che può essere deciso in tempo $o(n \log n)$ su una macchina di Turing a nastro singolo è regolare, come vi chiede di mostrare il Problema 7.20.

Possiamo decidere il linguaggio A in tempo $O(n)$ (chiamato anche ***tempo lineare***) se la macchina di Turing ha un secondo nastro. La seguente TM M_3 a due nastri decide A in tempo lineare. La macchina M_3 opera diversamente dalle macchine precedenti per A. Semplicemente copia tutti i simboli 0 sul suo secondo nastro e poi li accoppia con gli 1.

M_3 = "Su input w:

1. Scandisce il nastro 1 e *rifiuta* se trova uno 0 a destra di un 1.
2. Scandisce i simboli 0 sul nastro 1 fino al primo 1. Contemporaneamente, copia ogni 0 sul nastro 2.
3. Scandisce i simboli 1 sul nastro 1 fino alla fine dell'input. Per ogni 1 letto sul nastro 1, cancella uno 0 sul nastro 2.

Se ogni 0 è stato cancellato prima di aver letto tutti gli 1, *rifiuta*.

4. Se tutti gli 0 sono stati cancellati, *accetta*. Se rimane qualche 0, *rifiuta*."

Questa macchina è semplice da analizzare. Ciascuna delle quattro fasi utilizza $O(n)$ passi, in modo che il tempo di esecuzione complessivo risulta $O(n)$ e quindi lineare. Si noti che questo tempo di esecuzione è il migliore possibile perché n passi sono necessari semplicemente per leggere l'input.

Riassumiamo quello che abbiamo dimostrato circa la complessità temporale di A, la quantità di tempo necessaria per decidere A. Abbiamo progettato una TM M_1 a singolo nastro che decide A in tempo $O(n^2)$ ed una TM M_2 a singolo nastro che decide A in tempo $O(n \log n)$. La soluzione del Problema 7.20 implica che nessuna TM a nastro singolo può farlo più velocemente. Poi abbiamo progettato una TM M_3 a due nastri che decide A in tempo $O(n)$. Quindi la complessità di tempo di A è $O(n \log n)$ su una TM a nastro singolo e $O(n)$ su una TM a due nastri. Notate che la complessità di A dipende dal modello di calcolo scelto.

Questa discussione evidenzia una differenza importante tra la teoria della complessità e la teoria della computabilità. Nella teoria della computabilità, la tesi di Church-Turing implica che tutti i modelli computazionali sono equivalenti – ossia che tutti decidono la stessa classe di linguaggi. Nella teoria della complessità, la scelta del modello influisce sulla complessità di tempo dei linguaggi. Per esempio, i linguaggi decidibili in tempo lineare in un modello non sono necessariamente decidibili in tempo lineare in un altro modello. Nella teoria della complessità, classifichiamo i problemi computazionali secondo la loro complessità di tempo. Ma con quale modalità misuriamo il tempo? Uno stesso linguaggio può avere differenti necessità di tempo in modelli diversi. Fortunatamente, i requisiti di tempo non differiscono molto per i modelli deterministici usuali. Quindi, se il nostro sistema di classificazione non è molto sensibile a differenze relativamente piccole nella complessità, la scelta dello specifico modello deterministico non è importante. Discuteremo questo concetto ulteriormente nelle prossime sezioni.

Relazioni di complessità tra modelli

Qui esaminiamo come la scelta del modello di calcolo può influenzare la complessità di tempo dei linguaggi. Consideriamo tre modelli: macchine di Turing a singolo nastro, macchine di Turing a due nastri e macchine di Turing non deterministiche.

TEOREMA 7.8

Sia $t(n)$ una funzione, tale che $t(n) \geq n$. Ogni macchina di Turing multinastro di tempo $t(n)$ ammette una macchina di Turing equivalente a nastro singolo di tempo $O(t^2(n))$.

IDEA. L'idea alla base della dimostrazione di questo teorema è molto semplice. Ricordiamo che nel Teorema 3.13, abbiamo mostrato come convertire qualsiasi TM multinastro in una TM a nastro singolo che la simula. Ora analizziamo la simulazione per determinare la quantità di tempo supplementare richiesta. Mostriamo che possiamo simulare ogni passo della macchina multinastro con $O(t(n))$ passi della macchina a nastro singolo. Per cui il tempo totale impiegato è $O(t^2(n))$ passi.

DIMOSTRAZIONE. Sia M una TM a k-nastri avente tempo di esecuzione $t(n)$. Costruiamo una TM S a singolo nastro che ha tempo di esecuzione $O(t^2(n))$. La macchina S opera simulando M, come descritto nel Teorema 3.13. Nel rivedere tale simulazione, ricordiamo che S utilizza il suo unico nastro per rappresentare il contenuto di tutti i k nastri di M. I nastri sono memorizzati consecutivamente, con le posizioni delle testine di M marcate nelle celle appropriate. Inizialmente, S mette il nastro nel formato che rappresenta tutti i nastri di M e poi simula i passi di M. Per simulare un passo, S scorre tutte le informazioni memorizzate sul suo nastro per determinare i simboli presenti sotto le testine di M. Poi S esegue un'altra scansione del suo nastro per aggiornare il contenuto dei nastri e le posizioni delle testine. Se una testina di M si sposta a destra su una parte non letta del nastro, S deve aumentare la quantità di spazio allocato per questo nastro. Lo fa spostando una parte del suo nastro di una cella a destra.

Ora analizziamo questa simulazione. Per ogni passo di M, la macchina S fa due passi sulla parte attiva del suo nastro. Il primo ottiene le informazioni necessarie per determinare la prossima mossa e il secondo la esegue. La lunghezza della parte attiva del nastro di S determina il tempo che S impiega per eseguire la scansione, quindi dobbiamo determinare un limite superiore per questa lunghezza. Per farlo prendiamo la somma delle lunghezze delle parti attive dei k nastri di M. Ciascuna di queste parti attive ha lunghezza al più $t(n)$ poichè M utilizza $t(n)$ celle del nastro in $t(n)$ passi, se la testina si sposta verso destra ad ogni passo e anche meno se vi sono spostamenti di qualche testina a sinistra. Quindi una scansione della parte di nastro attiva di S impiega $O(t(n))$ passi. Per simulare ciascuna delle fasi di M, S esegue due scansioni ed eventualmente fino a k spostamenti a destra. Ognuno impiega un tempo $O(t(n))$, per cui il tempo totale per S per simulare un passo di M è $O(t(n))$. Ora possiamo limitare il tempo totale impiegato dalla simulazione. La fase iniziale, in cui S mette il nastro nel formato corretto, usa $O(n)$ passi. In seguito, S simula ciascuno dei $t(n)$ passi di M, utilizzando $O(t(n))$ passi, per cui questa parte della simulazione utilizza $t(n) \times O(t(n)) = O(t^2(n))$ passi. Quindi l'intera simulazione di M utilizza $O(n) + O(t^2(n))$ passi.

Abbiamo assunto che $t(n) \geq n$ (un'ipotesi ragionevole perché M non potrebbe nemmeno leggere l'intero input in meno tempo). Pertanto il tempo di esecuzione di S è $O(t^2(n))$ e la dimostrazione è completa.

Ora vedremo un teorema analogo nel caso di macchine di Turing non deterministiche a nastro singolo. Mostreremo che ogni linguaggio decidibile su tale macchina è anche decidibile su una macchina di Turing deterministica a nastro singolo che richiede molto più tempo. Prima di farlo, dobbiamo definire il tempo di esecuzione di una macchina di Turing non deterministica. Ricordiamo che una macchina di Turing non deterministica è un decisore se tutte le sue computazioni si fermano su tutti gli input.

DEFINIZIONE **7.9**

Sia N una macchina di Turing non deterministica che sia anche un decisore. Il ***tempo di esecuzione*** di N è la funzione $f: \mathcal{N} \longrightarrow \mathcal{N}$, tale che $f(n)$ è il massimo numero di passi che N usa per ognuna delle computazioni su ogni input di lunghezza n, come mostrato nella figura seguente.

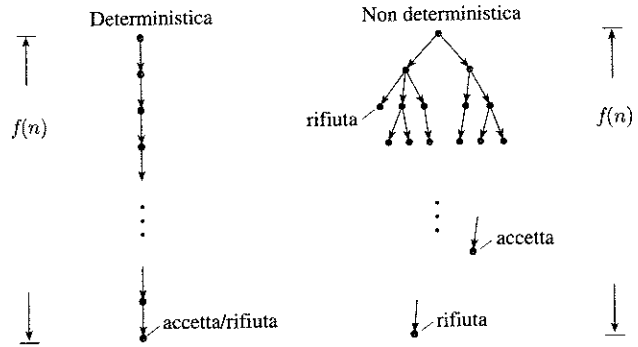


FIGURA 7.10
Misurazione del tempo nei casi deterministico e non deterministico

La definizione del tempo di esecuzione di una macchina di Turing non deterministica non è destinata a corrispondere ad un qualche dispositivo informatico reale. Piuttosto, si tratta di un'utile definizione matematica che aiuta a caratterizzare la complessità di una classe importante di problemi computazionali, come dimostreremo a breve.

TEOREMA 7.11

Sia $t(n)$ una funzione tale che $t(n) \geq n$. Ogni macchina di Turing non deterministica a singolo nastro avente tempo di esecuzione $t(n)$ ha una macchina di Turing deterministica a nastro singolo equivalente di tempo $2^{O(t(n))}$.

DIMOSTRAZIONE. Sia N una TM non deterministica avente tempo di esecuzione $t(n)$. Costruiamo una TM deterministica D che simula N, come nel Teorema 3.16, effettuando una ricerca sull'albero delle computazioni di N. Ora analizziamo tale simulazione. Su un input di lunghezza n, ogni ramificazione dell'albero delle computazioni di N ha lunghezza al più $t(n)$. Ogni nodo dell'albero può avere al più b figli, dove b è il massimo numero di scelte possibili in accordo alla funzione di transizione di N. Così il numero totale di foglie nell'albero è al massimo $b^{t(n)}$. La simulazione procede esplorando l'albero prima in ampiezza. In altre parole, si visitano tutti i nodi a profondità d prima di passare ad uno qualsiasi dei nodi a profondità $d+1$. L'algoritmo riportato nella dimostrazione del Teorema 3.16 inizia inefficientemente dalla radice e si sposta in basso verso un nodo ogni volta che visita il nodo stesso. Tuttavia l'eliminanzione di tale inefficienza non altera l'enunciato del Teorema, quindi la lasciamo in questa forma. Il numero totale di nodi dell'albero è inferiore al doppio del numero di foglie, quindi è limitato da $O(b^{t(n)})$. Il tempo per partire dalla radice e raggiungere un nodo è $O(t(n))$. Pertanto il tempo di esecuzione di D è $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

Come descritto nel Teorema 3.16, la TM D ha tre nastri. Convertirla in una TM a nastro singolo al più fa sì che si elevi al quadrato il tempo di esecuzione, per il Teorema 7.8. Quindi il tempo di esecuzione del simulatore a nastro singolo è $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$ ed il teorema è dimostrato.

7.2
LA CLASSE P

I Teoremi 7.8 e 7.11 illustrano una distinzione importante. Da una parte, abbiamo dimostrato una differenza al più quadratica o *polinomiale* tra le complessità di tempo dei problemi misurati su macchine di Turing deterministiche a singolo nastro e multinastro. D'altra parte, abbiamo mostrato una differenza al più *esponenziale* tra la complessità temporale di problemi su macchine di Turing deterministiche e non deterministiche.

Tempo polinomiale

Per i nostri scopi, differenze polinomiali nel tempo di esecuzione sono considerate piccole, mentre differenze esponenziali sono considerate grandi. Diamo un'occhiata al motivo per cui abbiamo scelto di fare questa separazione tra polinomi ed esponenziali, piuttosto che tra altre classi di funzioni. In primo luogo, si noti la differenza drastica tra il tasso di crescita di polinomi che si incontrano usualmente come ad esempio n^3 e di quella di un esponenziale quale 2^n. Per esempio, sia n uguale a 1000, la dimensione di un input ragionevole per un algoritmo. In tal caso, n^3 è 1 miliardo, un numero grande, ma gestibile, mentre 2^n è un numero molto più grande del numero di atomi dell'universo. Algoritmi aventi tempo polinomiale sono abbastanza veloci per molti scopi, ma algoritmi aventi tempo esponenziale sono raramente utili. Algoritmi aventi tempo esponenziale si presentano in genere quando risolviamo problemi mediante una ricerca esaustiva nello spazio delle soluzioni, denominata ***ricerca mediante forza bruta.*** Ad esempio, un modo per fattorizzare un numero nei suoi fattori primi consiste nel cercare attraverso tutti i suoi possibili divisori. La dimensione dello spazio di ricerca è esponenziale, quindi questa ricerca impiega un tempo esponenziale. A volte, la ricerca mediante forza bruta può essere evitata attraverso una comprensione più approfondita del problema, che può suggerire un algoritmo polinomiale di maggiore utilità.

Tutti i modelli computazionali deterministici ragionevoli sono ***polinomialmente equivalenti.*** Cioè, uno di essi può simularne un altro con aumento solo polinomiale del tempo di esecuzione. Quando diciamo che tutti i modelli deterministici ragionevoli sono polinomialmente equivalenti, non cerchiamo di definire il termine *ragionevole.* Tuttavia, abbiamo in mente una nozione abbastanza ampia da includere modelli che approssimano molto da vicino i tempi di esecuzione dei computer reali. Ad esempio, il Teorema 7.8 mostra che le macchine di Turing deterministiche a nastro singolo e multinastro sono polinomialmente equivalenti. Da qui in poi ci concentreremo sugli aspetti della teoria della complessità temporale che non sono influenzati da differenze polinomiali del tempo di esecuzione. Ignorando tali differenze, possiamo sviluppare la teoria in un modo che non dipenda dalla scelta di un particolare modello di computazione. Ricordiamo che il nostro obiettivo è presentare le proprietà fondamentali della *computazione*, piuttosto che le proprietà delle macchine di Turing o di un qualsiasi altro modello particolare. Si può pensare che non tener conto di differenze polinomiali nel tempo di esecuzione è assurdo. Programmatori reali hanno certamente a cuore tali differenze e lavorano sodo per permettere ai loro programmi di essere eseguiti anche soltanto due volte più velocemente. Tuttavia, abbiamo trascurato i fattori costanti quando in precedenza abbiamo introdotto la notazione asintotica. Ora ci proponiamo di ignorare

differenze polinomiali, molto più grandi, come quella tra il tempo n ed il tempo n^3.

La nostra decisione di non tener conto delle differenze polinomiali non significa che consideriamo tali differenze non importanti. Al contrario, certamente consideriamo importante la differenza tra il tempo n ed il tempo n^3. Ma alcune questioni, come ad esempio l'essere polinomiale o non polinomiale del problema della fattorizzazione, non dipendono da differenze di tipo polinomiale e sono ugualmente importanti. Abbiamo semplicemente scelto di concentrarci su questo tipo di questioni. Ignorando gli alberi per guardare la foresta non significa che una è più importante degli altri – fornisce soltanto una diversa prospettiva.

Veniamo ora ad una definizione importante nella teoria della complessità.

DEFINIZIONE **7.12**

P è la classe di linguaggi che sono decidibili in tempo polinomiale su una macchina di Turing deterministica a singolo nastro. In altre parole,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

La classe P ha un ruolo centrale nella nostra teoria ed è importante perché

1. P è invariante per tutti i modelli di calcolo che sono polinomialmente equivalenti ad una macchina di Turing deterministica a nastro singolo, e
2. P corrisponde approssimativamente alla classe dei problemi che sono realisticamente risolvibili su un computer.

Il punto 1 indica che P è una classe matematicamente robusta. Non è influenzata dai particolari del modello di computazione che stiamo usando. Il punto 2 indica che P è importante da un punto di vista pratico. Quando un problema è in P, abbiamo un metodo per risolverlo che viene eseguito in tempo n^k, per una qualche costante k. Se questo tempo di esecuzione risulta pratico dipende da k e dall'applicazione. Ovviamente, un tempo di esecuzione di n^{100} è improbabile che sia di qualche utilità pratica. Tuttavia, fissare la soglia di risolubilità in pratica al tempo polonomiale si è dimostrato utile. Quando è stato trovato un algoritmo polinomiale per un problema che precedentemente sembrava richiedere tempo esponenziale, sono state acquisite su di esso alcune conoscenze chiave, e di solito seguono ulteriori riduzioni della sua complessità, spesso fino al punto di avere una reale utilità pratica.

Esempi di problemi in P

Quando presentiamo un algoritmo polinomiale, diamo una sua descrizione di alto livello senza alcun riferimento alle caratteristiche di un particolare modello computazionale. In questo modo evitiamo noiosi dettagli su nastri e movimenti della testina. Seguiamo alcune convenzioni per descrivere un algoritmo in modo che possiamo analizzarlo per polinomialità. Continuiamo a descrivere algoritmi mediante fasi numerate. Ora abbiamo bisogno di tener presente il numero di passi eseguiti dalla macchina di Turing per eseguire ogni fase, così come il numero totale di fasi dell'algoritmo.

Quando analizziamo un algoritmo per dimostrare che esso viene eseguito in tempo polinomiale, abbiamo bisogno di fare due cose. In primo luogo, dobbiamo dare un limite superiore polinomiale (in genere nella notazione O-grande) sul numero di fasi che l'algoritmo esegue quando viene eseguito su un input di lunghezza n. Poi, dobbiamo esaminare le singole fasi nella descrizione dell'algoritmo per essere sicuri che ognuna può essere implementata in tempo polinomiale su un modello deterministico ragionevole. Utilizziamo le fasi nel descrivere l'algoritmo per rendere questa seconda parte dell'analisi più semplice da eseguire.

Quando entrambi i compiti sono stati completati, possiamo concludere che l'algoritmo ha un tempo di esecuzione polinomiale perché abbiamo dimostrato che viene eseguito per un numero polinomiale di fasi, ciascuna delle quali può essere eseguita in tempo polinomiale, e la composizione di polinomi è ancora un polinomio.

Un punto che richiede attenzione è il metodo di codifica usato per i problemi. Continuiamo ad usare la notazione mediante parentesi angolari $\langle \cdot \rangle$ per indicare una codifica mediante una stringa ragionevole di uno o più oggetti, senza specificare un particolare metodo di codifica. Ora, un metodo ragionevole è quello che consente tempi polinomiali di codifica e decodifica di oggetti in rappresentazioni interne naturali o in altre codifiche ragionevoli. Metodi di codifica familiari per grafi, automi ed oggetti simili sono tutti ragionevoli.

Tuttavia, notate che la notazione unaria per la codifica dei numeri (come per il numero 17 codificato dalla stringa unaria 11111111111111111) non è ragionevole, perché è esponenzialmente più grande di una codifica realmente ragionevole, come la notazione in base k, per ogni $k \geq 2$.

Molti dei problemi computazionali che incontrerete in questo capitolo contengono codifiche di grafi. Una codifica ragionevole di un grafo consiste in un elenco dei suoi nodi ed archi. Un altro è la ***matrice di adiacenza***, dove l'elemento (i, j)-esimo è 1 se c'è un arco dal nodo i al nodo j e 0 se non c'è. Quando analizziamo algoritmi su grafi, il tempo di esecuzione può essere calcolato in termini del numero di nodi invece della dimensione della rappresentazione del grafo. In rappresentazioni ragionevoli di grafi, la

dimensione della rappresentazione è un polinomio nel numero di nodi. Così, se analizziamo un algoritmo e mostriamo che il suo tempo di esecuzione è polinomiale (o esponenziale) nel numero di nodi, sappiamo che esso è anche polinomiale (o esponenziale) nella dimensione dell'input.

Il primo problema riguarda grafi orientati. Un grafo orientato G contiene nodi s e t, come mostrato nella figura seguente. Il problema *PATH* consiste nel determinare, se esiste, un cammino diretto s a t. Sia

$$PATH = \{\langle G, s, t\rangle \mid G \text{ è un grafo orientato che contiene un cammino diretto da } s \text{ a } t\}.$$

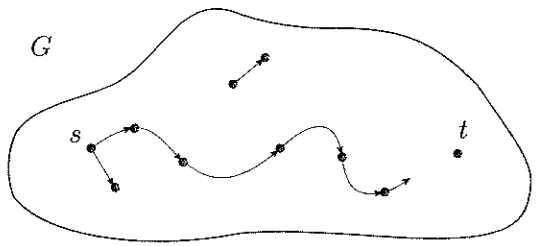


FIGURA 7.13
Il problema *PATH*: esiste un cammino da s a t?

TEOREMA 7.14

$PATH \in$ P.

IDEA. Dimostriamo questo teorema presentando un algoritmo di tempo polinomiale che decide *PATH*. Prima di descrivere l'algoritmo, notiamo che un algoritmo di forza bruta per questo problema non è abbastanza veloce. Un algoritmo di forza bruta per *PATH* procede esaminando tutti i potenziali cammini in G e determinando se uno di essi è un cammino diretto da s a t. Un potenziale cammino è una sequenza di nodi in G avente lunghezza al più m, dove m è il numero di nodi in G. (Se esiste un qualche cammino diretto da s a t, ne deve esistere necessariamente almeno uno avente lunghezza al massimo m, perché non è mai necessario ripetere un nodo.) Tuttavia il numero di tali percorsi potenziali è approssimativamente m^m, che è esponenziale nel numero di nodi in G. Pertanto questo algoritmo di forza bruta utilizza tempo esponenziale.

Per ottenere un algoritmo polinomiale per *PATH*, dobbiamo fare qualcosa che evita la forza bruta. Un modo è quello di utilizzare un metodo visita dei grafi quale la visita in ampiezza. In questo caso, contrassegniamo successivamente tutti nodi in G che sono raggiungibili da s mediante

cammini diretti di lunghezza 1, poi 2, poi 3, fino ad m. Possiamo facilmente limitare il tempo di esecuzione di questa strategia mediante un polinomio.

DIMOSTRAZIONE. Un algoritmo in tempo polinomiale M per *PATH* funziona come segue.

M = "Su input $\langle G, s, t\rangle$, dove G è un grafo orientato con nodi s e t:

1. Marca il nodo s.
2. Ripete il seguente passo fino a quando nessun nuovo nodo viene marcato:
3. Scansiona tutti gli archi di G. Se trova un arco (a, b) che va da un nodo a marcato ad un nodo b non marcato, marca il nodo b.
4. Se t è marcato, *accetta*. Altrimenti, *rifiuta*."

Ora analizziamo questo algoritmo per dimostrare che lavora in tempo polinomiale. Ovviamente, le fasi 1 e 4 sono eseguite una sola volta. La fase 3 è eseguita al più m volte perché ogni volta, tranne l'ultima, marca un nodo aggiuntivo in G. Quindi il numero totale di fasi utilizzate è al massimo $1 + 1 + m$, che da un polinomio nella dimensione di G.

Le fasi 1 e 4 di M sono facilmente implementate in tempo polinomiale su un qualsiasi modello deterministico ragionevole. La fase 3 richiede una scansione dell'input ed un test che verifichi se certi nodi sono marcati o meno, ed è anch'essa facilmente implementabile in tempo polinomiale. Quindi M è un algoritmo di tempo polinomiale per *PATH*.

Consideriamo ora un altro esempio di algoritmo avente tempo polinomiale. Diciamo che due numeri sono ***relativamente primi*** se 1 è il più grande numero intero che li divide entrambi. Per esempio, 10 e 21 sono relativamente primi, anche se nessuno dei due è esso stesso un numero primo, mentre 10 e 22 non sono relativamente primi perché entrambi sono divisibili per 2. Sia *RELPRIME* il problema di verificare se due numeri sono relativamente primi. Quindi

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ e } y \text{ sono relativamente primi}\}.$$

TEOREMA 7.15

RELPRIME $\in$ P.

IDEA. Un algoritmo che risolve questo problema effettua una ricerca tra tutti i divisori di entrambi i numeri e accetta se nessuno è maggiore di 1. Tuttavia, la grandezza di un numero rappresentato in binario, o in qualsiasi altra notazione in base k per $k \geq 2$, è esponenziale nella lunghezza della sua rappresentazione. Quindi questo algoritmo di forza bruta ricerca attraverso un numero esponenziale di potenziali divisori e ha un tempo di esecuzione esponenziale.

Risolviamo invece questo problema con un antico procedimento numerico, chiamato ***algoritmo di Euclide***, per il calcolo del massimo comun divisore. Il ***massimo comune divisore*** dei numeri naturali x e y, scritto $\gcd(x, y)$, è il più grande intero che divide sia x che y. Per esempio, $\gcd(18, 24) = 6$. Ovviamente, x e y sono relativamente primi sse il $\gcd(x, y) = 1$. Nella dimostrazione indicheremo l'algoritmo di Euclide come l'algoritmo E. Esso utilizza la funzione mod, per cui x mod y è il resto della divisione intera di x per y.

DIMOSTRAZIONE. L'algoritmo euclideo E è il seguente.

E = "Su input $\langle x, y\rangle$, dove x e y sono numeri naturali in binario:

1. Ripete finché $y = 0$:
2. Pone $x \leftarrow x$ mod y.
3. Scambia x e y.
4. Output x."

L'algoritmo R risolve *RELPRIME*, utilizzando E come sottoprocedura.

R = "Su input $\langle x, y\rangle$, dove x and y sono numeri naturali in binario:

1. Esegue E su $\langle x, y\rangle$.
2. Se il risultato è 1, *accetta*. Altrimenti, *rifiuta*."

Chiaramente, se E lavora correttamente in tempo polinomiale, così fa R e quindi abbiamo solo bisogno di analizzare E per valutarne il tempo e la correttezza. La correttezza di questo algoritmo è ben nota quindi non ne discuteremo ulteriormente in questo contesto. Per analizzare la complessità di tempo di E, per prima cosa mostriamo che ogni esecuzione della fase 2 (tranne eventualmente la prima), taglia il valore di x di almeno la metà. Dopo l'esecuzione della la fase 2, risulta $x < y$ a causa della natura della funzione mod. Dopo la fase 3, risulta $x > y$ perché i due valori sono stati scambiati. Così, quando la fase 2 viene successivamente eseguita, $x > y$. Se $x/2 \geq y$, allora x mod $y < y \leq x/2$ e x diminuisce di almeno la metà. Se $x/2 < y$, allora x mod $y = x - y < x/2$ e x diminuisce di almeno la metà.

I valori di x e y sono scambiati ogni volta che viene eseguita la fase 3, così ognuno dei valori originali di x e y è ridotto di almeno la metà ad ogni iterazione del ciclo. Quindi, il numero massimo di volte in cui le fasi 2 e 3 sono eseguite risulta pari al minore tra $2\log_2 x$ and $2\log_2 y$. Questi logaritmi sono proporzionali alle lunghezze delle rappresentazioni, il che dà il numero di fasi eseguite pari a $O(n)$. Ogni fase di E utilizza solo tempo polinomiale, quindi il tempo totale di esecuzione è polinomiale.

L'ultimo esempio di algoritmo polinomiale mostra che ogni linguaggio context-free è decidibile in tempo polinomiale.

TEOREMA 7.16

Ogni linguaggio context-free è un elemento di P.

IDEA. Nel Teorema 4.9, abbiamo dimostrato che ogni CFL è decidibile. Per fare ciò abbiamo dato un algoritmo che lo decide. Se questo algoritmo venisse eseguito in tempo polinomiale, questo teorema seguirebbe come corollario. Ricapitoliamo l'algoritmo e vediamo se funziona abbastanza velocemente. Sia L un CFL generato da una CFG G in forma normale di Chomsky. Dal Problema 2.38, qualsiasi derivazione di una stringa w ha $2n-1$ passi, dove n è la lunghezza di w perché G è in forma normale di Chomsky. Il decisore per L funziona provando tutte le possibili derivazioni con $2n-1$ passi quando il suo input è una stringa di lunghezza n. Se uno di questi è una derivazione di w, il decisore accetta, altrimenti, rifiuta.

Una rapida analisi di questo algoritmo dimostra che non viene eseguito in tempo polinomiale. Il numero di derivazioni con k passi può essere esponenziale in k, per cui questo algoritmo può richiedere tempo esponenziale.

Per ottenere un algoritmo di tempo polinomiale introduciamo una potente tecnica chiamata ***programmazione dinamica***. Questa tecnica utilizza l'accumulo di informazioni su sottoproblemi più piccoli per risolvere problemi più grandi. Memorizziamo la soluzione ad ogni sottoproblema in modo da doverlo risolvere solo una volta. Lo facciamo creando una tabella di tutti i sottoproblemi e inserendo le loro soluzioni sistematicamente appena le troviamo. Nel caso in oggetto, consideriamo i sottoproblemi consistenti nel determinare se ogni variabile in G genera ciascuna sottostringa di w. L'algoritmo inserisce la soluzione a questo sottoproblema in una tabella $n \times n$. Per $i \leq j$, la voce (i,j)-esima della tabella contiene la collezione di variabili che generano la sottostringa $w_i w_{i+1} \cdots w_j$. Per $i > j$, le voci della tabella non sono utilizzate. L'algoritmo riempie le voci della tabella per ogni sottostringa di w. In primo luogo si riempiono le voci corrispondento alle sottostringhe di lunghezza 1, poi quelle di lunghezza 2 e così via. Si utilizzano le voci per le lunghezze minori per determinare quelle per le lunghezze superiori.

Per esempio, si supponga che l'algoritmo abbia già stabilito quali varibili generano tutte le sottostringhe di lunghezza fino a k. Per determinare se una variabile A genera una particolare sottostringa di lunghezza $k+1$, l'algoritmo suddivide la sottostringa in due parti non vuote in tutti i k modi possibili. Per ogni suddivisione, l'algoritmo esamina ogni regola $A \to BC$ per determinare se B genera la prima parte e C genera la seconda parte,

utilizzando le voci della tabella precedentemente calcolate. Se entrambe B e C generano le rispettive parti, A genera la sottostringa e quindi viene aggiunta la voce corrispondente nella tabella. L'algoritmo inizia il processo con le stringhe di lunghezza 1 esaminando la tabella per le regole $A \to \mathtt{b}$.

DIMOSTRAZIONE. Il seguente algoritmo D implementa l'idea della dimostrazione. Sia G una CFG in forma normale di Chomsky che genera il CFL L. Si supponga che S sia la variabile iniziale. (Ricordiamo che la stringa vuota viene gestita in modo speciale nella grammatica in forma normale di Chomsky. L'algoritmo gestisce il caso particolare in cui $w = \varepsilon$ nella fase 1.) I commenti appaiono tra doppie parentesi quadre.

D = "Su input $w = w_1 \cdots w_n$:

1. Per $w = \varepsilon$, se $S \to \varepsilon$ è una regola, *accetta*; altrimenti, *rifiuta*. [[caso $w = \varepsilon$]]
2. Per $i = 1$ to n: [[esamina ogni sottostringa di lunghezza 1]]
3. Per ogni variabile A:
4. Testa se $A \to \mathtt{b}$ è una regola, dove $\mathtt{b} = w_i$.
5. Se si, poni A in $table(i,i)$.
6. Per $l = 2$ to n: [[l è la lunghezza della sottostringa]]
7. Per $i = 1$ to $n - l + 1$: [[i è la posizione iniziale della sottostringa]]
8. Sia $j = i + l - 1$. [[j è la posizione finale della sottostringa]]
9. Per $k = i$ to $j - 1$: [[k è la posizione di separazione]]
10. Per ogni regola $A \to BC$:
11. Se $table(i,k)$ contiene B e $table(k+1,j)$ contiene C, pone A in $table(i,j)$.
12. Se S è in $table(1,n)$, *accetta*; altrimenti, *rifiuta*."

Ora analizziamo D. Ogni fase è facilmente implementata in modo da essere eseguita in tempo polinomiale. Le fasi 4 e 5 sono eseguite al più nv volte, dove v è il numero di variabili in G ed è una costante fissa indipendente da n; quindi queste fasi sono eseguite $O(n)$ volte. La fase 6 è eseguite al più n volte. Ogni volta la fase 6 viene eseguita, la fase 7 viene eseguita al più n volte. Ogni volta che la fase 7 viene eseguita, le fasi 8 e 9 sono eseguite al più n volte. Ogni volta che la fase 9 viene eseguita, la fase l0 è eseguita r volte, dove r è il numero di regole di G ed è un'altra costante. Quindi la fase 11, il ciclo interno dell'algoritmo, viene eseguito $O(n^3)$ volte. La somma totale mostra che D esegue $O(n^3)$ fasi.

7.3 LA CLASSE NP

Come abbiamo osservato nella Sezione 7.2, per molti problemi possiamo evitare la ricerca mediante forza bruta ed ottenere una soluzione polinomiale. Tuttavia, i tentativi di evitare la forza bruta nel caso di altri problemi, tra cui molti utili ed interessanti, non hanno avuto successo e non sono noti algoritmi polinomiali per risolvere tali problemi. Come mai i tentativi di trovare algoritmi polinomiali per questi problemi non hanno avuto successo? Non conosciamo la risposta a questa importante domanda.

Forse questi problemi ammettono algoritmi in tempo polinomiale che si basano su principi non ancora noti. O forse alcuni di questi problemi semplicemente *non possono* essere risolti in tempo polinomiale. Possono essere intrinsecamente difficili.

Una scoperta notevole riguardo a questa domanda dimostra che le complessità di molti problemi sono legate tra di loro. Un algoritmo polinomiale per un certo problema può essere utilizzato per risolvere un'intera classe di problemi. Per capire questo fenomeno, cominciamo con un esempio.

Un ***cammino Hamiltoniano*** in un grafo orientato G è un cammino orientato che attraversa ogni nodo del grafo esattamente una volta. Consideriamo il problema di verificare se un grafo orientato contiene un cammino Hamiltoniano che collega due nodi specificati, come mostrato nella seguente figura. Sia

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ è un grafo orientato con un cammino Hamiltoniano da } s \text{ a } t\}.$$

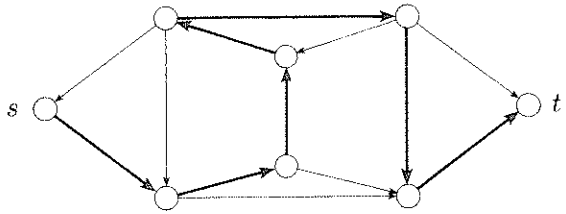


FIGURA 7.17
Un cammino Hamiltoniano attraversa ogni nodo esattamente una volta

Possiamo facilmente ottenere un algoritmo avente tempo esponenziale per il problema *HAMPATH* modificando l'algoritmo di forza bruta per *PATH* dato nel Teorema 7.14. Abbiamo bisogno solo di aggiungere un controllo per verificare che il cammino potenziale è Hamiltoniano. Nessuno sa se *HAMPATH* è risolvibile in tempo polinomiale.

Il problema *HAMPATH* ha una caratteristica chiamata ***verificabilità polinomiale*** che è importante per capire la sua complessità. Anche se non conosciamo una maniera veloce (ad esempio, in tempo polinomiale) per determinare se un grafo contiene un cammino Hamiltoniano, se un tale cammino è stato scoperto in qualche modo (magari utilizzando un algoritmo di tempo esponenziale), si potrebbe facilmente convincere qualcun altro della sua esistenza semplicemente esibendolo. In altre parole, *verificare* l'esistenza di un cammino Hamiltoniano può essere molto più facile che *determinare* la sua esistenza.

Un altro problema polinomialmente verificabile è l'essere composto. Ricordiamo che un numero naturale è ***composto*** se è il prodotto di due numeri interi maggiori di 1 (cioè, un numero composto è un numero che non è primo). Sia

$$COMPOSITES = \{x \mid x = pq, \text{ per gli interi } p, q > 1\}.$$

Possiamo facilmente verificare che un numero è composto – tutto ciò che è necessario è un divisore di tale numero. Recentemente, è stato scoperto un algoritmo di tempo polinomiale per testare se un numero è primo o composto, tuttavia è molto più complicato rispetto al metodo precedente per verificare tale proprietà.

Alcuni problemi potrebbero non essere polinomialmente verificabili. Per esempio, consideriamo $\overline{HAMPATH}$, il complemento del problema *HAMPATH*. Anche se riuscissimo a determinare (in qualche modo) che un grafo *non* ha un cammino Hamiltoniano, non conosciamo alcun modo per verificarne l'inesistenza senza utilizzare lo stesso algoritmo avente tempo esponenziale usato per la determinazione originaria. Una definizione formale è la seguente.

DEFINIZIONE **7.18**

Un ***verificatore*** per un linguaggio A è un algoritmo V, dove

$$A = \{w \mid V \text{ accetta } \langle w, c\rangle \text{ per qualche stringa } c\}.$$

Misuriamo il tempo di un verificatore solo in termini della lunghezza di w, quindi un ***verificatore in tempo polinomiale*** viene eseguito in tempo polinomiale nella lunghezza di w. Un linguaggio A è ***polinomialmente verificabile*** se ammette un verificatore in tempo polinomiale.

Un verificatore usa ulteriori informazioni, rappresentate dal simbolo c nella Definizione 7.18, per verificare che una stringa w è un elemento di A.

Questa informazione è chiamata ***certificato***, o ***prova***, di appartenenza ad A. Osservate che per i verificatori polinomiali, il certificato ha lunghezza polinomiale (nella lunghezza di w) perché questo è tutto ciò cui il verificatore può accedere a causa del suo limite sul tempo. Proviamo ad applicare questa definizione per i linguaggi *HAMPATH* e *COMPOSITES*. Per il problema *HAMPATH*, un certificato per una stringa $\langle G, s, t\rangle \in$ *HAMPATH* è semplicemente un cammino Hamiltoniano da s a t. Per il problema *COMPOSITES*, un certificato per il numero x è semplicemente uno dei suoi divisori. In entrambi i casi, il verificatore, avendo il certificato, può verificare in tempo polinomiale che l'input è nel linguaggio.

DEFINIZIONE **7.19**

NP è la classe dei linguaggi che ammettono un verificatore in tempo polinomiale.

La classe NP è importante perché contiene molti problemi di interesse pratico. Dalla discussione precedente, sia *HAMPATH* che *COMPOSITES* sono elementi di NP. Come abbiamo accennato, *COMPOSITES* è anche elemento di P, che è un sottoinsieme di NP; ma dimostrare questo risultato più forte è molto più difficile. Il termine NP proviene da ***tempo polinomiale non deterministico*** e deriva da una caratterizzazione alternativa che utilizza macchine di Turing non deterministiche di tempo polinomiale. I problemi in NP sono a volte chiamati problemi NP.

Quella che segue è una macchina di Turing non deterministica che decide il problema *HAMPATH* in tempo polinomiale non deterministico. Ricordiamo che nella Definizione 7.9, abbiamo definito il tempo di una macchina non deterministica come il tempo usato dalla computazione corrispondente alla ramificazione più lunga.

N_1 = "Su input $\langle G, s, t\rangle$, dove G è un grafo orientato con nodi s e t:

1. Scrive una lista di m numeri, $p_1, \ldots, p_m$, dove m è il numero di nodi in G. Ogni numero nella lista è scelto in modo non deterministico tra 1 e m.
2. Controlla se vi sono ripetizioni nella lista. Se ne trova una, *rifiuta*.
3. Controlla se $s = p_1$ e $t = p_m$. Se una delle due è falsa, *rifiuta*.
4. Per ogni i tra 1 e $m - 1$, controlla se (p_i, p_{i+1}) è un arco di G. Se non lo è, *rifiuta*. Altrimenti tutti i test sono stati superati, quindi *accetta*."

Per analizzare questo algoritmo e verificare che viene eseguito in tempo polinomiale non deterministico, esaminiamo ogni sua fase. Nella fase 1, la selezione non deterministica richiede chiaramente tempo polinomiale. Nelle fasi 2 e 3, ogni parte è un semplice controllo, quindi entrambe richiedono tempo polinomiale. Infine, anche la fase 4 richiede chiaramente tempo polinomiale. Quindi, questo algoritmo viene eseguito in tempo polinomiale non deterministico.

TEOREMA 7.20

Un linguaggio è in NP sse esso viene deciso in tempo polinomiale da una macchina di Turing non deterministica.

IDEA. Mostriamo come convertire un verificatore di tempo polinomiale in una NTM equivalente e viceversa. La NTM simula il verificatore per indovinare il certificato. Il verificatore simula la NTM utilizzando il ramo di computazione accettante come certificato.

DIMOSTRAZIONE. Per la parte diretta di questo teorema, assumiamo $A \in$ NP e mostriamo che A è deciso da una NTM N avente tempo polinomiale. Sia V il verificatore polinomiale per A la cui esistenza è assicurata dalla definizione di NP. Si supponga che V è una TM avente tempo di esecuzione n^k e costruiamo N come segue.

N = "Su input w di lunghezza n:

1. In maniera non deterministica seleziona una stringa c di lunghezza al più n^k.
2. Esegue V su input $\langle w, c\rangle$.
3. Se V accetta, *accetta*; altrimenti, *rifiuta*."

Per dimostrare la direzione inversa del teorema, supponiamo che A è deciso da una NTM N avente tempo polinomiale e costruiamo come segue un verificatore V avente tempo polinomiale.

V = "Su input $\langle w, c\rangle$, dove w e c sono stringhe:

1. Simula N su input w, trattando ogni simbolo di c come la descrizione di una scelta non deterministica da formulare ad ogni passo (come nella dimostrazione del Teorema 3.16).
2. Se questo ramo di computazione di N accetta, *accetta*; altrimenti, *rifiuta*."

Definiamo la classe di complessità di tempo non deterministico NTIME($t(n)$) in modo analogo alla classe di complessità di tempo deterministico TIME($t(n)$).

DEFINIZIONE **7.21**

NTIME*(t(n))* $= \{L \mid L$ è un linguaggio deciso da una macchina di Turing non deterministica di tempo $O(t(n))\}$.

COROLLARIO 7.22

NP $= \bigcup_k$ NTIME(n^k).

La classe NP è insensibile alla scelta di un modello computazionale ragionevole non deterministico in quanto tutti questi modelli sono polinomialmente equivalenti. Nel descrivere e analizzare algoritmi non deterministici aventi tempo polinomiale, seguiamo le convenzioni precedenti per gli algoritmi di tempo polinomiale deterministici. Ogni fase di un algoritmo non deterministico di tempo polinomiale deve avere una chiara implementazione in un tempo polinomiale non deterministico su un modello di calcolo non deterministico ragionevole. Analizziamo l'algoritmo per dimostrare che ogni ramo utilizza al massimo un numero polinomiale di fasi.

Esempi di problemi in NP

Una ***clique*** in un grafo non orientato è un sottografo, in cui ogni due nodi sono collegati da un arco. Una ***k-clique*** è una clique che contiene k nodi. La figura 7.23 illustra un grafo con una 5-clique.

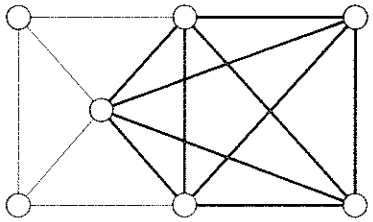


FIGURA 7.23
Un grafo con una 5-clique

Il problema della clique consiste nel determinare se un grafo contiene una clique di una dimensione specificata. Sia

$CLIQUE = \{\langle G, k\rangle \mid G$ è un grafo non orientato contenente una k-clique$\}$.

TEOREMA 7.24

CLIQUE è in NP.

IDEA. La clique è il certificato.

DIMOSTRAZIONE. Il seguente algoritmo è un verificatore V per *CLIQUE*.

V = "Su input $\langle\langle G, k\rangle, c\rangle$:

1. Controlla se c è un sottografo con k nodi in G.
2. Controlla se G contiene tutti gli archi tra i nodi in c.
3. Se entrambe le condizioni sono verificate, *accetta*; altrimenti, *rifiuta*."

DIMOSTRAZIONE ALTERNATIVA. Se preferite pensare a NP in termini di macchine di Turing di tempo polinomiale non deterministico, si può dimostrare questo teorema dandone una che decide *CLIQUE*. Osservate la somiglianza tra le due prove.

N = "Su input $\langle G, k\rangle$, dove G è un grafo:

1. Non deterministicamente seleziona un sottoinsieme c di k nodi di G.
2. Controlla se G contiene tutti gli archi tra i nodi in c
3. Se si, *accetta*; altrimenti, *rifiuta*."

Come altro esempio, consideriamo il problema di aritmetica sugli interi *SUBSET-SUM*. Ci viene dato un insieme di numeri $x_1, \ldots, x_k$ ed un valore obiettivo t. Vogliamo determinare se l'insieme contiene un sottoinsieme che somma a t. Quindi,

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ e per qualche } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ si ha } \Sigma y_i = t\}.$$

Per esempio, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *SUBSET-SUM* perché $4 + 21 = 25$. Notate che $\{x_1, \ldots, x_k\}$ e $\{y_1, \ldots, y_l\}$ sono considerati ***multinsiemi*** e quindi permettono la ripetizione di elementi.

TEOREMA 7.25

SUBSET-SUM è in NP.

IDEA. Il sottoinsieme è il certificato.

DIMOSTRAZIONE. L'algoritmo seguente è un verificatore V per *SUBSET-SUM*.

V = "Su input $\langle\langle S, t\rangle, c\rangle$:

1. Verifica se c è una collezione di numeri la cui somma è t.
2. Verifica se S contiene tutti i numeri in c.
3. Se le condizioni sono entrambe vere, *accetta*; altrimenti, *rifiuta*."

DIMOSTRAZIONE ALTERNATIVA. Possiamo anche dimostrare questo teorema esibendo una macchina di Turing di tempo polinomiale non deterministico per *SUBSET-SUM* come segue.

N = "Su input $\langle S, t\rangle$:

1. In modo non deterministico seleziona un sottoinsieme c dei numeri in S.
2. Verifica se c è una collezione di numeri la cui somma è t.
3. Se la condizione è vera, *accetta*; altrimenti, *rifiuta*."

Si osservi che i complementi di questi insiemi, $\overline{CLIQUE}$ e $\overline{SUBSET\text{-}SUM}$, non sono elementi di NP. Verificare che qualcosa *non* è presente sembra essere più difficile di verificare che è presente. Definiamo una classe di complessità separata, denominata **coNP**, che contiene i linguaggi che sono il complemento di un linguaggio in NP. Non sappiamo se coNP è diversa da NP.

La questione P = NP

Come abbiamo detto, NP è la classe dei linguaggi che sono risolubili in tempo polinomiale con una macchina di Turing non deterministica; o, equivalentemente, è la classe dei linguaggi per cui l'appartenenza può essere verificata in tempo polinomiale. P è la classe dei linguaggi per cui l'appartenenza può essere decisa in tempo polinomiale. Riassumiamo queste informazioni come segue, dove informalmente ci riferiamo ad un problema

risolvibile in tempo polinomiale come risolvibile "velocemente."

P = la classe dei linguaggi per cui l'appartenenza è *decidibile* velocemente.

NP = la classe dei linguaggi per cui l'appartenenza è *verificabile* velocemente.

Abbiamo presentato esempi di linguaggi, quali *HAMPATH* e *CLIQUE*, che sono elementi di NP, ma di cui non è nota l'appartenenza a P. Il potere della verificabilità polinomiale sembra essere molto maggiore di quello della decidibilità polinomiale. Ma, anche se difficile da immaginare, P e NP potrebbero essere uguali. Non siamo in grado di *dimostrare* l'esistenza di un solo linguaggio in NP che non è in P.

La domanda se P = NP è uno dei maggiori problemi irrisolti dell'informatica teorica e della matematica contemporanea. Se queste classi fossero uguali, qualsiasi problema polinomialmente verificabile sarebbe polinomialmente decidibile. La maggior parte dei ricercatori ritengono che le due classi non sono uguali, perché molte persone hanno investito, senza successo, enormi sforzi per trovare algoritmi aventi tempo polinomiale per problemi in NP. I ricercatori hanno anche tentato di dimostrare che le due classi sono diverse, ma ciò comporterebbe dimostrare che non esiste un algoritmo veloce che può sostituire la ricerca esaustiva. Ciò è fuori dalla portata scientifica attuale. La figura seguente illustra le due possibilità.

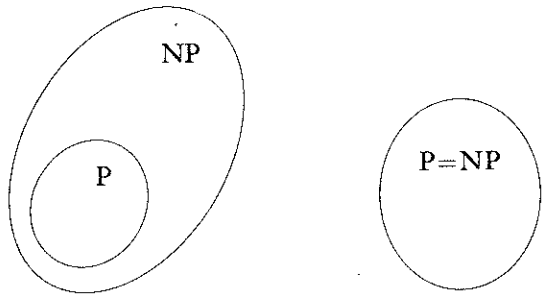


FIGURA 7.26
Una di queste due possibilità è corretta

Il miglior metodo deterministico attualmente noto per decidere se un linguaggio è in NP utilizza tempo esponenziale. In altre parole, possiamo dimostrare che

$$\text{NP} \subseteq EXPTIME = \bigcup_k \text{TIME}(2^{n^k}),$$

ma non sappiamo se NP è contenuta in una classe di complessità deterministica più piccola.

7.4

NP-COMPLETEZZA

Un progresso importante sulla questione P diverso da NP ci fu all'inizio degli anni '70 con il lavoro di Stephen Cook e Leonid Levin. Essi scoprirono vari problemi appartenenti a NP la cui complessità individuale è correlata a quella dell'intera classe. Se esistesse un algoritmo di tempo polinomiale per uno qualsiasi di essi, tutti i problemi in NP diventerebbero risolvibili in tempo polinomiale. Questi problemi vengono detti ***NP-completi***. Il fenomeno della NP-completezza è importante sia per ragioni teoriche che pratiche.

Sul versante teorico, un ricercatore che cerca di dimostrare che P è diverso da NP può focalizzare l'attenzione su un problema NP-completo. Se un qualsiasi problema appartenente a NP richiede tempo più che polinomiale, lo stesso vale per uno NP-completo. D'altra parte, un ricercatore che tenta di provare che P è uguale a NP deve soltanto trovare un algoritmo di tempo polinomiale per un problema NP-completo per raggiungere l'obiettivo.

Sul versante pratico, il fenomeno della NP-completezza può prevenire perdite di tempo nella ricerca di algoritmi di tempo polinomiale inesistenti per risolvere un problema particolare. Se anche non si dovesse disporre della matematica necessaria per dimostrare che il problema è irrisolvibile in tempo polinomiale, si ritiene che P non sia uguale a NP. Quindi, provare che il problema è NP-completo costituisce un'evidenza forte della sua non polinomialità.

Il primo problema NP-completo che presentiamo è il ***problema della soddisfacibilità***. Si ricordi che variabili che possono assumere i valori VERO e FALSO sono chiamate ***variabili booleane*** (si veda la Sezione 0.2). Solitamente rappresentiamo VERO con 1 e FALSO con 0. Le ***operazioni booleane*** AND, OR, e NOT, rappresentate con i simboli $\wedge$, $\vee$, e $\neg$, rispettivamente, sono descritte nella lista seguente. Utilizziamo la sopralineatura come un'abbreviazione per il simbolo $\neg$, quindi $\overline{x}$ significa $\neg\, x$.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

Una ***formula booleana*** è un'espressione che coinvolge variabili e operazioni booleane. Per esempio,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

è una formula booleana. Una formula booleana è ***soddisfacibile*** se qualche assegnamento di 0 e di 1 alle variabili fa sì che la formula valga 1. La

formula precedente è soddisfacibile perché l'assegnamento $x = 0$, $y = 1$ e $z = 0$ fa sì che ϕ valga 1. Diciamo che l'assegnamento *soddisfa* ϕ. Il ***problema della soddisfacibilità*** consiste nel verificare se una formula booleana è soddisfacibile. Sia

$$SAT = \{\langle\phi\rangle \mid \phi \text{ è una formula booleana soddisfacibile}\}.$$

Il teorema che ora enunciamo lega la complessità del problema SAT alle complessità di tutti i problemi appartenenti a NP.

TEOREMA 7.27

$SAT \in$ P se e solo se P = NP.

Nel seguito svilupperemo il metodo che risulta centrale per la dimostrazione del teorema.

Riducibilità in tempo polinomiale

Nel Capitolo 5 abbiamo definito il concetto di riduzione di un problema ad un altro. Quando il problema A si riduce al problema B, una soluzione per B può essere usata per risolvere A. Ora definiamo una versione della riducibilità che tiene conto dell'efficienza della computazione. Quando il problema A è riducibile *efficientemente* al problema B, una soluzione efficiente per B può essere usata per risolvere A efficientemente.

DEFINIZIONE **7.28**

Una funzione $f\colon \Sigma^* \longrightarrow \Sigma^*$ è una ***funzione calcolabile in tempo polinomiale*** se esiste una macchina di Turing di tempo polinomiale M che si arresta con $f(w)$ soltanto sul suo nastro, quando ha iniziato con un qualsiasi input w.

DEFINIZIONE **7.29**

Il linguaggio A è ***riducibile mediante funzione in tempo polinomiale***,[1] o semplicemente ***riducibile in tempo polinomiale***, al linguaggio B, denotato con $A \leq_{\mathrm{P}} B$, se esiste una funzione calcolabile in tempo polinomiale $f\colon \Sigma^* \longrightarrow \Sigma^*$ dove, per ogni w,

$$w \in A \iff f(w) \in B.$$

La funzione f è chiamata ***riduzione di tempo polinomiale*** di A a B.

La riducibilità in tempo polinomiale è l'analogo efficiente della riducibilità mediante funzione come definita in Sezione 5.3. Sono disponibili altre forme di riducibilità efficiente, ma la riducibilità in tempo polinomiale è una forma semplice che è adeguata per i nostri scopi e quindi non discuteremo le altre in questa sede. La Figura 7.30 illustra la riducibilità in tempo polinomiale.

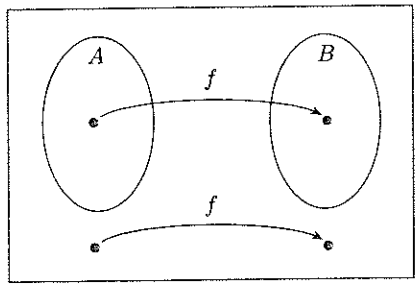


FIGURA 7.30
Funzione di tempo polinomiale f che riduce A a B

Come con un'ordinaria riduzione mediante funzione, una riduzione in tempo polinomiale di A a B fornisce un modo per convertire la verifica dell'appartenenza ad A nella verifica dell'appartenenza a B - ma ora la conversione viene realizzata efficientemente. Per verificare se $w \in A$, utilizziamo la riduzione f per associare w ad $f(w)$ e verificare se $f(w) \in B$.

Se un linguaggio è riducibile in tempo polinomiale ad un linguaggio per cui già si sa che possiede una soluzione di tempo polinomiale, si ottiene una soluzione di tempo polinomiale per il linguaggio originale, come prova il teorema seguente.

TEOREMA 7.31

Se $A \leq_P B$ e $B \in P$, allora $A \in P$.

DIMOSTRAZIONE. Sia M l'algoritmo di tempo polinomiale che decide B ed f la riduzione di tempo polinomiale di A a B. Descriviamo un algoritmo di tempo polinomiale N che decide A come segue.

N = "Su input w:

1. Calcola $f(w)$.

[1] È chiamata ***riducibilità molti-a-uno in tempo polinomiale*** in altri libri di testo.

2. Esegui M con input $f(w)$ e dai in output qualsiasi cosa M dà in output."

Risulta $w \in A$ ogni volta che $f(w) \in B$ perché f è una riduzione di A a B. Pertanto, M accetta $f(w)$ ogni volta che $w \in A$. Inoltre, N computa in tempo polinomiale perché ciascuna delle sue due fasi viene eseguita in tempo polinomiale. Si noti che la fase 2 viene eseguita in tempo polinomiale perché la composizione di due polinomi è un polinomio.

Prima di mostrare una riduzione di tempo polinomiale, introduciamo *3SAT*, un caso speciale di problema di soddisfacibilità per cui tutte le formule sono in una forma speciale. Un ***letterale*** è una variabile booleana o una variabile booleana negata, come x o $\overline{x}$. Una ***clausola*** consiste in diversi letterali connessi tramite operatori $\vee$, come $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. Una formula booleana è in ***forma normale congiuntiva***, ed è detta una ***formula cnf***, se comprende diverse clausole connesse tramite operatori $\wedge$, come

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

Essa è una ***formula 3cnf*** se tutte le clausole hanno tre letterali, come in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Sia $3SAT = \{\langle\phi\rangle \mid \phi$ è una formula 3cnf soddisfacibile$\}$. Se un assegnamento soddisfa una formula cnf, ciascuna clausola deve contenere almeno un letterale che vale 1.

Il teorema seguente presenta una riduzione di tempo polinomiale dal problema *3SAT* al problema *CLIQUE*.

TEOREMA 7.32

3SAT è riducibile in tempo polinomiale a *CLIQUE*.

IDEA. La riduzione di tempo polinomiale f che mostriamo di *3SAT* a *CLIQUE* converte formule in grafi. Nei grafi costruiti, clique di una dimensione specificata corrispondono ad assegnamenti soddisfacenti per la formula. Le strutture all'interno del grafo sono progettate per simulare il comportamento delle variabili e delle clausole.

DIMOSTRAZIONE. Sia ϕ una formula con k clausole come

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

La riduzione f genera la stringa $\langle G, k\rangle$, dove G è un grafo non orientato definito come segue.

I nodi in G sono organizzati in k gruppi di tre nodi ciascuno, detti ***triple***, $t_1, \ldots, t_k$. Ciascuna tripla corrisponde ad una delle clausole in ϕ, e ciascun nodo in una tripla corrisponde ad un letterale nella clausola associata. Si etichetti ciascun nodo di G con il suo letterale corrispondente in ϕ.

Gli archi di G connettono tutti meno due tipi di coppie di nodi in G. Non ci sono archi presenti tra nodi nella stessa tripla, e non ci sono archi presenti tra nodi con etichette complementari, come x_2 ed $\overline{x_2}$. La Figura 7.33 illustra questa costruzione quando $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

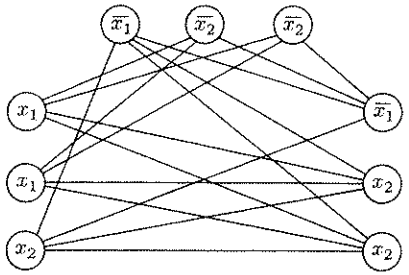


FIGURA 7.33
Il grafo che la riduzione produce per $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Facciamo vedere ora perché questa costruzione funziona. Dimostriamo che ϕ è soddisfacibile se e solo se G ha una clique di dimensione k.

Si supponga che ϕ abbia un assegnamento che la soddisfa. In questo assegnamento soddisfacente, almeno un letterale è vero in ogni clausola. In ciascuna tripla di G, si selezioni un nodo corrispondente al letterale vero nell'assegnamento soddisfacente. Se più di un letterale è vero in una particolare clausola, si scelga uno dei letterali veri arbitrariamente. I nodi appena selezionati formano una clique di dimensione k. Il numero di nodi selezionati è k perché ne è stato scelto uno per ognuna delle k triple. Ciascuna coppia dei nodi selezionati è collegata da un arco perché nessuna coppia rientra in una delle eccezioni descritte in precedenza. Non possono essere della stessa tripla perché ne è stato scelto soltanto uno per tripla. Non possono avere etichette complementari perché i letterali associati erano entrambi veri nell'assegnamento soddisfacente. Pertanto, G contiene una clique di dimensione k.

Si supponga che G abbia una clique di dimensione k. Nessuna coppia di nodi di una clique occorre nella stessa tripla perché i nodi nella stessa tripla non sono connessi da archi. Pertanto, ciascuna delle k triple contiene esattamente uno dei k nodi della clique. Si assegnino valori di verità alle variabili di ϕ, in modo tale che ciascun letterale che etichetta un nodo della

clique risulti vero. Questa operazione è sempre possibile perché due nodi etichettati in modo complementare non sono connessi da un arco e, quindi, non possono essere entrambi nella clique. L'assegnamento così definito per le variabili soddisfa ϕ perché ciascuna tripla contiene un nodo della clique e, quindi, ciascuna clausola contiene un letterale a cui è stato assegnato VERO. Pertanto, ϕ è soddisfacibile.

I Teoremi 7.31 e 7.32 ci dicono che se *CLIQUE* è risolvibile in tempo polinomiale lo è anche *3SAT*. Di primo acchito questa connessione tra i due problemi sembra veramente notevole perché superficialmente appaiono piuttosto differenti. Ma la riducibilità in tempo polinomiale ci permette di collegare le rispettive complessità. A questo punto volgiamo l'attenzione ad una definizione che ci permette in modo simile di collegare le complessità di un'intera classe di problemi.

Definizione di NP-completezza

DEFINIZIONE **7.34**

Un linguaggio B è ***NP-completo*** se soddisfa due condizioni:

1. B appartiene a NP, e
2. ogni A appartenente a NP è riducibile in tempo polinomiale a B.

TEOREMA 7.35

Se B è NP-completo e $B \in$ P, allora P = NP.

DIMOSTRAZIONE. Il teorema discende direttamente dalla definizione di riducibilità in tempo polinomiale.

TEOREMA 7.36

Se B è NP-completo e $B \leq_P C$ per C appartenente a NP, allora C è NP-completo.

DIMOSTRAZIONE. Già sappiamo che C appartiene a NP, quindi dobbiamo far vedere che ogni A appartenente a NP è riducibile in tempo polinomiale a C. poiché

B è NP-completo, ogni linguaggio appartenente a NP è riducibile in tempo polinomiale a B, e B a sua volta è riducibile in tempo polinomiale a C. Le riduzioni di tempo polinomiale possono essere composte; cioè, se A è riducibile in tempo polinomiale a B e B è riducibile in tempo polinomiale a C, allora A è riducibile in tempo polinomiale a C. Pertanto, ogni linguaggio appartenente a NP è riducibile in tempo polinomiale a C.

Il Teorema di Cook e Levin

Una volta che disponiamo di un problema NP-completo, possiamo ottenerne altri attraverso riduzioni polinomiali da esso. Tuttavia, determinare il primo problema NP-completo è più difficile. Lo facciamo ora, provando che SAT è NP-completo.

TEOREMA 7.37

SAT è NP-completo.[2]

Questo teorema implica il Teorema 7.27.

IDEA. Mostrare che SAT appartiene a NP è facile, e lo faremo velocemente. La parte difficile della dimostrazione è far vedere che qualsiasi linguaggio appartenente a NP è riducibile in tempo polinomiale a SAT.

Per far ciò, costruiremo una riduzione di tempo polinomiale per ciascun linguaggio A appartenente a NP a SAT. La riduzione per A prende una stringa w e produce una formula booleana ϕ che simula la macchina NP per A su input w. Se la macchina accetta, ϕ ha un assegnamento che la soddisfa che corrisponde ad una computazione accettante. Se la macchina non accetta, nessun assegnamento soddisfa ϕ. Pertanto, w appartiene ad A se e solo se ϕ è soddisfacibile.

In realtà, costruire la riduzione per farla funzionare in questo modo è un compito concettualmente semplice, sebbene debbano essere gestiti diversi dettagli. Una formula booleana può contenere le operazioni booleane AND, OR, e NOT, e queste operazioni costituiscono la base per la circuiteria usata nei calcolatori elettronici. Quindi, il fatto che possiamo progettare una formula booleana per simulare una macchina di Turing non è sorprendente. I dettagli sono nell'implementazione di questa idea.

DIMOSTRAZIONE. Prima di tutto, mostriamo che SAT appartiene a NP. Una macchina non deterministica di tempo polinomiale può ipotizzare un assegnamento per una data formula ϕ ed accettare se l'assegnamento soddisfa ϕ.

[2]Una dimostrazione alternativa del teorema viene data in Sezione 9.3.

Successivamente, prendiamo un qualsiasi linguaggio A appartenente a NP e facciamo vedere che A è riducibile in tempo polinomiale a SAT. Sia N una macchina di Turing non deterministica che decide A in tempo n^k per qualche costante k. (Per comodità in effetti assumiamo che N computi in tempo $n^k - 3$; ma soltato i lettori interessati ai dettagli dovrebbero preoccuparsi di questo punto minore.) La nozione che segue aiuta a descrivere la riduzione.

Un ***tableau*** per N su w è una tabella $n^k \times n^k$ le cui righe sono le configurazioni di una diramazione della computazione di N su input w, come mostrato nella figura seguente.

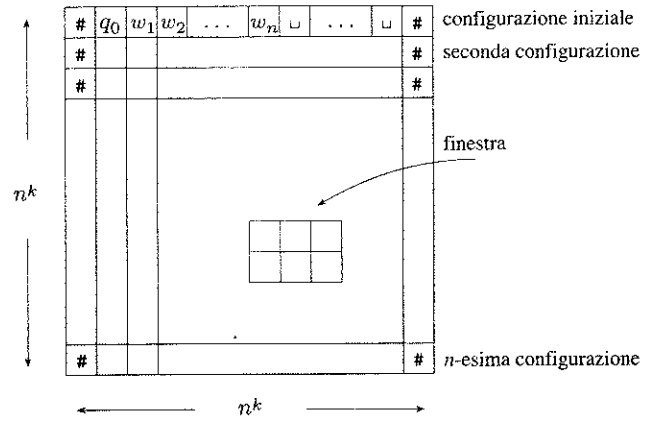


FIGURA 7.38
Un tableau è una tabella $n^k \times n^k$ di configurazioni

Per convenienza nel seguito assumiamo che ciascuna configurazione inizi e finisca con un simbolo #. Pertanto, la prima e l'ultima colonna di un tableau sono tutti #. La prima riga di un tableau è la configurazione iniziale di N su w, e ciascuna riga segue dalla precedente in accordo alla funzione di transizione di N. Un tableau è ***accettante*** se qualche riga del tableau è una configurazione accettante.

Ogni tableau accettante per N su w corrisponde ad una diramazione della computazione accettante di N su w. Quindi, il problema di stabilire se N accetta w è equivalente al problema di stabilire se esiste un tableau accettante per N su w.

Passiamo ora alla descrizione della riduzione di tempo polinomiale f di A a SAT. Su input w, la riduzione produce una formula ϕ. Iniziamo descrivendo le variabili di ϕ. Siano Q e Γ l'insieme degli stati e l'alfabeto del nastro di N, rispettivamente. Sia $C = Q \cup \Gamma \cup \{\#\}$. Per ciascun i e j tra 1 ed n^k e per ciascun s in C, introduciamo una variabile, $x_{i,j,s}$.

Ciascuna delle $(n^k)^2$ entrate di un tableau è chiamata ***cella***. La cella in riga i e colonna j viene denotata con $cell[i,j]$ e contiene un simbolo di C. Rappresentiamo i contenuti delle celle con le variabili di ϕ. Se $x_{i,j,s}$ assume il valore 1, significa che $cell[i,j]$ contiene s.

Progettiamo a questo punto ϕ in modo tale che un assegnamento che soddisfa le variabili di ϕ corrisponda ad un tableau accettante per N su w. La formula ϕ è l'AND di quattro parti: $\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$. Descriviamo ciascuna parte, una per volta.

Come anticipato precedentemente, porre a 1 la variabile $x_{i,j,s}$ corrisponde a posizionare il simbolo s in $cell[i,j]$. La prima cosa che dobbiamo garantire al fine di ottenere una corrispondenza tra un assegnamento ed un tableau è che l'assegnamento ponga ad 1 esattamente una variabile per ciascuna cella. La formula ϕ_{cell} garantisce questo requisito esprimendolo in termini di operazioni booleane:

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

I simboli $\bigwedge$ e $\bigvee$ stanno per AND ed OR iterati. Per esempio, l'espressione nella formula precedente

$$\bigvee_{s \in C} x_{i,j,s}$$

è un'abbreviazione per

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \cdots \vee x_{i,j,s_l}$$

dove $C = \{s_1, s_2, \ldots, s_l\}$. Quindi, ϕ_{cell} è in realtà un'espressione grande che contiene un frammento per ciascuna cella nel tableau, poiché i e j variano da 1 a n^k. La prima parte del frammento stabilisce che almeno una variabile assume valore 1 nella cella corrispondente. La seconda parte di ciascun frammento stabilisce che non più di una variabile assume valore 1 (letteralmente, stabilisce che, in ogni coppia di variabili, almeno una assume valore 0) nella cella corrispondente. Questi frammenti sono collegati attraverso operazioni $\wedge$.

La prima parte di ϕ_{cell} all'interno delle parentesi garantisce che almeno una variabile che è associata con ciascuna cella vale 1, laddove la seconda parte garantisce che non più di una variabile vale 1 per ciascuna cella. Qualsiasi assegnamento alle variabili che soddisfa ϕ (e di conseguenza ϕ_{cell}) deve avere esattamente una variabile ad 1 per ogni cella. Pertanto, qualsiasi assegnamento che soddisfa ϕ specifica un simbolo in ciascuna cella della tabella. Le parti ϕ_{start}, ϕ_{move}, e ϕ_{accept} garantiscono che questi simboli corrispondano realmente ad un tableau accettante come segue.

La formula ϕ_{start} assicura che la prima riga della tabella sia la configurazione iniziale di N su w, richiedendo esplicitamente che le variabili

corrispondenti siano ad 1:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}.\end{aligned}$$

La formula ϕ_{accept} garantisce che una configurazione accettante sia presente nel tableau. Essa assicura che q_{accept}, il simbolo per lo stato di accettazione, compaia in una delle celle del tableau, richiedendo che una delle variabili corrispondenti sia a 1:

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}}.$$

Infine, la formula ϕ_{move} garantisce che ciascuna riga del tableau corrisponda ad una configurazione che segue legittimamente dalla configurazione della riga precedente, in accordo alle regole di N. Lo fa assicurando che ogni finestra di celle 2×3 sia lecita. Diciamo che una finestra 2×3 è ***lecita*** se non viola le azioni specificate dalla funzione di transizione di N. In altre parole, una finestra è lecita se può esser presente quando una configurazione segue correttamente da un'altra.[3]

Per esempio, siano a, b, e c elementi dell'alfabeto di nastro, e q_1 e q_2 stati di N. Si assuma che quando nello stato q_1 con la testina che legge una a, N scrive una b, resta nello stato q_1, e muove a destra; e che quando nello stato q_1 con la testina che legge una b, N non deterministicamente

1. scrive una c, entra in q_2, e muove a sinistra, oppure
2. scrive una a. entra in q_2, e muove a destra.

In termini formali, $\delta(q_1, \mathtt{a}) = \{(q_1, \mathtt{b}, \mathrm{R})\}$ e $\delta(q_1, \mathtt{b}) = \{(q_2, \mathtt{c}, \mathrm{L}), (q_2, \mathtt{a}, \mathrm{R})\}$. Esempi di finestre lecite per questa macchina sono mostrati in Figura 7.39.

[3] Potremmo dare una qui definizione precisa di ***finestra lecita***, in termini della funzione di transizione. Ma far ciò è alquanto tedioso e ci distrarrebbe dal cuore dell'argomento della dimostrazione. Chiunque desiderasse maggior precisione dovrebbe far riferimento all'analisi relativa nella dimostrazione del Teorema 5.15, l'indecidibilità del Problema della Corrispondenza di Post.

(a)
a	q_1	b
q_2	a	c

(b)
a	q_1	b
a	a	q_2

(c)
a	a	q_1
a	a	b

(d)
#	b	a
#	b	a

(e)
a	b	a
a	b	q_2

(f)
b	b	b
c	b	b

FIGURA 7.39
Esempi di finestre lecite

In Figura 7.39, le finestre (a) e (b) sono lecite perché la funzione di transizione permette ad N di muovere nella direzione indicata. La finestra (c) è lecita perché, con q_1 presente sul lato destro della riga di sopra, non sappiamo su quale simbolo la testina sia. Detto simbolo potrebbe essere una a, e q_1 potrebbe cambiarlo in una b e muovere a destra. Una tale possibilità genererebbe questa finestra, quindi essa non viola le regole di N. La finestra (d) è ovviamente lecita perché la riga di sopra e quella di sotto sono identiche, situazione che si verifica se la testina non è adiacente alla posizione della finestra. Si noti che # in una finestra lecita può comparire a sinistra o a destra sia nella riga superiore che in quella inferiore. La finestra (e) è lecita perché lo stato q_1 potrebbe essere immediatamente a destra della riga superiore e la macchina, leggendo una b, si sarebbe mossa a sinistra nello stato q_2 che ora compare all'estremità destra della riga inferiore. Infine, la finestra (f) è lecita perché lo stato q_1 potrebbe essere stato immediatamente a sinistra della riga superiore, e la macchina potrebbe aver cambiato la b in una c ed effettuato una mossa a sinistra.

Le finestre mostrate nella figura seguente non sono lecite per la macchina N.

(a)
a	b	a
a	a	a

(b)
a	q_1	b
q_2	a	a

(c)
b	q_1	b
q_2	b	q_2

FIGURA 7.40
Esempi di finestre non lecite

Nella finestra (a), il simbolo centrale nella riga superiore non può cambiare perché non c'è uno stato ad esso adiacente. La finestra (b) non è lecita perché la funzione di transizione specifica che la b viene cambiata in una c ma non in una a. La finestra (c) non è lecita perché due stati compaiono nella riga inferiore.

FATTO 7.41

Se la riga superiore del tableau è la configurazione iniziale ed ogni finestra nel tableau è lecita, ciascuna riga del tableau è una configurazione che segue legittimamente dalla precedente.

Proviamo l'asserto considerando ogni coppia di configurazioni adiacenti nel tableau, indicate come la configurazione superiore e la configurazione inferiore. Nella configurazione superiore, ogni cella che contiene un simbolo di nastro e non è adiacente ad un simbolo di stato rappresenta la cella centrale superiore in una finestra in cui la riga superiore non contiene stati. Pertanto, questo stesso simbolo deve comparire al centro in basso nella finestra. Quindi, esso è presente nella stessa posizione nella configurazione inferiore.

La finestra contenente il simbolo di stato nella cella centrale superiore garantisce che le tre posizioni corrispondenti vengano aggiornate consistentemente tramite la funzione di transizione. Pertanto, se la configurazione superiore è una configurazione lecita, altrettanto lo è la configurazione inferiore, e quella inferiore discende da quella superiore in accordo alle regole di N. Si noti che questa dimostrazione, seppur immediata, dipende in maniera cruciale dalla nostra scelta di utilizzare una finestra di dimensione 2×3, come mostra il Problema 7.26.

Torniamo ora alla costruzione di ϕ_{move}. Essa garantisce che tutte le finestre nel tableau sono lecite. Ciascuna finestra contiene sei celle, che possono essere impostate in un numero fissato di modi per produrre una finestra lecita. La formula ϕ_{move} stabilisce che le impostazioni delle sei celle devono essere uno di questi modi, ossia

$$\phi_{\text{move}} = \bigwedge_{1 \le i < n^k,\ 1 < j < n^k} (\text{la finestra } (i,j) \text{ è lecita}).$$

La finestra (i,j) ha $cell[i,j]$ in posizione centrale in alto. Sostituiamo il testo "la finestra (i,j) è lecita" in questa formula con la formula seguente. Denotiamo il contenuto delle sei celle di una finestra con $a_1, \ldots, a_6$.

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{è una finestra lecita}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

Nel seguito analizziamo la complessità della riduzione per mostrare che essa opera in tempo polinomiale. Per far ciò, esaminiamo la dimensione di ϕ. Prima di tutto, stimiamo il numero di variabili che ha. Si ricordi che il tableau è una tabella $n^k \times n^k$, quindi contiene n^{2k} celle. Ciascuna cella ha l variabili associate ad essa, dove l è il numero di simboli in C. Poiché l dipende soltanto dalla TM N e non dalla lunghezza dell'input n, il numero totale di variabili è $O(n^{2k})$.

Stimiamo la dimensione di ciascuna delle parti di ϕ. La formula ϕ_{cell} contiene un frammento di dimensione fissata della formula per ciascuna

cella del tableau, quindi la sua dimensione è $O(n^{2k})$. La formula ϕ_{start} ha un frammento per ciascuna cella nella riga superiore, quindi la sua dimensione è $O(n^k)$. Le formule ϕ_{move} e ϕ_{accept} contengono ciascuna un frammento di dimensione fissata della formula per ciascuna cella del tableau, quindi la loro dimensione è $O(n^{2k})$. Pertanto, la dimensione complessiva di ϕ è $O(n^{2k})$. Tale limite è sufficiente per i nostri scopi perché dimostra che la dimensione di ϕ è polinomiale in n. Se fosse stata più che polinomiale, la riduzione non avrebbe avuto alcuna possibilità di generarla in tempo polinomiale. (In realtà, le nostre stime sono più basse di un fattore $O(\log n)$, poiché ciascuna variabile ha indici che possono arrivare fino a n^k e, quindi, possono richiedere $O(\log n)$ simboli da scrivere nella formula, ma questo fattore addizionale non cambia la polinomialità del risultato.)

Per rendersi conto che la formula può essere generata in tempo polinomiale, si noti la sua natura altamente ripetitiva. Ciascun componente della formula è composto da molti frammenti quasi identici, che differiscono soltanto negli indici in modo molto semplice. Pertanto, possiamo costruire facilmente una riduzione che produce ϕ in tempo polinomiale dall'input w.

Quindi, abbiamo concluso la dimostrazione del teorema di Cook e Levin, mostrando che *SAT* è NP-completo. Mostrare la NP-completezza di altri linguaggi generalmente non richiede una dimostrazione così lunga. Al contrario, la NP-completezza può essere provata con una riduzione di tempo polinomiale da un linguaggio che è già noto essere NP-completo. Possiamo usare *SAT* per questo scopo; ma usare *3SAT*, il caso speciale di *SAT* che abbiamo definito a pagina 323, è solitamente più facile. Si ricordi che le formule appartenenti a *3SAT* sono in forma normale congiuntiva (cnf) con tre letterali per clausola. Prima di tutto, dobbiamo dimostrare che *3SAT* stesso è NP-completo. Proviamo questo asserto come corollario del Teorema 7.37.

COROLLARIO 7.42

3SAT è NP-completo.

DIMOSTRAZIONE. Ovviamente *3SAT* appartiene a NP, quindi dobbiamo provare solamente che tutti i linguaggi appartenenti a NP si riducono a *3SAT* in tempo polinomiale. Un modo per farlo è facendo vedere che *SAT* si riduce in tempo polinomiale a *3SAT*. Invece, preferiamo farlo modificando la dimostrazione del Teorema 7.37, in modo tale che produca direttamente una formula in forma normale congiuntiva con tre letterali per clausola.

Il Teorema 7.37 produce una formula che è già quasi in forma congiuntiva normale. La formula ϕ_{cell} è un grosso AND di sottoformule, ciascuna delle quali contiene un grosso OR ed un grosso AND di diversi OR. Quindi, ϕ_{cell} è un AND di clausole e, di conseguenza, è già in forma cnf. La formula ϕ_{start} è un grosso AND di variabili. Prendendo ciascuna di queste variabili come clausole di dimensione 1, notiamo che ϕ_{start} è in forma cnf. La formula ϕ_{accept} è un grosso OR di variabili, ed è perciò una singola clausola. La formula ϕ_{move} è l'unica che non è già in forma cnf, ma possiamo facilmente convertirla in una formula che è in forma cnf come segue.

Si ricordi che ϕ_{move} è un grosso AND di sottoformule, ciascuna delle quali è un OR di diversi AND che descrive tutte le possibili finestre lecite. La legge distributiva, come descritto nel Capitolo 0, stabilisce che possiamo sostituire un OR di vari AND con un equivalente AND di vari OR. Fare ciò può incrementare significativamente la dimensione di ciascuna sottoformula, ma tale operazione può incrementare la dimensione complessiva di ϕ_{move} solamente di un fattore costante, poiché la dimensione di ciascuna sottoformula dipende solo da N. Il risultato è una formula che è in forma normale congiuntiva.

Ora che abbiamo scritto la formula in forma cnf, la convertiamo in una con tre letterali per clausola. In ciascuna clausola che correntemente ha uno o due letterali, duplichiamo uno dei letterali, fino a quando il numero totale diventa tre. Ciascuna clausola che ha più di tre letterali la dividiamo in più clausole e aggiungiamo ulteriori variabili per preservare la soddisfacibilità o non soddisfacibilità della clausola originale.

Per esempio, sostituiamo la clausola $(a_1 \vee a_2 \vee a_3 \vee a_4)$, dove ciascun a_i è un letterale, con l'espressione di due clausole $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$, in cui z è una variabile nuova. Se qualche impostazione degli a_i soddisfa la clausola originale, possiamo trovare una impostazione di z in modo tale che le due nuove clausole siano soddisfatte e viceversa. In generale, se la clausola contiene l letterali,

$$(a_1 \vee a_2 \vee \cdots \vee a_l),$$

possiamo sostituirla con le $l-2$ clausole

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$

È possibile verificare facilmente che la nuova formula è soddisfacibile se e solo se la formula originale lo era, quindi la dimostrazione è completa.

7.5 ULTERIORI PROBLEMI NP-COMPLETI

Il fenomeno della NP-completezza è molto diffuso. Problemi NP-completi emergono in svariati campi. Per ragioni che non sono ben comprese, molti

problemi in NP che si presentano in modo naturale o appartengono a P o sono NP-completi. Se si cerca un algoritmo di tempo polinomiale per un nuovo problema in NP, spendere parte dei proprio sforzi per cercare di provare che risulta NP-completo è ragionevole, perché può prevenire lavoro nella ricerca di un algoritmo di tempo polinomiale che non esiste.

In questa sezione presentiamo teoremi aggiuntivi che mostrano che diversi linguaggi sono NP-completi. I suddetti teoremi forniscono esempi delle tecniche che vengono utilizzate nelle dimostrazioni di questo tipo. La nostra strategia generale è esibire una riduzione di tempo polinomiale da *3SAT* al linguaggio in questione, sebbene qualche volta riduciamo da altri linguaggi NP-completi nei casi in cui risulta più conveniente.

Quando costruiamo una riduzione di tempo polinomiale da *3SAT* ad un linguaggio, cerchiamo strutture in quel linguaggio che possono simulare le variabili e le clausole nelle formule booleane. Tali strutture sono a volte chiamate ***gadget***. Per esempio, nella riduzione da *3SAT* a *CLIQUE* presentata nel Teorema 7.32, nodi individuali simulano variabili e triple di nodi simulano clausole. Un nodo individuale può essere un elemento della clique come può non esserlo, in corrispondenza ad una variabile che può essere vera in un assegnamento che la soddisfa come può non esserlo. Ciascuna clausola deve contenere un letterale a cui viene assegnato il valore VERO. Corrispondentemente, ciascuna tripla deve contenere un nodo nella clique (al fine di raggiungere la dimensione voluta). Il corollario seguente che discende dal Teorema 7.32 stabilisce che *CLIQUE* è NP-completo.

COROLLARIO 7.43

CLIQUE è NP-completo.

Il problema del vertex cover

Se G è un grafo non orientato, un ***vertex cover*** (copertura mediante vertici) di G è un sottoinsieme dei nodi in cui ogni arco di G tocca uno di quei nodi. Il problema del vertex cover chiede di stabilire se un grafo contiene un vertex cover di una dimensione specificata:

$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ è un grafo non orientato che ha un vertex cover di } k \text{ nodi}\}.$$

TEOREMA 7.44

VERTEX-COVER è NP-completo.

IDEA. Per mostrare che *VERTEX-COVER* è NP-completo, dobbiamo far vedere che appartiene a NP e che tutti i problemi appartenenti a NP sono riducibili in tempo polinomiale ad esso. La prima parte è facile; un certificato è semplicemente un vertex cover di dimensione k. Per provare la seconda parte, mostriamo che *3SAT* è riducibile in tempo polinomiale a *VERTEX-COVER*. La riduzione converte una formula 3cnf ϕ in un grafo G ed un numero k, in modo tale che ϕ è soddisfacibile se e solo se G ha un vertex cover con k nodi. La conversione viene effettuata senza sapere se ϕ è soddisfacibile. In effetti, G simula ϕ. Il grafo contiene gadget che simulano le variabili e le clausole della formula. La progettazione di questi gadget richiede un minimo di ingegno.

Per il gadget di una variabile, cerchiamo una struttura in G che possa partecipare al vertex cover in uno di due modi possibili, corrispondenti ai due possibili assegnamenti di verità alla variabile. Il gadget di una variabile contiene due nodi connessi da un arco. La struttura funziona perché uno di questi nodi deve comparire nel vertex cover. Associamo arbitrariamente VERO e FALSO a questi due nodi.

Per il gadget di una clausola, cerchiamo una struttura che induca il vertex cover ad includere nodi nei gadget delle variabili che corrispondono ad almeno un letterale vero nella clausola. Il gadget contiene tre nodi ed archi aggiuntivi tali che ogni vertex cover deve includere almeno due dei nodi, o possibilmente tutti e tre. Soltanto due nodi sarebbero richiesti se uno dei nodi del gadget di una variabile contribuisse coprendo un arco, come accadrebbe se il letterale associato soddisfacesse quella clausola. Altrimenti, tre nodi sarebbero richiesti. Infine, scegliamo k in modo tale che il vertex cover ricercato abbia un nodo per ogni gadget di variabile e due nodi per ogni gadget di clausola.

DIMOSTRAZIONE. Diamo ora i dettagli della riduzione di *3SAT* a *VERTEX-COVER* che opera in tempo polinomiale. La riduzione associa ad una formula boolena ϕ un grafo G ed un valore k. Per ciascuna variabile x in ϕ, produciamo un arco che connette due nodi. Etichettiamo i due nodi in questo gadget con x ed $\overline{x}$. Impostare x a VERO corrisponde a selezionare per il vertex cover il nodo con etichetta x, laddove FALSO corrisponde al nodo con etichetta $\overline{x}$.

I gadget per le clausole sono leggermente più complicati. Ciascun gadget di clausola è una tripla di nodi che sono etichettati con i tre letterali della clausola. Questi tre nodi sono connessi l'uno all'altro ed ai nodi nei gadget delle variabili che hanno etichette identiche. Pertanto, il numero complessivo di nodi che sono presenti in G è $2m + 3l$, dove ϕ ha m variabili ed l clausole. Sia k uguale a $m + 2l$.

Per esempio, se $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$, la riduzione da ϕ produce $\langle G, k\rangle$, dove $k = 8$ e G assume la forma mostrata nella figura che segue.

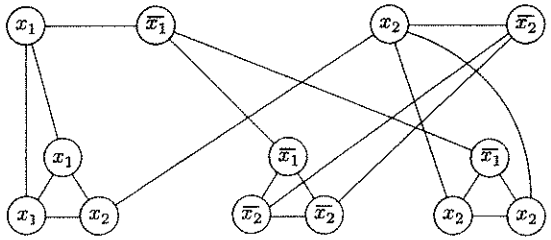


FIGURA 7.45
Il grafo che la riduzione produce a partire da
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Per provare che questa riduzione funziona, dobbiamo mostrare che ϕ è soddisfacibile se e solo se G ha un vertex cover con k nodi. Iniziamo con un assegnamento che soddisfa ϕ. Poniamo prima nel vertex cover i nodi dei gadget delle variabili che corrispondono ai letterali veri nell'assegnamento. Successivamente, selezioniamo un letterale vero in ogni clausola e poniamo i due nodi rimanenti da ogni gadget di clausola nel vertex cover. A questo punto abbiamo un totale di k nodi. Essi coprono tutti gli archi poiché ogni arco di un gadget di una variabile è chiaramente coperto, tutti e tre gli archi in ogni gadget di una clausola sono coperti, e tutti gli archi tra gadget di variabili e di clausole sono coperti. Pertanto G ha un vertex cover con k nodi.

D'altra parte, se G possiede un vertex cover con k nodi, mostriamo che ϕ è soddisfacibile costruendo l'assegnamento che soddisfa ϕ. Il vertex cover deve contenere un nodo in ciascun gadget di variabile e due in ogni gadget di clausola, al fine di coprire gli archi dei gadget delle variabili e i tre archi nei gadget delle clausole. In tal modo si tiene conto di tutti i nodi, non ne viene lasciato fuori nessuno. Prendiamo i nodi dei gadget delle variabili che sono nel vertex cover e assegniamo VERO ai letterali corrispondenti. Tale assegnamento soddisfa ϕ, poiché ciascuno dei tre archi che connettono i gadget delle variabili con ciascun gadget di clausola risulta coperto, e solo due nodi di un gadget di clausola si trovano nel vertex cover. Pertanto, uno degli archi deve essere coperto da un nodo di un gadget di variabile e, quindi, l'assegnamento soddisfa la clausola corrispondente.

Il problema del cammino Hamiltoniano

Ricordiamo che il problema del cammino Hamiltoniano chiede di stabilire se il grafo di input contiene un cammino da s a t che attraversa tutti i nodi esattamente una volta.

TEOREMA 7.46

HAMPATH è NP-completo.

IDEA. Abbiamo mostrato che *HAMPATH* appartiene a NP nella Sezione 7.3. Per far vedere che ogni problema in NP è riducibile in tempo polinomiale ad *HAMPATH*, mostriamo che *3SAT* è riducibile in tempo polinomiale a *HAMPATH*. Diamo un modo per convertire formule 3cnf in grafi, in cui i cammini Hamiltoniani corrispondono ad assegnamenti soddisfacenti la formula. I grafi contengono gadget che simulano variabili e clausole. Il gadget di una variabile è una struttura romboidale che può essere attraversata in due modi, corrispondenti ai due assegnamenti di verità. Il gadget di una clausola è un nodo. Assicurare che il cammino attraversa ciascun gadget di clausola corrisponde ad assicurare che ciascuna clausola è soddisfatta nell'assegnamento che soddisfa la formula.

DIMOSTRAZIONE. Abbiamo dimostrato in precedenza che *HAMPATH* appartiene a NP, quindi tutto ciò che resta da fare è mostrare che $3SAT \leq_P HAMPATH$. Per ciascuna formula 3cnf ϕ, facciamo vedere come costruire un grafo diretto G con due nodi, s e t, in cui esiste un cammino Hamiltoniano tra s e t se e solo se ϕ è soddisfacibile.

Iniziamo la costruzione con una formula 3cnf ϕ contenente k clausole,

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k),$$

dove ciascun a, b, e c è un letterale x_i or $\overline{x_i}$. Siano $x_1, \ldots, x_l$ le l variabili di ϕ.

Facciamo vedere ora come convertire ϕ in un grafo G. Il grafo G che costruiamo ha varie parti per rappresentare le variabili e le clausole che sono presenti in ϕ. Rappresentiamo ciascuna variabile x_i con una struttura di forma romboidale che contiene una riga orizzontale di nodi, come mostrato nella figura seguente. Specificheremo dopo il numero di nodi che compaiono nella riga orizzontale.

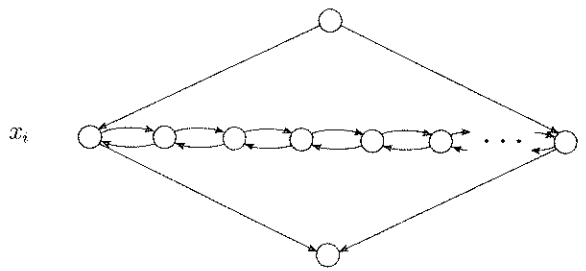


FIGURA 7.47
Rappresentazione della variabile x_i con una struttura romboidale

Rappresentiamo ciascuna clausola di ϕ con un singolo nodo, come segue.

FIGURA 7.48
Rappresentazioni della clausola c_j con un nodo

La figura seguente riporta la struttura globale di G. Essa mostra tutti gli elementi di G e le loro relazioni, eccetto gli archi che rappresentano la relazione delle variabili con le clausole che le contengono.

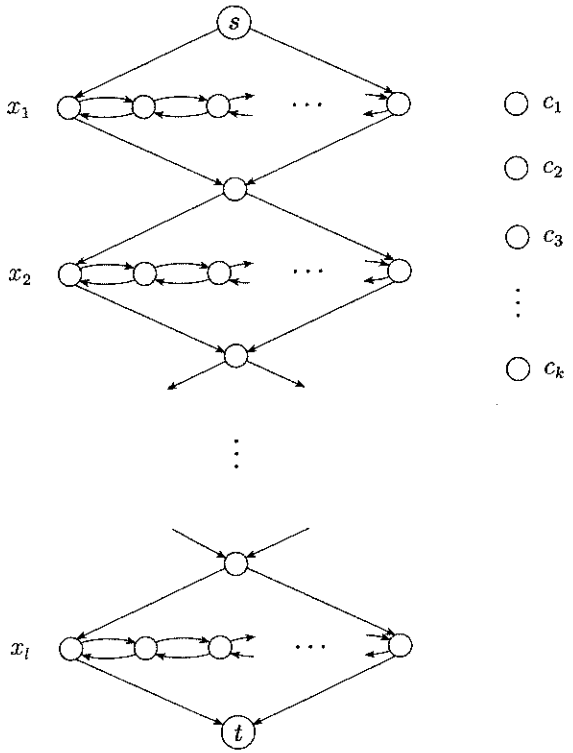


FIGURA 7.49
La struttura ad alto livello di G

Nel seguito mostriamo come collegare i rombi che rappresentano le variabili ai nodi che rappresentano le clausole. Ciascuna struttura romboidale contiene una riga orizzontale di nodi collegati tramite archi orientati in entrambe le direzioni. La riga orizzontale contiene $3k + 1$ nodi in aggiunta ai due nodi alle estremità del rombo.

Questi nodi sono raggruppati in coppie adiacenti, una per ciascuna clausola, con nodi separatori aggiuntivi tra le coppie, come mostrato nella figura seguente.

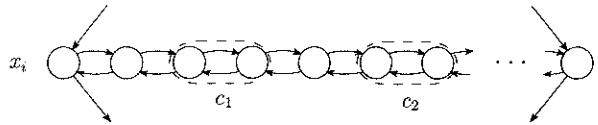


FIGURA 7.50
I nodi orizzontali in una struttura romboidale

Se la variabile x_i è presente nella clausola c_j, aggiungiamo i due archi seguenti dalla coppia j-esima nell'i-esimo rombo al j-esimo nodo della clausola.

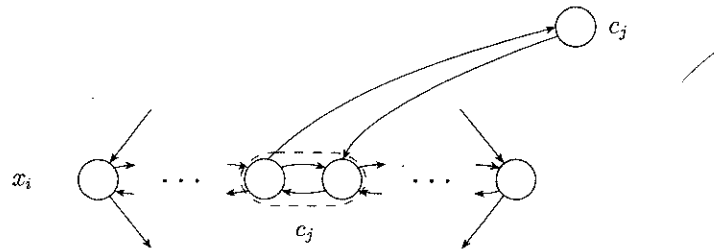


FIGURA 7.51
Gli archi aggiuntivi quando la clausola c_j contiene x_i

Se $\overline{x_i}$ è presente nella clausola c_j, aggiungiamo due archi dalla coppia j-esima nell'i-esimo rombo al j-esimo nodo di clausola, come mostrato in Figura 7.52.

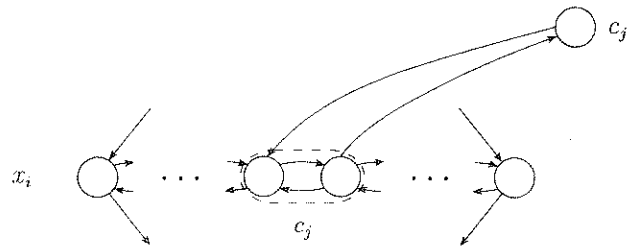


FIGURA 7.52
Gli archi aggiuntivi quando la clausola c_j contiene $\overline{x_i}$

Dopo aver aggiunto tutti gli archi corrispondenti a ciascuna occorrenza di x_i o di $\overline{x_i}$ in ciascuna clausola, la costruzione di G è completa. Per far vedere che questa costruzione funziona, dimostriamo che se ϕ è soddisfacibile, allora esiste un cammino Hamiltoniano da s a t; e, al contrario, se un tale cammino esiste, allora ϕ è soddisfacibile.

Supponiamo che ϕ sia soddisfacibile. Per mostrare un cammino Hamiltoniano da s a t, in un primo momento ignoriamo i nodi delle clausole. Il cammino inizia da s, attraversa ciascun rombo in successione, e termina in t.

Per raggiungere i nodi orizzontali in un rombo, il cammino procede a zig-zag da sinistra a destra oppure a zag-zig da destra a sinistra; l'assegnamento che soddisfa ϕ determina quale. Se a x_i viene assegnato VERO, il cammino procede a zig-zag attraverso il rombo corrispondente. Se a x_i viene assegnato FALSO, il cammino procede a zag-zig. Mostriamo entrambe le possibilità nella figura seguente.

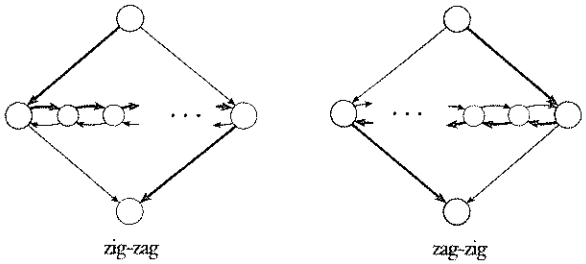


FIGURA 7.53
Zig-zag (da sinistra a destra) o zag-zig (da destra a sinistra), come stabilito dall'assegnamento soddisfacente

Fin qui questo cammino copre tutti i nodi in G, eccetto i nodi delle clausole. Possiamo facilmente includerli aggiungendo deviazioni ai nodi orizzontali. In ciascuna clausola, scegliamo uno dei letterali a cui è assegnato VERO.

Se selezioniamo x_i nella clausola c_j, possiamo deviare alla j-esima coppia nell'i-esimo rombo. Far ciò è possibile perché x_i deve essere VERO, quindi il cammino procede a zig-zag da sinistra a destra attraverso il rombo corrispondente. Pertanto, gli archi verso il nodo c_j sono nell'ordine corretto per permettere una deviazione ed un ritorno.

Allo stesso modo, se avessimo selezionato $\overline{x_i}$ nella clausola c_j, avremmo potuto deviare alla j-esima coppia nell'i-esimo rombo perché x_i deve essere FALSO, quindi il cammino procede a zag-zig da destra a sinistra attraverso il rombo corrispondente. Pertanto, gli archi verso il nodo c_j sono nuovamente nell'ordine corretto per permettere una deviazione ed un ritorno. (Si noti che ciascun letterale vero in una clausola fornisce una *opzione* di deviazione per raggiungere il nodo clausola. Come risultato, se diversi letterali in una clausola sono veri, viene presa soltanto una deviazione.) Così abbiamo costruito il cammino Hamiltoniano desiderato.

Per la direzione inversa, se G ha un cammino Hamiltoniano da s a t, facciamo vedere un assegnamento che soddisfa ϕ. Se il cammino Hamiltoniano è *normale* - cioè, passa attraverso i rombi in ordine da quello più in alto a quello più in basso, eccetto per le deviazioni verso i nodi delle clausole - possiamo facilmente ottenere l'assegnamento che soddisfa ϕ. Se il cammino procede a zig-zag attraverso il rombo, assegniamo alla variabile corrispondente VERO; e se procede a zag-zig, assegniamo FALSO. Poiché ciascun nodo clausola è presente sul cammino, osservando come la deviazione verso di esso avviene, possiamo stabilire quale dei letterali nella clausola corrispondente è VERO.

Tutto ciò che resta da mostrare è che un cammino Hamiltoniano deve essere normale. La normalità può venir meno solo se il cammino entra in una clausola da un rombo e ritorna in un altro, come nella figura seguente.

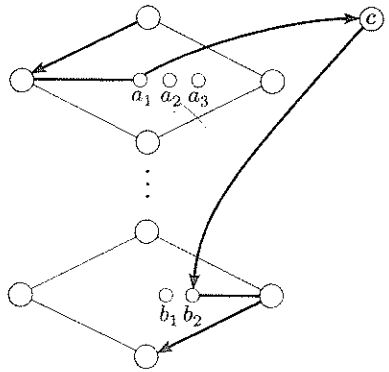


FIGURA 7.54
Questa situazione non può verificarsi

Il cammino va dal nodo a_1 a c; ma invece di ritornare in a_2 nello stesso rombo, ritorna in b_2 in un rombo differente. Se ciò accade, a_2 o a_3 deve essere un nodo separatore. Se a_2 fosse un nodo separatore, gli unici archi entranti in a_2 sarebbero da a_1 ed a_3. Se a_3 fosse un nodo separatore, a_1 ed a_2 sarebbero nella stessa coppia di clausola e, quindi, gli unici archi entranti in a_2 sarebbero da a_1, a_3, e c. In ogni caso, il cammino non potrebbe contenere il nodo a_2. Il cammino non può entrare in a_2 da c o a_1 perché il cammino va altrove da questi nodi. Il cammino non può entrare in a_2 da a_3 perché a_3 è l'unico nodo disponibile a cui a_2 punta, quindi il cammino deve uscire da a_2 passando per a_3. Pertanto, il cammino Hamiltoniano deve essere normale. Questa riduzione opera ovviamente in tempo polinomiale e la dimostrazione è completa.

Di seguito consideriamo una versione non orientata del problema del cammino Hamiltoniano, detta *UHAMPATH*. Per mostrare che *UHAMPATH* è NP-completo, diamo una riduzione di tempo polinomiale dalla versione orientata del problema.

TEOREMA 7.55

UHAMPATH è NP-completo.

DIMOSTRAZIONE. La riduzione prende in input un grafo diretto G con nodi s e t, e costruisce un grafo non orientato G' con nodi s' e t'. Il grafo G ha un cammino Hamiltoniano da s a t se e solo se G' ha un cammino Hamiltoniano da s' a t'. Descriviamo G' come segue.

Ciascun nodo u di G, eccetto s e t, viene sostituito da una tripla di nodi u^{in}, u^{mid}, ed u^{out} in G'. I nodi s e t in G sono sostituiti dai nodi $s^{\text{out}} = s'$ e $t^{\text{in}} = t'$ in G'. In G' sono presenti archi di due tipi. In primo luogo, archi che collegano u^{mid} con u^{in} ed u^{out}. In secondo luogo, un arco che collega u^{out} con v^{in} se un arco va da u a v in G. Ciò completa la costruzione di G'.

Possiamo dimostrare che questa costruzione funziona facendo vedere che G ha un cammino Hamiltoniano da s a t se e solo se G' ha un cammino Hamiltoniano da s^{out} a t^{in}. Per mostrare una direzione, osserviamo che un cammino Hamiltoniano P in G,

$$s, u_1, u_2, \ldots, u_k, t,$$

ha un corrispondente cammino Hamiltoniano P' in G',

$$s^{\text{out}}, u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \ldots, t^{\text{in}}.$$

Per mostrare l'altra direzione, affermiamo che qualsiasi cammino Hamiltoniano in G' da s^{out} a t^{in} deve andare da una tripla di nodi ad una tripla

di nodi, eccetto per l'inizio e la fine, come fa il cammino P' che abbiamo appena descritto. Ciò completerebbe la dimostrazione perché qualsiasi cammino di questo tipo ha un cammino Hamiltoniano corrispondente in G. Proviamo l'affermazione seguendo il cammino che parte dal nodo s^{out}. Si osservi che il nodo successivo nel cammino deve essere u_i^{in} per qualche i, perché solo quei nodi sono collegati a s^{out}. Il nodo successivo deve essere u_i^{mid}, perché non c'è altra strada disponibile per includere u_i^{mid} nel cammino Hamiltoniano. Dopo u_i^{mid} viene u_i^{out}, perché è l'unico altro nodo a cui u_i^{mid} è collegato. Il nodo successivo deve essere u_j^{in} per qualche j, poiché nessun altro nodo disponibile è collegato a u_i^{out}. L'argomento successivamente si ripete fino a quando viene raggiunto t^{in}.

Il problema SUBSET-SUM

Riprendiamo il problema *SUBSET-SUM* definito a pagina 317. Nel problema viene data una collezione di numeri $x_1, \ldots, x_k$ insieme con un numero obiettivo t, ed occorreva stabilire se la collezione contenesse una sottocollezione la cui somma fosse uguale a t. Ora facciamo vedere che il problema è NP-completo.

TEOREMA 7.56

SUBSET-SUM è NP-completo.

IDEA. Abbiamo già dimostrato che *SUBSET-SUM* appartiene a NP nel Teorema 7.25. Proviamo che tutti i linguaggi appartenenti a NP sono riducibili in tempo polinomiale a *SUBSET-SUM* riducendo il linguaggio NP-completo *3SAT* ad esso. Data una formula 3cnf ϕ, costruiamo un'istanza del problema *SUBSET-SUM* che contiene una sottocollezione la cui somma è uguale a t se e solo se ϕ è soddisfacibile. Chiamiamo questa sottocollezione T.

Per ottenere la riduzione, individuiamo strutture del problema *SUBSET-SUM* che rappresentano variabili e clausole. L'istanza del problema *SUBSET-SUM* che costruiamo contiene numeri grandi espressi in notazione decimale. Rappresentiamo le variabili attraverso coppie di numeri e le clausole attraverso determinate posizioni nelle rappresentazioni decimali dei numeri.

Rappresentiamo la variabile x_i con due numeri, y_i e z_i. Proviamo che y_i o z_i deve stare in T per ciascun i, condizione che stabilisce l'impostazione del valore di verità di x_i.

La posizione di ciascuna clausola contiene un certo valore nell'obiettivo t, che impone un requisito sul sottoinsieme T. Proviamo che questo requisito

è lo stesso che è presente nella clausola corrispondente - ossia, che a uno dei letterali nella clausola viene assegnato VERO.

DIMOSTRAZIONE. Già sappiamo che *SUBSET-SUM* $\in$ NP, quindi ora mostriamo che *3SAT* $\leq_P$ *SUBSET-SUM*.

Sia ϕ una formula booleana con variabili $x_1, \ldots, x_l$ e clausole $c_1, \ldots, c_k$. La riduzione converte ϕ in un'istanza $\langle S, t\rangle$ del problema *SUBSET-SUM*, in cui gli elementi di S ed il numero t sono le righe nella tabella in Figura 7.57, espressi nella notazione decimale ordinaria. Le righe al di sopra della linea doppia sono etichettate

$$y_1, z_1, y_2, z_2, \ldots, y_l, z_l \quad \text{e} \quad g_1, h_1, g_2, h_2, \ldots, g_k, h_k$$

e costituiscono gli elementi di S. La riga sotto la linea doppia è t.

Pertanto, S contiene una coppia di numeri, y_i, z_i, per ciascuna variabile x_i in ϕ. La rappresentazione decimale di questi numeri è in due parti, come indicato nella tabella. La parte sinistra comprende un 1 seguito da $l - i$ cifre 0. La parte destra contiene una cifra per ciascuna clausola, dove la cifra di y_i in colonna c_j è 1 se la clausola c_j contiene il letterale x_i, e la cifra di z_i in colonna c_j è 1 se la clausola c_j contiene il letterale $\overline{x_i}$. Le cifre non specificate uguali a 1 sono uguali a 0.

La tabella è parzialmente riempita per illustrare esempi di clausole, c_1, c_2, e c_k:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots).$$

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

FIGURA 7.57
Riduzione di *3SAT* a *SUBSET-SUM*

Inoltre, S contiene una coppia di numeri, g_j, h_j, per ciascuna clausola c_j. Questi due numeri sono uguali e consistono di un 1 seguito da $k - j$ cifre 0. Infine, il numero obiettivo t, la riga in basso nella tabella, consiste di l cifre uguali a 1 seguite da k cifre uguali a 3.

Di seguito mostriamo perché questa costruzione funziona. Facciamo vedere che ϕ è soddisfacibile se e solo se qualche sottoinsieme di S ha somma t.

Supponiamo che ϕ sia soddisfacibile. Costruiamo un sottoinsieme di S come segue. Scegliamo y_i se a x_i è assegnato VERO, e z_i se a x_i è assegnato FALSO. Se sommiamo ciò che abbiamo scelto fino ad ora, otteniamo un 1 in ciascuna delle prime l cifre perché abbiamo selezionato y_i o z_i per ciascun i. Inoltre, ciascuna delle ultime k cifre è un numero tra 1 e 3 perché ciascuna clausola è soddisfatta e quindi contiene tra 1 e 3 letterali veri. In aggiunta scegliamo abbastanza numeri g ed h per portar ciascuna delle ultime k cifre fino a 3, raggiungendo così l'obiettivo.

Supponiamo che un sottoinsieme di S abbia somma t. Costruiamo un assegnamento che soddisfa ϕ dopo aver fatto alcune osservazioni. La prima è che tutte le cifre negli elementi di S sono 0 o 1. La seconda è che ciascuna colonna nella tabella che descrive S contiene al più cinque 1. Pertanto, un "riporto" nella colonna successiva non si verifica mai quando un sottoinsieme di S viene sommato. Per ottenere un 1 in ciascuna delle prime l colonne, il sottoinsieme deve avere y_i oppure z_i per ciascun i, ma non entrambi.

Costruiamo ora l'assegnamento che soddisfa ϕ. Se il sottoinsieme contiene y_i, assegniamo a x_i VERO; altrimenti, assegniamo ad essa FALSO. Questo assegnamento deve soddisfare ϕ perché in ciascuna delle k colonne finali, la somma è sempre 3. In colonna c_j, al più 2 può venire da g_j ed h_j, quindi almeno un 1 in questa colonna deve venire da qualche y_i o z_i nel sottoinsieme. Se è y_i, allora x_i è presente in c_j e gli viene assegnato VERO, quindi c_j è soddisfatta. Se è z_i, allora $\overline{x_i}$ è presente in c_j e ad x_i viene assegnato FALSO, quindi c_j è soddisfatta. Pertanto, ϕ è soddisfatta.

Infine, dobbiamo esser certi che la riduzione possa essere effettuata in tempo polinomiale. La tabella ha taglia approssimativamente $(k + l)^2$ e ciascuna entrata può esser calcolata facilmente per qualsiasi ϕ. Quindi, il tempo complessivo consiste in $O(n^2)$ semplici passi.

ESERCIZI

7.1 Si risponda a ciascun punto con VERO o FALSO.

a. $2n = O(n)$.
b. $n^2 = O(n)$.
[A]c. $n^2 = O(n \log^2 n)$.
[A]d. $n \log n = O(n^2)$.
e. $3^n = 2^{O(n)}$.
f. $2^{2^n} = O(2^{2^n})$.

7.2 Si risponda a ciascun punto con VERO o FALSO.

a. $n = o(2n)$.
b. $2n = o(n^2)$.
[A]c. $2^n = o(3^n)$.
[A]d. $1 = o(n)$.
e. $n = o(\log n)$.
f. $1 = o(1/n)$.

7.3 Quali delle seguenti coppie sono formate da numeri relativamente primi? Si mostrino i calcoli che portano alle conclusioni.

a. 1274 e 10505
b. 7289 e 8029

7.4 Si riempia la tabella descritta nell'algoritmo di tempo polinomiale per il riconoscimento di linguaggi context-free del Teorema 7.16 per la stringa $w =$ baba e la CFG G:

$$S \to RT$$
$$R \to TR \mid \mathtt{a}$$
$$T \to TR \mid \mathtt{b}$$

7.5 La formula seguente è soddisfacibile?

$$(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$$

7.6 Si mostri che P è chiusa rispetto ad unione, concatenazione, e complemento.

7.7 Si mostri che NP è chiusa rispetto ad unione e concatenazione.

7.8 Sia $CONNECTED = \{\langle G \rangle \mid G$ è un grafo non orientato connesso$\}$. Si analizzi l'algoritmo dato a pagina 195 per mostrare che questo linguaggio appartiene a P.

7.9 Un ***triangolo*** in un grafo non orientato è una clique di dimensione 3. Si mostri che $TRIANGLE \in$ P, dove $TRIANGLE = \{\langle G \rangle \mid G$ contiene un triangolo$\}$.

7.10 Si mostri che ALL_{DFA} appartiene a P.

7.11 Per entrambi i punti, si fornisca un'analisi della complessità di tempo dell'algoritmo proposto.

a. Si mostri che $EQ_{\mathsf{DFA}} \in$ P.
b. Si dice che un linguaggio A è ***chiuso rispetto a star*** se $A = A^*$. Si dia un algoritmo di tempo polinomiale per verificare se un DFA riconosce un linguaggio chiuso rispetto a star. (Si noti che non si sa se EQ_{NFA} appartiene a P.)

7.12 I grafi G e H sono ***isomorfi*** se i nodi di G possono essere riordinati in modo tale che risulti identico ad H. Sia $ISO = \{\langle G, H \rangle \mid G$ ed H sono grafi isomorfi$\}$. Si mostri che $ISO \in$ NP.

PROBLEMI

7.13** Questo problema studia ***risoluzione, un metodo per provare la non soddisfacibilità di formule cnf. Sia $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ una formula cnf, dove le C_i sono le sue clausole. Sia $\mathcal{C} = \{C_i \mid C_i$ è una clausola di $\phi\}$. In un *passo di risoluzione*, prendiamo due clausole C_a e C_b in $\mathcal{C}$, che hanno entrambe qualche variabile x presente positivamente in una delle clausole e negativamente nell'altra. Pertanto, $C_a = (x \vee y_1 \vee y_2 \vee \cdots \vee y_k)$ e $C_b = (\overline{x} \vee z_1 \vee z_2 \vee \cdots \vee z_l)$, dove gli y_i e z_i sono letterali. Si formi la nuova clausola $(y_1 \vee y_2 \vee \cdots \vee y_k \vee z_1 \vee z_2 \vee \cdots \vee z_l)$ e si rimuovano i letterali ripetuti. Si aggiunga questa nuova clausola a $\mathcal{C}$. Si ripetano i passi di risoluzione fino a quando non può essere ottenuta alcuna clausola aggiuntiva. Se la clausola vuota ( ) è presente in $\mathcal{C}$, allora si dichiari ϕ non soddisfacibile.

Diremo che risoluzione è ***solido*** se non dichiara mai soddisfacibili formule che non sono soddisfacibili. Diremo che risoluzione è ***completo*** se tutte le formule non soddisfacibili sono dichiarate non soddisfacibili.

a. Si mostri che risoluzione è solido e completo.
b. Si usi il punto (a) per mostrare che $2SAT \in$ P.

***7.14** Si mostri che P è chiusa rispetto agli omomorfismi se e solo se P = NP.

***7.15** Sia $A \subseteq 1^*$ un qualsiasi linguaggio unario. Si mostri che se A è NP-completo, allora P = NP. (Suggerimento: si consideri una riduzione di tempo polinomiale f da SAT ad A. Per la formula ϕ, sia ϕ_{0100} la formula ridotta dove le variabili x_1, x_2, x_3, ed x_4 in ϕ sono impostate ai valori 0, 1, 0, e 0, rispettivamente. Cosa accade quando si applica f a tutte queste formule ridotte, che sono presenti in quantità esponenziale?)

7.16 In un grafo diretto, il ***grado in ingresso*** di un nodo è il numero di archi entranti e il ***grado in uscita*** è il numero di archi uscenti. Si mostri che il problema seguente è NP-completo. Dato un grafo non orientato G ed un sottoinsieme designato C dei nodi di G, è possibile convertire G in un grafo diretto assegnando direzioni a ciascuno dei suoi archi in modo tale che ogni nodo in C abbia grado in ingresso 0 o grado in uscita 0, e ogni altro nodo in G abbia grado in ingresso almeno 1?

7.17** Diciamo ***star-free un'espressione regolare che non contiene alcuna operazione star. Allora, sia $EQ_{\mathsf{SF-REX}} = \{\langle R, S \rangle \mid R$ ed S sono espressioni regolari equivalenti star-free$\}$. Si mostri che $EQ_{\mathsf{SF-REX}}$ appartiene a coNP. Perché l'argomento fallisce per espressioni regolari generali?

7.18 La ***gerarchia della differenza*** $\mathrm{D_iP}$ è definita ricorsivamente come

a. $\mathrm{D_1P} =$ NP e
b. $\mathrm{D_iP} = \{A \mid A = B \setminus C$ per $B \in$ NP e $C \in \mathrm{D_{i-1}P}\}$.
(Qui $B \setminus C = B \cap \overline{C}$.)

Per esempio, un linguaggio appartenente a $\mathrm{D_2P}$ è la differenza di due linguaggi appartenenti a NP. Qualche volta $\mathrm{D_2P}$ è chiamato DP (e può essere scritto $\mathrm{D^P}$). Sia

$$Z = \{\langle G_1, k_1, G_2, k_2 \rangle \mid G_1 \text{ ha una clique di dimensione } k_1 \text{ e } G_2 \text{ non ne ha una di dimensione } k_2\}.$$

Si mostri che Z è completo per DP. In altre parole, si mostri che Z appartiene a DP e ogni linguaggio appartenente a DP è riducibile in tempo polinomiale a Z.

*7.19 Sia $MAX\text{-}CLIQUE = \{\langle G, k\rangle |$ una clique massima in G ha dimensione esattamente $k\}$. Si usi il risultato del Problema 7.18 per mostrare che $MAX\text{-}CLIQUE$ è DP-completo.

*7.20 Sia $f: \mathcal{N} \longrightarrow \mathcal{N}$ una qualsiasi funzione per cui $f(n) = o(n \log n)$. Si mostri che $\text{TIME}(f(n))$ contiene soltanto i linguaggi regolari.

7.21 Diciamo che due formule booleane sono ***equivalenti*** se hanno lo stesso insieme di variabili e sono vere sullo stesso insieme di assegnamenti per quelle variabili (i.e., esse descrivono la stessa funzione booleana). Una formula booleana è ***minimale*** se nessuna formula più corta è equivalente ad essa. Sia $MIN\text{-}FORMULA$ la collezione delle formule booleane minimali. Si mostri che se P = NP, allora $MIN\text{-}FORMULA \in$ P.

7.22 Si modifichi l'algoritmo per il riconoscimento dei linguaggi context-free nella dimostrazione del Teorema 7.16 per fornire un algoritmo di tempo polinomiale che produce un albero sintattico per una stringa, data la stringa ed una CFG, se detta grammatica genera la stringa.

7.23 Per una formula cnf ϕ con m variabili e c clausole, si mostri che si può costruire in tempo polinomiale un NFA con $O(cm)$ stati che accetta tutti gli assegnamenti che non soddisfano la formula, rappresentati come stringhe booleane di lunghezza m. Si concluda che P $\neq$ NP implica che gli NFA non possono essere minimizzati in tempo polinomiale.

*7.24 Una ***formula 2cnf*** è un AND di clausole, dove ciascuna clausola è un OR di al più due letterali. Sia $2SAT = \{\langle\phi\rangle | \ \phi$ è una formula 2cnf soddisfacibile$\}$. Si mostri che $2SAT \in$ P.

*7.25 Si consideri l'algoritmo $MINIMIZE$, che prende un DFA M in input e dà in output un DFA M'.

$MINIMIZE$ = "Su input $\langle M\rangle$, dove $M = (Q, \Sigma, \delta, q_0, A)$ è un DFA:

1. Rimuovi tutti gli stati di M che non sono raggiungibili dallo stato iniziale.
2. Costruisci il seguente grafo non orientato G i cui nodi sono gli stati di M.
3. Colloca archi in G per connettere ogni stato di accettazione con ogni stato di non accettazione. Aggiungi ulteriori archi come segue.
4. Ripeti fino a quando nessun arco nuovo può essere aggiunto a G:
5. Per ogni coppia di stati distinti q ed r di M ed ogni $a \in \Sigma$:
6. Aggiungi l'arco (q, r) a G se $(\delta(q, a), \delta(r, a))$ è un arco di G.
7. Per ogni stato q, sia $[q]$ la collezione di stati
   $[q] = \{r \in Q |$ nessun arco unisce q ed r in $G\}$.
8. Crea un nuovo DFA $M' = (Q', \Sigma, \delta', q_0', A')$ dove
   $Q' = \{[q] | \ q \in Q\}$ (se $[q] = [r]$, solo uno di essi appartiene a Q'),
   $\delta'([q], a) = [\delta(q, a)]$ per ogni $q \in Q$ ed $a \in \Sigma$,
   $q_0' = [q_0]$, e
   $A' = \{[q] | \ q \in A\}$.
9. Output $\langle M'\rangle$."

a. Si mostri che M ed M' sono equivalenti.

b. Si mostri che M' è minimale - cioè, nessun DFA con meno stati riconosce lo stesso linguaggio. È possibile usare il risultato del Problema 7.48 senza dimostrazione.

c. Si mostri che $MINIMIZE$ opera in tempo polinomiale.

7.26 Nella dimostrazione del teorema di Cook e Levin, una finestra è un rettangolo di celle 2×3. Si faccia vedere perché la dimostrazione avrebbe fallito se invece avessimo usato finestre 2×2.

7.27 Questo problema è ispirato al gioco a singolo giocatore *Dragamine*, generalizzato ad un grafo arbitrario. Sia G un grafo non orientato, dove ciascun nodo contiene una singola *mina* nascosta o è vuoto. Il giocatore sceglie i nodi, uno per uno. Se il giocatore sceglie un nodo contenente una mina, il giocatore perde. Se il giocatore sceglie un nodo vuoto, il giocatore apprende il numero di nodi vicini che contengono mine. (Un nodo vicino è un nodo collegato al nodo scelto da un arco.) Il giocatore vince se e quando tutti i nodi vuoti sono stati scelti in questo modo.

Nel *problema della consistenza delle mine*, viene fornito un grafo G insieme a numeri che etichettano alcuni dei nodi di G. È necessario stabilire se è possibile un collocamento delle mine sui nodi rimanenti, in modo tale che qualsiasi nodo v che ha etichetta m ha esattamente m nodi vicini che contengono mine. Si formuli il problema in termini di linguaggio e si mostri che è NP-completo.

^7.28 Nel gioco seguente del solitario, viene fornita una tavola $m \times m$. Su ciascuna delle sue m^2 posizioni si trova o una pietra blu, o una pietra rossa, o niente di niente. Si gioca rimuovendo le pietre dalla tavola fino a quando ciascuna colonna contiene solo pietre di un singolo colore e ciascuna riga contiene almeno una pietra. Si vince quando si raggiunge l'obiettivo. La vittoria può essere possibile come può non esserlo, a seconda della configurazione iniziale. Sia $SOLITAIRE = \{\langle G\rangle | \ G$ è una configurazione del gioco che ammette vittoria$\}$. Si provi che $SOLITAIRE$ è NP-completo.

7.29 Sia $SET\text{-}SPLITTING = \{\langle S, C\rangle | \ S$ è un insieme finito e $C = \{C_1, \ldots, C_k\}$ è una collezione di sottoinsiemi di S, per qualche $k > 0$, tale che gli elementi di S possono essere colorati *rosso* o *blu* in modo tale che nessun C_i abbia tutti i suoi elementi colorati con lo stesso colore$\}$. Si mostri che $SET\text{-}SPLITTING$ è NP-completo.

7.30 Si consideri il seguente problema di pianificazione. Sono dati una lista di esami finali $F_1, \ldots, F_k$ da pianificare, ed una lista di studenti $S_1, \ldots, S_l$. Ciascuno studente sostiene un sottoinsieme specificato di questi esami. Occorre pianificare gli esami in archi temporali in modo tale che a nessuno studente venga richiesto di sostenere due esami nello stesso arco temporale. Il problema consiste nel determinare se esiste una tale pianificazione che usa soltanto h archi temporali. Si formuli il problema in termini di linguaggio e si mostri che il linguaggio è NP-completo.

7.31 Ricordiamo che, nella discussione della tesi di Church e Turing, abbiamo introdotto il linguaggio $D = \{\langle p\rangle | \ p$ è un polinomio a più variabili che ha una radice intera$\}$. Abbiamo asserito, ma non dimostrato, che D è indecidibile. In questo problema si chiede di provare una proprietà differente di D - ossia, che D è NP-hard. Un problema è **NP-*hard*** se tutti i problemi appartenenti a NP sono riducibili ad esso in tempo polinomiale, anche se esso stesso potrebbe non appartenere a NP. Pertanto, si chiede di mostrare che tutti i problemi in NP sono riducibili in tempo polinomiale a D.

7.32 Un sottoinsieme di nodi di un grafo G è un ***insieme dominante*** se ogni altro nodo di G è adiacente a qualche nodo nel sottoinsieme. Sia

$$DOMINATING\text{-}SET = \{\langle G, k\rangle | \ G \text{ ha un insieme dominante con } k \text{ nodi}\}.$$

Si mostri che è NP-completo dando una riduzione da $VERTEX\text{-}COVER$.

*7.33 Si mostri che il problema seguente è NP-completo. Dati un insieme di stati $Q = \{q_0, q_1, \ldots, q_l\}$ ed una collezione di coppie $\{(s_1, r_1), \ldots, (s_k, r_k)\}$ dove gli s_i sono stringhe distinte su $\Sigma = \{0, 1\}$, e gli r_i sono elementi di Q (non necessariamente distinti), si determini se esiste un DFA $M = (Q, \Sigma, \delta, q_0, F)$ dove $\delta(q_0, s_i) = r_i$ per ciascun i. Qui, $\delta(q, s)$ è lo stato in cui M entra dopo aver letto s, partendo dallo stato q. (Si noti che F è irrilevante in questo contesto.)

7.34 Sia $U = \{\langle M, x, \#^t\rangle |$ NTM M accetta x in t passi su almeno una diramazione$\}$. Si noti che ad M non viene richiesto di arrestarsi su tutte le diramazioni. Si mostri che U è NP-completo.

*7.35 Si mostri che se P = NP, esiste un algoritmo di tempo polinomiale che produce un assegnamento che soddisfa la formula quando riceve una formula booleana soddisfacibile. (Una nota: l'algoritmo che si chiede di fornire calcola una funzione; ma NP contiene linguaggi, non funzioni. L'assunzione P = NP implica che *SAT* appartiene a P, pertanto verificare la soddisfacibilità è fattibile in tempo polinomiale. Ma l'assunzione non dice come questo test venga effettuato, ed il test potrebbe non rivelare gli assegnamenti che soddisfano le formule. Occorre mostrare che possono comunque essere trovati. Suggerimento: si usi ripetutamente il verificatore della soddisfacibilità per trovare l'assegnamento bit a bit.)

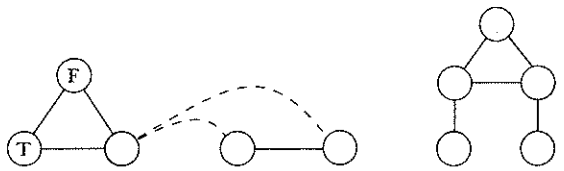
*7.36 Si mostri che se P = NP, è possibile fattorizzare interi in tempo polinomiale. (Si veda la nota nel Problema 7.35.)

A*7.37 Si mostri che se P = NP, esiste un algoritmo di tempo polinomiale che prende un grafo non orientato in input e trova una clique massima contenuta nel grafo. (Si veda la nota nel Problema 7.35.)

7.38 Una ***colorazione*** di un grafo è un assegnamento di colori per i suoi nodi tale che a nessuna coppia di nodi adiacenti venga assegnato lo stesso colore. Sia

$$3COLOR = \{\langle G\rangle |\ G \text{ è colorabile con 3 colori}\}.$$

Si mostri che *3COLOR* è NP-completo. (Suggerimento: si usino i tre sottografi seguenti.)

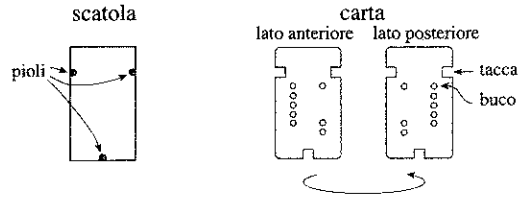


palette

variabile

gadget per OR

7.39 Siano dati una scatola e una collezione di carte come indicato nella figura seguente. A causa dei pioli nella scatola e delle tacche nelle carte, ciascuna carta entra nella scatola in due modi. Ciascuna carta contiene due colonne di buchi, alcuni dei quali possono non essere perforati. Il puzzle è risolto collocando tutte le carte nella scatola in modo tale da ricoprire completamente la base della scatola (i.e., ogni posizione di un buco è ostruita da almeno una carta che non ha un buco in quella posizione.) Sia $PUZZLE = \{\langle c_1, \ldots, c_k\rangle |$ ciascun c_i rappresenta una carta e la collezione di carte ha una soluzione$\}$. Si mostri che *PUZZLE* è NP-completo.



7.40 Sia

$$MODEXP = \{\langle a, b, c, p\rangle |\ a, b, c, \text{ e } p \text{ sono interi positivi in binario tali che } a^b \equiv c \pmod p\}.$$

Si mostri che $MODEXP \in$ P. (Si noti che l'algoritmo più immediato non computa in tempo polinomiale. Suggerimento: si provi prima quando b è una potenza di 2.)

7.41 Una ***permutazione*** dell'insieme $\{1, \ldots, k\}$ è una funzione iniettiva e suriettiva sull'insieme. Quando p è una permutazione, p^t indica la composizione di p con se stessa t volte. Sia

$$PERM\text{-}POWER = \{\langle p, q, t\rangle |\ p = q^t \text{ dove } p \text{ e } q \text{ sono permutazioni su } \{1, \ldots, k\} \text{ e } t \text{ è un intero in binario}\}.$$

Si mostri che *PERM-POWER* $\in$ P. (Si noti che l'algoritmo più immediato non computa in tempo polinomiale. Suggerimento: si provi prima quando t è una potenza di 2.)

7.42 Si mostri che P è chiusa rispetto all'operazione star. (Suggerimento: si usi la programmazione dinamica. Su input $y = y_1 \cdots y_n$ per $y_i \in \Sigma$, si costruisca una tabella indicante per ciascun $i \leq j$ se la sottostringa $y_i \cdots y_j \in A^*$ per qualsiasi $A \in$ P.)

A7.43 Si mostri che NP è chiusa rispetto all'operazione star.

7.44 Sia *UNARY-SSUM* il problema della somma di sottoinsiemi in cui tutti i numeri sono rappresentati in unario. Perché la dimostrazione di NP-completezza per *SUBSET-SUM* fallisce nel mostrare che *UNARY-SSUM* è NP-completo? Si mostri che *UNARY-SSUM* $\in$ P.

7.45 Si mostri che se P = NP, allora ogni linguaggio $A \in$ P, eccetto $A = \emptyset$ ed $A = \Sigma^*$, è NP-completo.

*7.46 Si mostri che $PRIMES = \{m |\ m$ è un numero primo in binario$\} \in$ NP. (Suggerimento: per $p > 1$, il gruppo moltiplicativo $Z_p^* = \{x |\ x$ è relativamente primo a p ed $1 \leq x < p\}$ è sia ciclico che di ordine $p - 1$ se e solo se p è primo. Questa proprietà può essere usata senza giustificazione. L'enunciato più forte *PRIMES* $\in$ P è oggi noto esser vero, ma è molto più difficile da dimostrare.)

7.47 Generalmente si ritiene che *PATH* non sia NP-completo. Si spieghi la ragione che sta dietro a questa convinzione. Si mostri che provare che *PATH* non è NP-completo proverebbe che P $\neq$ NP.

7.48 Sia G un grafo non orientato. Inoltre, siano

$$SPATH = \{\langle G, a, b, k\rangle |\ G \text{ contiene un cammino semplice di lunghezza al più } k \text{ da } a \text{ a } b\},$$

e

$$LPATH = \{\langle G, a, b, k\rangle \mid G \text{ contiene un cammino semplice di lunghezza almeno } k \text{ da } a \text{ a } b\}.$$

a. Si mostri che $SPATH \in$ P.

b. Si mostri che $LPATH$ è NP-completo.

7.49 Sia $DOUBLE\text{-}SAT = \{\langle\phi\rangle \mid \phi$ ha almeno due assegnamenti che la soddisfano$\}$. Si mostri che $DOUBLE\text{-}SAT$ è NP-completo.

A7.50 Sia $HALF\text{-}CLIQUE = \{\langle G\rangle \mid G$ è un grafo non orientato che possiede un sottografo completo con almeno $m/2$ nodi, dove m è il numero di nodi di $G\}$. Si mostri che $HALF\text{-}CLIQUE$ è NP-completo.

7.51 Sia $CNF_k = \{\langle\phi\rangle \mid \phi$ è una formula cnf soddisfacibile dove ciascuna variabile è presente in al più k posti$\}$.

a. Si mostri che $CNF_2 \in$ P.

b. Si mostri che CNF_3 è NP-completo.

7.52 Sia $CNF_H = \{\langle\phi\rangle \mid \phi$ è una formula cnf soddisfacibile dove ciascuna clausola contiene un numero arbitrario di letterali, ma al più un letterale negato$\}$. Si mostri che $CNF_H \in$ P.

7.53 Sia ϕ una formula 3cnf. Un ***assegnamento*** $\neq$ per le variabili di ϕ è un assegnamento dove ciascuna clausola contiene due letterali con valori di verità diversi. In altre parole, un assegnamento $\neq$ soddisfa ϕ senza assegnare tre letterali veri in ogni clausola.

a. Si mostri che la negazione di ogni assegnamento $\neq$ per ϕ è ancora un assegnamento $\neq$.

b. Sia $\neq SAT$ la collezione di formule 3cnf che posseggono un assegnamento $\neq$. Si mostri che possiamo ottenere una riduzione di tempo polinomiale da $3SAT$ a $\neq SAT$ sostituendo ciascuna clausola c_i

$$(y_1 \vee y_2 \vee y_3)$$

con le due clausole

$$(y_1 \vee y_2 \vee z_i) \quad \text{e} \quad (\overline{z_i} \vee y_3 \vee b),$$

dove z_i è una variabile nuova per ciascuna clausola c_i, e b è una singola variabile nuova aggiuntiva.

c. Si concluda che $\neq SAT$ è NP-completo.

7.54 Un ***taglio*** in un grafo non orientato è una separazione dei vertici V in due sottoinsiemi disgiunti S e T. La cardinalità di un taglio è il numero di archi che hanno un'estremità in S e l'altra in T. Sia

$$MAX\text{-}CUT = \{\langle G, k\rangle \mid G \text{ ha un taglio di cardinalità } k \text{ o più }\}.$$

Si mostri che $MAX\text{-}CUT$ è NP-completo. Si può assumere il risultato del Problema 7.53. (Suggerimento: si mostri che $\neq SAT \leq_P MAX\text{-}CUT$. Il gadget di variabile per la variabile x è una collezione di 3c nodi etichettati con x ed altri $3c$ nodi etichettati con $\overline{x}$, dove c è il numero di clausole. Tutti i nodi etichettati con x sono collegati con tutti i nodi etichettati con $\overline{x}$. Il gadget di clausola è un triangolo di tre archi che collegano tre nodi etichettati con i letterali che sono presenti nella clausola. Non si usi lo stesso nodo in più di un gadget di clausola. Si provi che questa riduzione è corretta.)

SOLUZIONI SELEZIONATE

7.1 **(c)** FALSO; **(d)** VERO.

7.2 **(c)** VERO; **(d)** VERO.

7.28 Prima di tutto, $SOLITAIRE \in$ NP perché è possibile verificare che una soluzione è corretta in tempo polinomiale. Successivamente, si mostra che $3SAT \leq_P SOLITAIRE$. Data ϕ con m variabili $x_1, \ldots, x_m$ e k clausole $c_1, \ldots, c_k$, si costruisce il seguente gioco $k \times m$ G. Si assume che ϕ non ha clausole che contengono sia x_i che $\overline{x_i}$ perché tali clausole possono essere rimosse senza influenzare la soddisfacibilità. Se x_i è nella clausola c_j, si mette una pietra blu in riga c_j, colonna x_i. Se $\overline{x_i}$ è nella clausola c_j, si mette una pietra rossa in riga c_j, colonna x_i. Si può rendere la tavola quadrata ripetendo una riga o aggiungendo una colonna vuota se necessario senza influenzare la risolvibilità. Si fa vedere che ϕ è soddisfacibile se e solo se G ha una soluzione.

($\rightarrow$) Si consideri un assegnamento soddisfacibile. Se x_i è vero (falso), si rimuovono le pietre rosse (blu) dalla colonna corrispondente. Quindi restano le pietre corrispondenti ai letterali veri. Poiché ogni clausola ha un letterale vero, ogni riga ha una pietra.

($\leftarrow$) Si consideri una soluzione del gioco. Se le pietre rosse (blu) sono state rimosse dalla colonna, si imposta la variabile corrispondente a vero (falso). Ogni riga ha una pietra rimanente, quindi ogni clausola ha un letterale vero. Pertanto ϕ è soddisfatta.

7.37 Assumendo che P = NP, risulta $CLIQUE \in$ P, ed è possibile verificare se G contiene una clique di dimensione k in tempo polinomiale, per ogni valore di k. Verificando se G contiene una clique per ciascuna dimensione, da 1 fino al numero di nodi di G, è possibile stabilire la dimensione t di una clique massimale di G in tempo polinomiale. Una volta nota t, è possibile trovare una clique con t nodi come segue. Per ciascun nodo x di G, si rimuove x e si calcola la dimensione massima risultante in una clique. Se la dimensione risultante decresce, si rimette x e si continua con il nodo successivo. Se la dimensione risultante è ancora t, si rimuove x permanentemente e si continua con il nodo successivo. Quando tutti i nodi sono stati considerati in questo modo, i nodi restanti costituiscono una clique di dimensione t.

7.43 Sia $A \in$ NP. Si costruisce una NTM M per decidere A^* in tempo polinomiale non deterministico.

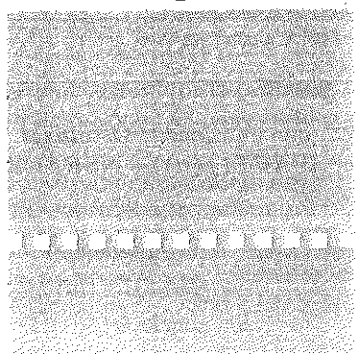
$M =$ "Su input w:

1. Dividi non deterministicamente w in parti $w = x_1 x_2 \cdots x_k$.
2. Per ciascun x_i, ipotizza non deterministicamente il certificato che mostra che $x_i \in A$.
3. Verifica tutti i certificati se possibile, poi *accetta*. Altrimenti, se la verifica fallisce, *rifiuta*."

7.50 Diamo una funzione di riduzione di tempo polinomiale da $CLIQUE$ a $HALF\text{-}CLIQUE$. L'input della riduzione è la coppia $\langle G, k\rangle$ e la riduzione produce il grafo $\langle H\rangle$ in output dove H è come segue. Se G ha m nodi e $k = m/2$, allora $H = G$. Se $k < m/2$, allora H è il grafo ottenuto da G aggiungendo j nodi, ciascuno connesso ad ognuno dei nodi originali e tra loro, dove $j = m - 2k$. Pertanto, H ha

$m + j = 2m - 2k$ nodi. Si noti che G ha una clique di dimensione k se e solo se H ha una clique di dimensione $k + j = m - k$, e quindi $\langle G, k \rangle \in CLIQUE$ se e solo se $\langle H \rangle \in HALF\text{-}CLIQUE$. Se $k > m/2$, allora H è il grafo ottenuto aggiungendo j nodi a G senza archi aggiuntivi, dove $j = 2k - m$. Quindi, H ha $m + j = 2k$ nodi, e pertanto G ha una clique di dimensione k se e solo se H ha una clique di dimensione k. Per cui, $\langle G, k \rangle \in CLIQUE$ se e solo se $\langle H \rangle \in HALF\text{-}CLIQUE$. Occorre anche far vedere che $HALF\text{-}CLIQUE \in$ NP. Il certificato è semplicemente la clique.

8



COMPLESSITÀ DI SPAZIO

In questo capitolo valutiamo la complessità dei problemi computazionali in termini della quantità di spazio, o memoria, che essi richiedono. Tempo e spazio sono due dei fattori più importanti quando ricerchiamo soluzioni pratiche per numerosi problemi computazionali. La complessità di spazio condivide molte delle caratteristiche della complessità di tempo e rappresenta un modo ulteriore per classificare i problemi in accordo alla propria difficoltà computazionale.

Come per la complessità di tempo, abbiamo necessità di selezionare un modello per misurare lo spazio utilizzato da un algoritmo. Continuiamo a servirci del modello della macchina di Turing per la stessa ragione per cui l'abbiamo usato per misurare il tempo. Le macchine di Turing sono matematicamente semplici ed abbastanza vicine ai calcolatori reali da fornire risultati significativi.

DEFINIZIONE **8.1**

Sia M una macchina di Turing deterministica che si arresta su tutti gli input. La ***complessità di spazio*** di M è la funzione $f: \mathcal{N} \longrightarrow \mathcal{N}$, dove $f(n)$ è il numero massimo di celle del nastro che M scandisce su ogni input di lunghezza n. Se la complessità di spazio di M è $f(n)$, diciamo anche che M computa in spazio $f(n)$.

Se M è una macchina di Turing non deterministica dove tutte le diramazioni si arrestano su tutti gli input, definiamo la sua complessità di spazio $f(n)$ come il massimo numero di celle del nastro che M scandisce su qualsiasi diramazione della sua computazione per ogni input di lunghezza n.

Tipicamente stimiamo la complessità di spazio delle macchine di Turing attraverso la notazione asintotica.

DEFINIZIONE **8.2**

Sia $f: \mathcal{N} \longrightarrow \mathcal{R}^+$ una funzione. Le ***classi di complessità di spazio***, $\mathbf{SPACE}(f(n))$ e $\mathbf{NSPACE}(f(n))$, sono definite come segue.

$\text{SPACE}(f(n)) = \{L \mid L$ è un linguaggio deciso da una macchina di Turing deterministica con spazio $O(f(n))\}$.

$\text{NSPACE}(f(n)) = \{L \mid L$ è un linguaggio deciso da una macchina di Turing non deterministica con spazio $O(f(n))\}$.

ESEMPIO 8.3

Nel Capitolo 7 abbiamo introdotto il problema NP-completo SAT. Qui mostriamo che SAT può essere risolto con un algoritmo che richiede una quantità di spazio lineare. Riteniamo che SAT non possa essere risolto con un algoritmo di tempo polinomiale, men che meno con un algoritmo di tempo lineare, perché SAT è NP-completo. Lo spazio sembra essere più potente del tempo perché lo spazio può essere riutilizzato, mentre il tempo no.

M_1 = "Su input $\langle \phi \rangle$, dove ϕ è una formula booleana:

1. Per ogni assegnamento di verità alle variabili $x_1, \ldots, x_m$ di ϕ:
2. Valuta ϕ sull'assegnamento di verità.
3. Se ϕ vale 1, *accetta*; altrimenti, *rifiuta*."

La macchina M_1 chiaramente computa in spazio lineare perché ad ogni iterazione del ciclo può riutilizzare la stessa porzione del nastro. La macchina ha necessità di memorizzare soltanto l'assegnamento di verità corrente, e ciò può esser fatto con spazio $O(m)$. Il numero di variabili m è al più n, la lunghezza dell' input, pertanto la macchina computa in spazio $O(n)$.

ESEMPIO 8.4

In questo esempio illustriamo la complessità di spazio non deterministica di un linguaggio. Nella prossima sezione mostreremo come determinare la complessità di spazio non deterministica possa essere utile per determinare la sua complessità di spazio deterministica. Si consideri il problema di

verificare se un automa finito non deterministico accetta tutte le stringhe. Sia

$$ALL_{\text{NFA}} = \{\langle A \rangle \mid A \text{ è un NFA e } L(A) = \Sigma^*\}.$$

Forniamo un algoritmo non deterministico lineare in spazio che decide il complemento di questo linguaggio, $\overline{ALL_{\text{NFA}}}$. L'idea che sta dietro all'algoritmo è di usare il non determinismo per individuare una stringa che viene rifiutata dall'NFA, e di usare una quantità di spazio lineare per tener traccia degli stati in cui l'NFA potrebbe trovarsi in un certo istante. Si noti che l'appartenenza di questo linguaggio a NP o a coNP non è nota.

N = "Su input $\langle M \rangle$, dove M è un NFA:

1. Poni un marcatore sullo stato iniziale dell'NFA.
2. Ripeti 2^q volte, dove q è il numero di stati di M:
3. Non deterministicamente seleziona un simbolo di input e cambia le posizioni dei marcatori sugli stati di M per simulare la lettura del simbolo.
4. *Accetta* se i passi 2 e 3 individuano una stringa che M rifiuta; cioè, se ad un certo punto nessuno dei marcatori si trova su stati finali di M. Altrimenti, *rifiuta*."

Se M rifiuta stringhe, ne deve rifiutare una di lunghezza al più 2^q, perché in ogni stringa più lunga che viene rifiutata, le locazioni dei marcatori descritti nell'algoritmo precedente dovrebbero ripetersi. La parte della stringa tra le ripetizioni può essere rimossa per ottenere una stringa rifiutata più corta. Pertanto, N decide $\overline{ALL_{\text{NFA}}}$. (Si noti che N accetta anche input formati impropriamente.)

Il solo spazio richiesto da questo algoritmo serve per memorizzare la locazione dei marcatori e per il contatore usato nel ciclo, e ciò può esser fatto in spazio lineare. Quindi, l'algoritmo computa con complessità di spazio non deterministica $O(n)$. Di seguito proviamo un teorema che fornisce informazioni sulla complessità di spazio deterministica di ALL_{NFA}.

8.1
TEOREMA DI SAVITCH

Il teorema di Savitch è uno dei primi risultati riguardanti la complessità di spazio. Mostra che le macchine deterministiche possono simulare le macchine non deterministiche usando una quantità di spazio sorprendentemente molto piccola. Per la complessità di tempo, tale simulazione sembra richiedere un incremento esponenziale in tempo. Per la complessità di spazio, il teorema di Savitch mostra che ogni TM non deterministica che usa spazio

$f(n)$ può essere convertita in una TM deterministica che usa soltanto spazio $f^2(n)$.

TEOREMA 8.5

Teorema di Savitch Per ogni[1] funzione $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, dove $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

IDEA. Dobbiamo simulare deterministicamente una NTM con complessità di spazio $f(n)$. Un approccio ingenuo consiste nel procedere tentando tutte le diramazioni seguite dalla computazione della NTM, una per una. La simulazione deve tener traccia di quale diramazione la macchina sta tentando al momento per poter passare alla successiva. Ma una diramazione che usa spazio $f(n)$ può richiedere $2^{O(f(n))}$ passi e ciascun passo potrebbe consistere in una scelta non deterministica. L'esplorazione sequenziale delle diramazioni richiederebbe la memorizzazione di tutte le scelte effettuate su una particolare diramazione per poter individuare la diramazione successiva. Pertanto, questo approccio può usare spazio $2^{O(f(n))}$, eccedendo il nostro obiettivo di usare spazio $O(f^2(n))$.

Invece, usiamo un approccio diverso, considerando il problema più generale che segue. Date due configurazioni della NTM, c_1 e c_2, e un numero t, verifichiamo se la NTM può transire da c_1 a c_2 in al più t passi, usando soltanto spazio $f(n)$. Chiamiamo questo problema il ***problema della resa***. Risolvendo il problema della resa, dove c_1 è la configurazione iniziale, c_2 è la configurazione finale, e t è il numero massimo di passi che la macchina non deterministica può effettuare, possiamo stabilire se la macchina accetta il proprio input.

A tal fine forniamo un algoritmo ricorsivo deterministico che risolve il problema della resa. L'algoritmo opera cercando una configurazione intermedia c_m, e verificando ricorsivamente se (1) c_1 può portare a c_m in al più $t/2$ passi, e (2) se c_m può portare a c_2 in al più $t/2$ passi. La riutilizzazione dello spazio per ognuno dei due test ricorsivi permette un risparmio di spazio significativo.

Questo algoritmo richiede spazio per memorizzare le chiamate ricorsive. Ciascun livello della ricorsione usa spazio $O(f(n))$ per memorizzare una configurazione. La profondità della ricorsione è $\log t$, dove t è il tempo massimo che la macchina non deterministica può usare su ogni diramazione. Risulta $t = 2^{O(f(n))}$, quindi $\log t = O(f(n))$. Pertanto la simulazione deterministica usa spazio $O(f^2(n))$.

DIMOSTRAZIONE. Sia N una NTM che decide il linguaggio A in spazio $f(n)$. Costruiamo una TM deterministica M che decide A. La macchina

[1] A pagina 377 mostreremo che il teorema di Savitch vale anche se $f(n) \geq \log n$.

M utilizza la procedura CANYIELD, che verifica se una delle configurazioni di N può darne un'altra entro uno specificato numero di passi. Questa procedura risolve il problema della resa descritto nell'idea della prova.

Sia w una stringa di input per N. Per le configurazioni c_1 e c_2 di N, e l'intero t, CANYIELD(c_1, c_2, t) dà in output *accetta* se N può passare dalla configurazione c_1 alla configurazione c_2 in t o meno passi attraverso un qualche cammino non deterministico. Altrimenti, CANYIELD dà in output *rifiuta*. Per convenienza, assumiamo che t sia una potenza di 2.

CANYIELD = "Su input c_1, c_2, e t:

1. Se $t = 1$, allora verifica direttamente se $c_1 = c_2$ o se c_1 dà c_2 in un passo, in accordo alle regole di N. *Accetta* se uno dei due test ha successo; *rifiuta* se falliscono entrambi.
2. Se $t > 1$, allora per ogni configurazione c_m di N usando spazio $f(n)$:
3. Esegui CANYIELD$(c_1, c_m, \frac{t}{2})$.
4. Esegui CANYIELD$(c_m, c_2, \frac{t}{2})$.
5. Se i passi 3 e 4 sono entrambi accettanti, allora *accetta*.
6. Se non ha ancora accettato, *rifiuta*."

Ora definiamo M per simulare N come segue. Prima modifichiamo N in modo tale che, quando accetta, cancella il proprio nastro e muove la testina sulla cella più a sinistra - entrando con ciò in una configurazione detta c_{accept}. Denotiamo con c_{start} la configurazione iniziale di N su w. Selezioniamo una costante d in modo tale che N abbia al più $2^{df(n)}$ configurazioni usando un nastro di $f(n)$ celle, dove n è la lunghezza di w. Allora sappiamo che $2^{df(n)}$ fornisce un limite superiore al tempo di esecuzione di ogni diramazione di N su w.

M = "Su input w:

1. Fornisci in output il risultato di CANYIELD$(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)})$."

L'algoritmo CANYIELD ovviamente risolve il problema della resa e, di conseguenza, M simula correttamente N. Dobbiamo analizzarne l'esecuzione per verificare che M lavora in spazio $O(f^2(n))$.

Ogni qualvolta CANYIELD invoca se stessa ricorsivamente, memorizza il numero della fase corrente e i valori di c_1, c_2, e t in una pila, in modo tale che questi valori possano essere recuperati dalla chiamata ricorsiva. Ciascun livello della ricorsione usa perciò uno spazio aggiuntivo pari a $O(f(n))$. Inoltre, ciascun livello della ricorsione divide la taglia di t a metà. Inizialmente t è uguale a $2^{df(n)}$, e quindi la profondità della ricorsione è $O(\log 2^{df(n)})$ ovvero $O(f(n))$. Pertanto, lo spazio totale usato è $O(f^2(n))$, come asserito.

Una difficoltà tecnica sorge in questo argomento perché l'algoritmo M deve conoscere il valore di $f(n)$ quando invoca CANYIELD. Possiamo ge-

stire questa difficoltà modificando M in modo tale che tenti con $f(n) = 1, 2, 3, \ldots$. Per ciascun valore $f(n) = i$, l'algoritmo modificato usa CANYIELD per stabilire se la configurazione di accettazione è raggiungibile. In aggiunta, l'algoritmo usa CANYIELD per stabilire se N usa almeno spazio $i + 1$ verificando se N può raggiungere qualcuna delle configurazioni di lunghezza $i + 1$ dalla configurazione iniziale.

Se la configurazione finale è raggiungibile, M accetta; se nessuna configurazione di lunghezza $i + 1$ è raggiungibile, M rifiuta; e altrimenti, M continua con $f(n) = i + 1$. (Avremmo potuto gestire questa difficoltà in un altro modo, assumendo che M possa calcolare $f(n)$ in spazio $O(f(n))$, ma poi avremmo dovuto aggiungere questa assunzione all'enunciato del teorema.)

8.2
LA CLASSE PSPACE

In analogia con la classe P, definiamo la classe PSPACE per la complessità di spazio.

DEFINIZIONE **8.6**

PSPACE è la classe dei linguaggi che sono decidibili in spazio polinomiale con una macchina di Turing deterministica. In altre parole,

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

Definiamo NPSPACE, la controparte non deterministica di PSPACE, in termini delle classi NSPACE. Tuttavia, in virtù del teorema di Savitch, PSPACE = NPSPACE, perché il quadrato di un polinomio è ancora un polinomio.

Negli esempi 8.3 e 8.4, abbiamo mostrato che SAT è contenuto in SPACE(n) e che ALL_{NFA} è contenuto in coNSPACE(n) e, quindi, per il teorema di Savitch, in SPACE(n^2), perché le classi di complessità di spazio deterministiche sono chiuse rispetto al complemento. Pertanto, entrambi i linguaggi sono in PSPACE.

Esaminiamo ora la relazione di PSPACE con P ed NP. Osserviamo che P $\subseteq$ PSPACE, perché una macchina che esegue velocemente non può usare una quantità di spazio grande. Precisamente, per $t(n) \geq n$, qualsiasi macchina che opera in tempo $t(n)$ può usare al più spazio $t(n)$, perché una

macchina può esplorare al più una nuova cella ad ogni passo della propria esecuzione. Similmente, NP $\subseteq$ NPSPACE e, pertanto, NP $\subseteq$ PSPACE.

Viceversa, possiamo limitare la complessità di tempo di una macchina di Turing in termini della sua complessità di spazio. Per $f(n) \geq n$, una TM che usa spazio $f(n)$ può avere al più $f(n)\,2^{O(f(n))}$ configurazioni differenti, attraverso una semplice generalizzazione della prova del Lemma 5.8 a pagina 234. Una computazione di una TM che si arresta non può ripetere una configurazione. Pertanto, una TM[2] che usa spazio $f(n)$ deve computare in tempo $f(n)\,2^{O(f(n))}$, quindi PSPACE $\subseteq$ EXPTIME $= \bigcup_k \text{TIME}(2^{n^k})$.

Riassumiamo le nostre conoscenze circa le relazioni tra le classi di complessità definite fino ad ora nella serie di inclusioni

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME}.$$

Non sappiamo se una qualsiasi di queste inclusioni è in realtà un'uguaglianza. Qualcuno potrebbe trovare una simulazione simile a quella usata nel teorema di Savitch che fonde alcune di queste classi nella stessa classe. Tuttavia, nel Capitolo 9 mostreremo che P $\neq$ EXPTIME. Pertanto, almeno una delle inclusioni precedenti è propria, anche se non siamo in grado di dire quale! In realtà, molti ricercatori ritengono che tutte le inclusioni siano proprie. Il diagramma seguente raffigura le relazioni tra queste classi, assumendo che siano tutte differenti.

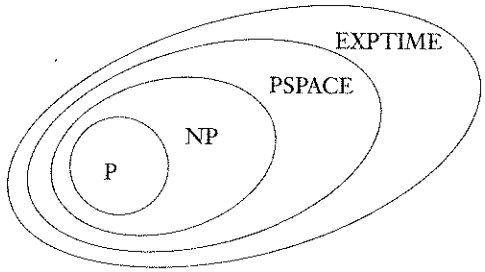


FIGURA 8.7
Relazioni congetturate tra P, NP, PSPACE, e EXPTIME

8.3
PSPACE-COMPLETEZZA

Nella Sezione 7.4 abbiamo introdotto la categoria dei linguaggi NP-completi per rappresentare i linguaggi più difficili in NP. Mostrare che un linguaggio

[2]La condizione che $f(n) \geq n$ viene generalizzata nel seguito a $f(n) \geq \log n$, quando introdurremo TM che usano spazio quasi-lineare a pagina 376.

è NP-completo fornisce un'evidenza forte che il linguaggio non appartiene a P. Se vi appartenesse, P ed NP sarebbero uguali. In questa sezione introduciamo la nozione analoga di PSPACE-completezza per la classe PSPACE.

DEFINIZIONE **8.8**

Un linguaggio B è ***PSPACE-completo*** se soddisfa due condizioni:

1. B appartiene a PSPACE, e
2. qualsiasi A appartenente a PSPACE è riducibile in tempo polinomiale a B.

Se B soddisfa soltanto la condizione 2, diciamo che è ***PSPACE-hard***.

Nel definire la PSPACE-completezza, usiamo la riducibilità polinomiale come specificata nella Definizione 7.29. Perché non definiamo una nozione di riducibilità *in spazio* polinomiale e usiamo questa invece della riducibilità in *tempo* polinomiale? Per comprendere la risposta a questa domanda importante, consideriamo in primo luogo la nostra motivazione nel definire problemi completi.

I problemi completi sono importanti perché sono esempi dei problemi più difficili in una classe di complessità. Un problema completo è più difficile perché ogni altro problema nella classe può essere facilmente ridotto ad esso. Quindi, se troviamo un modo facile per risolvere il problema completo, possiamo risolvere facilmente tutti gli altri problemi nella classe. Per applicare questo ragionamento, la riduzione deve essere *facile*, in relazione alla complessità dei problemi tipici nella classe. Se la riduzione stessa fosse difficile da calcolare, una soluzione facile al problema completo non darebbe necessariamente una soluzione facile ai problemi che ad esso si riducono.

Pertanto, la regola è: ogni volta che definiamo problemi per una classe di complessità, il modello per la riduzione deve essere più limitato del modello usato per definire la classe stessa.

Il problema TQBF

Il nostro primo esempio di problema PSPACE-completo richiede una generalizzazione del problema della soddisfacibilità. Ricordiamo che una ***formula booleana*** è un'espressione che contiene variabili booleane, le constanti 0 e 1, e gli operatori booleani $\wedge$, $\vee$, e $\neg$. Introduciamo ora un tipo di formula booleana più generale.

I ***quantificatori*** $\forall$ (per ogni) ed $\exists$ (esiste) compaiono frequentemente in enunciati matematici. Scrivere $\forall x\ \phi$ significa dire che per *tutti* i valori per la variabile x, l'asserzione ϕ è vera. Allo stesso modo, scrivere $\exists x\ \phi$ significa dire che per *qualche* valore della variabile x, l'asserzione ϕ è vera. A volte $\forall$ viene detto ***quantificatore universale*** ed $\exists$ ***quantificatore esistenziale***. Diciamo che la variabile x che segue immediatamente il quantificatore è ***legata*** al quantificatore.

Per esempio, considerando i numeri naturali, l'affermazione $\forall x\ [x + 1 > x]$ significa che il successore $x + 1$ di ogni numero naturale x è più grande del numero stesso. Ovviamente, questa affermazione è vera. Tuttavia, l'affermazione $\exists y\ [y + y = 3]$ ovviamente è falsa. Nell'interpretare il significato di enunciati che coinvolgono quantificatori, dobbiamo considerare l' ***universo*** dal quale i valori vengono presi. Nei casi precedenti, l'universo consiste nei numeri naturali: se, invece, considerassimo i numeri reali, l'affermazione quantificata esistenzialmente diventerebbe vera.

Gli enunciati possono contenere diversi quantificatori, come in $\forall x\ \exists y\ [y > x]$. Relativamente all'universo dei numeri naturali, questa affermazione asserisce che ogni numero naturale ha un altro numero naturale più grande di esso. L'ordine dei quantificatori è importante. L'inversione dell'ordine, come nell'affermazione $\exists y\ \forall x\ [y > x]$, dà un significato totalmente diverso, - ossia, che qualche numero naturale è più grande di tutti gli altri. Ovviamente, la prima affermazione è vera e la seconda è falsa.

Un quantificatore può comparire ovunque in un enunciato matematico. Si applica al frammento dell'enunciato che compare tra la coppia di parentesi tonde o quadre che seguono la variabile quantificata. Questo frammento è chiamato l'***ambito*** del quantificatore. Spesso conviene richiedere che tutti i quantificatori siano presenti all'inizio dell'enunciato, e che l'ambito di ciascun quantificatore sia tutto ciò che lo segue. Enunciati di questi tipo vengono detti in ***forma normale prenex***. Qualsiasi enunciato può essere messo facilmente in forma normale prenex. Se non specificato diversamente, consideriamo enunciati soltanto in questa forma.

Formule booleane con quantificatori sono dette ***formule booleane quantificate***. Per tali formule, l'universo è $\{0, 1\}$. Per esempio,

$$\phi = \forall x\ \exists y\ \left[(x \vee y) \wedge (\overline{x} \vee \overline{y}) \right]$$

è una formula booleana quantificata. In questo caso ϕ è vera, ma sarebbe falsa se i quantificatori $\forall x$ ed $\exists y$ venissero invertiti.

Quando ogni variabile di una formula è presente nell'ambito di qualche quantificatore, la formula si dice ***completamente quantificata***. Una formula booleana completamente quantificata a volte viene detta una ***formula chiusa***, ed è sempre vera o falsa. Per esempio, la formula precedente ϕ è completamente quantificata. Però, se la parte iniziale, $\forall x$, di ϕ venisse rimossa, la formula non sarebbe più completamente quantificata e non sarebbe nè vera nè falsa.

Il problema *TQBF* consiste nel determinare se una formula booleana completamente quantificata è vera o falsa. Definiamo il linguaggio

$TQBF = \{\langle\phi\rangle \mid \phi \text{ è una formula booleana vera completamente quantificata}\}$.

TEOREMA 8.9

TQBF è PSPACE-completo.

IDEA. Per dimostrare che *TQBF* appartiene a PSPACE, diamo un semplice algoritmo che assegna valori alle variabili e ricorsivamente valuta la verità della formula per quei valori. Da questa informazione, l'algoritmo può determinare la verità della formula quantificata originale.

Per dimostrare che ogni linguaggio A appartenente a PSPACE si riduce a *TQBF* in tempo polinomiale, consideriamo una macchina di Turing per A polinomialmente limitata in spazio. Quindi forniamo una riduzione in tempo polinomiale che mappa una stringa in una formula booleana quantificata ϕ che codifica una simulazione della macchina su quell'input. La formula è vera se e solo se la macchina accetta.

Come primo tentativo di realizzazione di questa costruzione, proviamo ad emulare la prova del teorema di Cook e Levin, il Teorema 7.37. Possiamo costruire una formula ϕ, che simula M su un input w esprimendo i requisiti per un tableau accettante. Un tableau per M su w ha larghezza $O(n^k)$, lo spazio usato da M, ma altezza esponenziale in n^k, perché M può restare in esecuzione per un tempo esponenziale. Pertanto, se dovessimo rappresentare il tableau con una formula direttamente, otterremmo una formula di dimensione esponenziale. Ma una riduzione in tempo polinomiale non può produrre un risultato di dimensione esponenziale, per cui questo tentativo di mostrare che $A \leq_P TQBF$ fallisce.

Invece, per costruire la formula utilizziamo una tecnica vicina alla prova del teorema di Savitch. La formula divide il tableau a metà ed impiega il quantificatore universale per rappresentare ciascuna metà con la stessa parte della formula. Il risultato è una formula molto più corta.

DIMOSTRAZIONE. Prima di tutto, diamo un algoritmo polinomiale in spazio che decide *TQBF*.

T = "Su input $\langle\phi\rangle$, una formula booleana completamente quantificata:

1. Se ϕ non contiene quantificatori, allora è un'espressione con valori costanti soltanto, quindi valuta ϕ e *accetta* se è vera; altrimenti, *rifiuta*.
2. Se ϕ è uguale a $\exists x\ \psi$, invoca ricorsivamente T su ψ, prima con 0 al posto di x e poi con 1 al posto di x. Se uno dei risultati è di accettazione, allora *accetta*; altrimenti, *rifiuta*.

3. Se ϕ è uguale a $\forall x\ \psi$, invoca ricorsivamente T su ψ, prima con 0 al posto di x e poi con 1 al posto di x. Se entrambi i risultati sono di accettazione, allora *accetta*; altrimenti, *rifiuta*."

L'algoritmo T ovviamente decide *TQBF*. Per analizzare la sua complessità di spazio, notiamo che la profondità della ricorsione è data al più dal numero di variabili. A ciascun livello dobbiamo memorizzare soltanto il valore di una variabile, quindi, lo spazio totale utilizzato è $O(m)$, dove m è il numero di variabili presenti in ϕ. Pertanto, T computa in spazio lineare.

Nel seguito mostriamo che *TQBF* è PSPACE-hard. Sia A un linguaggio deciso da una TM M in spazio n^k per qualche costante k. Diamo una riduzione di tempo polinomiale di A a *TQBF*.

La riduzione associa una stringa w ad una formula booleana quantificata ϕ che è vera se e solo se M accetta w. Per mostrare come costruire ϕ, risolviamo un problema più generale. Usando due collezioni di variabili denotate con c_1 e c_2, rappresentanti due configurazioni, ed un numero $t > 0$, costruiamo una formula $\phi_{c_1,c_2,t}$. Se assegniamo a c_1 e c_2 configurazioni reali, la formula è vera se e solo se M può andare da c_1 a c_2 in al più t passi. Allora possiamo definire ϕ come la formula $\phi_{c_{\text{start}},c_{\text{accept}},h}$, dove $h = 2^{df(n)}$ per una costante d, scelta in modo tale che M abbia non più di $2^{df(n)}$ configurazioni possibili su un input di lunghezza n. Sia $f(n) = n^k$. Per comodità, assumiamo che t sia una potenza di 2.

La formula codifica i contenuti delle celle di una configurazione come nella prova del teorema di Cook e Levin. Ciascuna cella ha diverse variabili ad essa associate, una per ogni simbolo del nastro e stato, corrispondenti alle possibili situazioni per la cella. Ciascuna configurazione ha n^k celle e, quindi, è codificata da $O(n^k)$ variabili.

Se $t = 1$, possiamo costruire facilmente $\phi_{c_1,c_2,t}$. Progettiamo la formula per dire che c_1 risulta uguale a c_2, oppure che c_2 segue da c_1 tramite un singolo passo di M. Esprimiamo l'uguaglianza scrivendo un'espressione booleana che afferma che ciascuna delle variabili che rappresentano c_1 contiene lo stesso valore booleano della variabile corrispondente che rappresenta c_2. Esprimiamo la seconda possibilità usando la tecnica presentata nelle prova del teorema di Cook e Levin. Cioè, possiamo esprimere che c_1 dà c_2 in un singolo passo di M scrivendo espressioni booleane che affermano che il contenuto di ciascuna tripla di celle di c_1 dà correttamente il contenuto della tripla di celle di c_2 corrispondente.

Se $t > 1$, costruiamo $\phi_{c_1,c_2,t}$ ricorsivamente. Per acquisire familiarità con la tecnica, ragioniamo su un'idea che non funziona del tutto, per poi sistemare i dettagli. Sia

$$\phi_{c_1,c_2,t} = \exists m_1 \left[\phi_{c_1,m_1,\frac{t}{2}} \wedge \phi_{m_1,c_2,\frac{t}{2}}\right].$$

Il simbolo m_1 rappresenta una configurazione di M. Scrivere $\exists m_1$ è un modo breve per $\exists x_1, \ldots, x_l$, dove $l = O(n^k)$ e $x_1, \ldots, x_l$ sono le variabili che

codificano m_1. Quindi, questa costruzione di $\phi_{c_1,c_2,t}$ dice che M può andare da c_1 a c_2 in al più t passi se esiste una configurazione intermedia m_1, per cui M può andare da c_1 a m_1 in al più $\frac{t}{2}$ passi e poi da m_1 a c_2 in al più $\frac{t}{2}$ passi. Successivamente costruiamo le due formule $\phi_{c_1,m_1,\frac{t}{2}}$ e $\phi_{m_1,c_2,\frac{t}{2}}$ ricorsivamente.

La formula $\phi_{c_1,c_2,t}$ assume il valore corretto; cioè, dà VERO ogni volta che M può andare da c_1 a c_2 in al più t passi. Tuttavia, è troppo grande. Ogni livello della ricorsione coinvolto nella costruzione dimezza t ma approssimativamente raddoppia la dimensione della formula. Pertanto, otteniamo alla fine una formula di dimensione circa t. Inizialmente $t = 2^{df(n)}$, e quindi il metodo produce una formula di dimensione esponenziale.

Per ridurre la dimensione della formula, utilizziamo il quantificatore $\forall$, in aggiunta al quantificatore $\exists$. Sia

$$\phi_{c_1,c_2,t} = \exists m_1 \, \forall (c_3,c_4) \in \{(c_1,m_1),(m_1,c_2)\} \; [\phi_{c_3,c_4,\frac{t}{2}}].$$

L'introduzione delle nuove variabili che rappresentano le configurazioni c_3 e c_4 ci permette di "racchiudere" le due sottoformule ricorsive in una singola sottoformula, allo stesso tempo preservando il significato originale. Scrivendo $\forall (c_3,c_4) \in \{(c_1,m_1),(m_1,c_2)\}$, indichiamo che le variabili che rappresentano le configurazioni c_3 e c_4 possono assumere i valori delle variabili di c_1 ed m_1 o di m_1 e c_2, rispettivamente, e che la formula risultante $\phi_{c_3,c_4,\frac{t}{2}}$ è vera in entrambi i casi. Possiamo sostituire il costrutto $\forall x \in \{y,z\} \; [\ldots]$ con il costrutto equivalente $\forall x \; [(x = y \vee x = z) \rightarrow \ldots]$ per ottenere una formula booleana sintatticamente corretta. Ricordiamo che in Sezione 0.2, abbiamo mostrato che l'implicazione booleana ($\rightarrow$) e l'uguaglianza booleana ($=$) possono essere espresse in termini di AND e NOT. Qui, per chiarezza, usiamo il simbolo $=$ per l'uguaglianza booleana invece del simbolo equivalente $\leftrightarrow$ usato nella Sezione 0.2.

Per calcolare la dimensione della formula $\phi_{c_{\text{start}},c_{\text{accept}},h}$, dove $h = 2^{df(n)}$, notiamo che ciascun livello della ricorsione aggiunge una porzione della formula che è lineare nella dimensione delle configurazioni, ed è perciò di dimensione $O(f(n))$. Il numero di livelli della ricorsione è $\log(2^{df(n)})$, ovvero $O(f(n))$. Pertanto, la dimensione della formula risultante è $O(f^2(n))$.

Strategie vincenti per giochi

Per gli scopi di questa sezione, un ***gioco*** è vagamente definito come una competizione in cui le parti che si contrappongono cercano di ottenere un obiettivo in accordo a regole predefinite. I giochi sono presenti in molte forme, dai giochi da tavolo come gli scacchi, ai giochi economici e di guerra, che modellano conflitti sociali e corporativi.

I giochi sono strettamente legati ai quantificatori. Un enunciato quantificato ha un gioco corrispondente; viceversa, un gioco spesso ha un enunciato quantificato corrispondente. Queste corrispondenze sono utili in molti modi. Per un verso, esprimere un enunciato matematico che usa quantificatori in termini di un gioco corrispondente, può fornire intuizioni sul significato dell'enunciato. Per l'altro, esprimere un gioco in termini di un enunciato quantificato, aiuta a comprendere la complessità del gioco.

Per illustrare la corrispondenza tra giochi e quantificatori, soffermiamoci su un gioco artificiale, detto il ***gioco della formula.***

Sia $\phi = \exists x_1 \, \forall x_2 \, \exists x_3 \cdots \mathrm{Q} x_k \; [\psi]$ una formula Booleana quantificata in forma normale prenex. In questo contesto, Q rappresenta un quantificatore $\forall$ o $\exists$. Associamo un gioco a ϕ come segue. Due giocatori, chiamati Giocatore A e Giocatore E, a turno selezionano i valori delle variabili $x_1, \ldots, x_k$. Il Giocatore A seleziona valori per le variabili che sono legate ai quantificatori $\forall$, ed il Giocatore E seleziona valori per le variabili che sono legate ai quantificatori $\exists$. L'ordine del gioco è lo stesso ordine dei quantificatori all'inizio della formula. Alla fine del gioco, usiamo i valori che i giocatori hanno selezionato per le variabili e diciamo che ha vinto il Giocatore E se ψ, la parte della formula in cui i quantificatori sono stati rimossi, dà VERO. Il Giocatore A ha vinto se ψ dà FALSO.

ESEMPIO 8.10

Sia ϕ_1 la formula

$$\exists x_1 \, \forall x_2 \, \exists x_3 \; \big[(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})\big].$$

Nel gioco della formula per ϕ_1, il Giocatore E sceglie il valore di x_1, successivamente il Giocatore A sceglie il valore di x_2, ed infine il Giocatore E sceglie il valore di x_3.

Per fornire un esempio di esecuzione del gioco, cominciamo con il rappresentare, come al solito, i valori booleani VERO con 1 e FALSO con 0. Supponiamo che il Giocatore E scelga $x_1 = 1$, quindi il Giocatore A scelga $x_2 = 0$, ed infine il Giocatore E scelga $x_3 = 1$. Con questi valori di x_1, x_2, ed x_3, la sottoformula

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

vale 1, e quindi il Giocatore E ha vinto il gioco. In realtà, il Giocatore E può vincere sempre questo gioco selezionando $x_1 = 1$ e, successivamente, selezionando x_3 come il complemento del valore che il Giocatore A seleziona per x_2. Diciamo che il Giocatore E ha una ***strategia vincente*** per questo gioco. Un giocatore ha una strategia vincente per un gioco se il giocatore vince quando entrambe le parti giocano in modo ottimale.

A questo punto, cambiamo leggermente la formula per ottenere un gioco in cui il Giocatore A ha una strategia vincente. Sia ϕ_2 la formula

$$\exists x_1 \, \forall x_2 \, \exists x_3 \; \big[(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})\big].$$

Il Giocatore A adesso ha una strategia vincente perché, indipendentemente da ciò che il Giocatore E sceglie per x_1, il Giocatore A può scegliere $x_2 = 0$, falsificando con ciò la parte della formula presente dopo i quantificatori, qualunque sia l'ultima mossa del Giocatore E.

Di seguito consideriamo il problema di determinare quale giocatore ha una strategia vincente nel gioco della formula associato ad una specifica formula. Sia

$$FORMULA\text{-}GAME = \{\langle\phi\rangle \mid \text{Il Giocatore E ha una strategia vincente nel gioco della formula associato a } \phi\}.$$

TEOREMA 8.11

FORMULA-GAME è PSPACE-completo.

IDEA. *FORMULA-GAME* è PSPACE-completo per una ragione semplice. Coincide con *TQBF*. Per vedere che $FORMULA\text{-}GAME = TQBF$, si noti che una formula è VERA esattamente quando il Giocatore E ha una strategia vincente nel gioco della formula associato. I due enunciati rappresentano modi diversi per dire la stessa cosa.

DIMOSTRAZIONE. La formula $\phi = \exists x_1 \forall x_2 \exists x_3 \cdots [\psi]$ è VERA quando esiste qualche assegnamento per x_1 tale che, per qualsiasi assegnamento per x_2, esiste un assegnamento per x_3 tale che, e così via ..., dove ψ è VERA rispetto agli assegnamenti delle variabili. Analogamente, il Giocatore E ha una strategia vincente nel gioco associato a ϕ quando il Giocatore E può produrre un assegnamento per x_1 tale che, per qualsiasi assegnamento per x_2, il Giocatore E può produrre un assegnamento per x_3 tale che, e così via ..., dove ψ è VERA rispetto agli assegnamenti delle variabili.

Lo stesso ragionamento si applica quando la formula non alterna quantificatori esistenziali ed universali. Se ϕ ha la forma $\forall x_1, x_2, x_3 \exists x_4, x_5 \forall x_6 [\psi]$, il Giocatore A farebbe le prime tre mosse nel gioco della formula per assegnare valori a x_1, x_2, e x_3; successivamente il Giocatore E farebbe due mosse per assegnare valori a x_4 e x_5; ed infine il Giocatore A assegnerebbe un valore a x_6.

Pertanto, $\phi \in TQBF$ esattamente quando $\phi \in FORMULA\text{-}GAME$, ed il teorema discende dal Teorema 8.9.

Gioco Geografia generalizzato

Ora che sappiamo che il gioco della formula è PSPACE-completo, possiamo stabilire la PSPACE-completezza o PSPACE-difficoltà di altri giochi più facilmente. Cominceremo con una generalizzazione del gioco Geografia e successivamente discuteremo giochi quali gli scacchi, la dama e GO.

Geografia è un gioco per ragazzi in cui i giocatori nominano a turno città di qualsiasi parte del mondo. Ciascuna città scelta deve iniziare con la stessa lettera con cui finisce il nome della città precedente. La ripetizione non è ammessa.

Il gioco parte da una città designata iniziale e finisce quando qualche giocatore perde perché non è in grado di continuare. Per esempio, se il gioco parte con Peoria, allora Amherst può seguire legittimamente (perché Peoria finisce con la lettera *a*, ed Amherst comincia con la lettera *a*), quindi Tucson, poi Nashua, e così via fino a quando uno dei giocatori non riesce a proseguire e di conseguenza perde.

Possiamo modellare questo gioco con un grafo diretto i cui nodi sono le città del mondo. Disegnamo una freccia da una città ad un'altra se la prima può portare alla seconda in accordo alle regole del gioco. In altre parole, il grafo contiene un arco da una città X verso una città Y se la città X finisce con la stessa lettera con cui comincia la città Y. Illustriamo una porzione del grafo di Geografia in Figura 8.12.

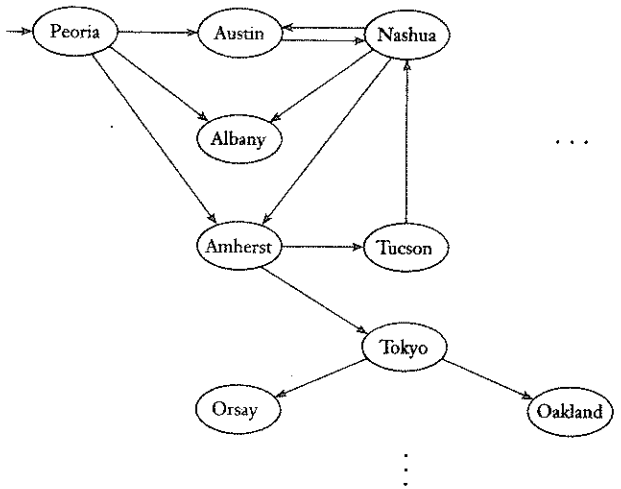


FIGURA 8.12
Porzione del grafo rappresentante il gioco Geografia

Quando le regole di Geografia vengono interpretate tramite questa rappresentazione grafica, un giocatore inizia scegliendo il nodo designato come nodo di partenza, e poi i giocatori, a turno, scelgono nodi che formano un cammino semplice nel grafo. La condizione che il cammino sia sempli-

ce (i.e., non usi nessun nodo più di una volta) corrisponde alla condizione che una città non può esser ripetuta. Il primo giocatore che non riesce ad estendere il cammino perde il gioco.

In ***Geografia generalizzato***, prendiamo un grafo diretto arbitrario con un nodo designato come nodo iniziale invece del grafo associato alle città reali. Per esempio, il grafo che segue è un esempio di Geografia generalizzato.

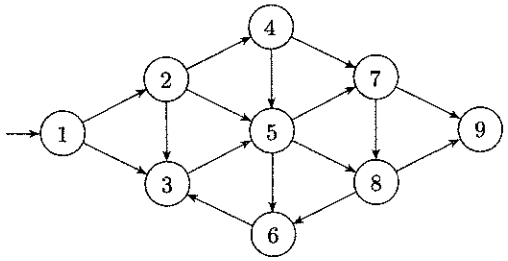


FIGURA 8.13
Un esempio del gioco Geografia generalizzato

Supponiamo che il Giocatore I sia quello che muove per primo e il Giocatore II per secondo. In questo esempio, il Giocatore I ha una strategia vincente. Il Giocatore I parte dal nodo 1, designato come nodo iniziale. Il nodo 1 punta soltanto ai nodi 2 e 3, quindi la prima mossa del Giocatore I deve essere una di queste due scelte. Supponiamo scelga 3. Adesso il Giocatore II deve muoversi, ma il nodo 3 punta soltanto al nodo 5, pertanto è costretto a scegliere il nodo 5. Successivamente, il Giocatore I sceglie 6, tra le scelte 6, 7, e 8. Ora il Giocatore II deve giocare dal nodo 6, che punta soltanto al nodo 3, e 3 è stato giocato in precedenza. Il Giocatore II non può muovere e, pertanto, il Giocatore I vince.

Se modifichiamo l'esempio invertendo la direzione dell'arco tra i nodi 3 e 6, il Giocatore II ha una strategia vincente. Si vede? Se il Giocatore I inizia come prima con il nodo 3, il Giocatore II risponde con 6 e vince immediatamente, quindi l'unica speranza del Giocatore I è iniziare con 2. In tal caso, tuttavia, il Giocatore II risponde con 4. Se a questo punto il Giocatore I prende 5, il Giocatore II vince con 6. Se il Giocatore I prende 7, il Giocatore II vince con 9. Indipendentemente da cosa fa il Giocatore I, il Giocatore II può trovare un modo per vincere, pertanto il Giocatore II ha una strategia vincente.

Il problema di determinare quale giocatore ha una strategia vincente nel gioco Geografia generalizzato è PSPACE-completo. Sia

$$GG = \{\langle G, b\rangle \mid \text{Il Giocatore I ha una strategia vincente per il gioco Geografia generalizzato giocato sul grafo } G \text{ a partire dal nodo } b\}.$$

TEOREMA 8.14

GG è PSPACE-completo.

IDEA. Un algoritmo ricorsivo simile a quello usato per *TQBF* nel Teorema 8.9 determina quale giocatore ha una strategia vincente. Questo algoritmo computa in spazio polinomiale e, quindi, $GG \in$ PSPACE.

Per provare che *GG* è PSPACE-hard, forniamo una riduzione in tempo polinomiale da *FORMULA-GAME* a *GG*. La riduzione converte un gioco della formula in un grafo di Geografia generalizzato, in modo tale che l'agire sul grafo imiti l'agire nel gioco della formula. In effetti, i giocatori nel gioco di geografia generalizzato stanno realmente giocando una forma codificata del gioco della formula.

DIMOSTRAZIONE. L'algoritmo che segue decide se il Giocatore I ha una strategia vincente in istanze del gioco di geografia generalizzato; in altre parole, decide *GG*. Mostreremo che computa in spazio polinomiale.

M = "Su input $\langle G, b\rangle$, dove G è un grafo diretto e b è un nodo di G:

1. Se b ha grado uscente 0, *rifiuta* poiché il Giocatore I perde immediatamente.
2. Rimuovi il nodo b e tutti gli archi connessi per ottenere un nuovo grafo G'.
3. Per ciascuno dei nodi $b_1, b_2, \ldots, b_k$ a cui originariamente b puntava, invoca ricorsivamente M su $\langle G', b_i\rangle$.
4. Se in tutte queste invocazioni M accetta, il Giocatore II ha una strategia vincente nel gioco originale, pertanto *rifiuta*. Altrimenti, il Giocatore II non ha una strategia vincente, e quindi il Giocatore I deve averla; perciò, *accetta*."

Il solo spazio richiesto da questo algoritmo serve per memorizzare la pila per la ricorsione. Ciascun livello della ricorsione aggiunge un singolo nodo alla pila, ed al più occorrono m livelli, dove m è il numero di nodi in G. Pertanto, l'algoritmo computa in spazio lineare.

Per stabilire che *GG* è PSPACE-hard, facciamo vedere che *FORMULA-GAME* è riducibile in tempo polinomiale a *GG*. La riduzione associa alla formula

$$\phi = \exists x_1 \forall x_2 \exists x_3 \cdots \mathrm{Q} x_k \, [\psi]$$

un'istanza $\langle G, b\rangle$ di Geografia generalizzato. Assumiamo per semplicità che i quantificatori di ϕ comincino e finiscano con $\exists$, e che si alternino strettamente tra $\exists$ e $\forall$. Una formula non conforme all'assunzione può essere convertita in una conforme leggermente più grande, aggiungendo quantificatori extra che legano variabili altrimenti inutilizzate o fittizie. Supponiamo pure che ψ sia in forma normale congiuntiva (vedi il Problema 8.28).

La riduzione costruisce un gioco di geografia su un grafo G dove una giocata ottima simula una giocata ottima nel gioco della formula su ϕ. Il Giocatore I nel gioco di geografia assume il ruolo del Giocatore E nel gioco della formula, ed il Giocatore II assume il ruolo del Giocatore A.

La struttura del grafo G è mostrata parzialmente nella figura seguente. Il gioco inizia dal nodo b, che compare in alto sul lato sinistro di G. Al di sotto di b, compare una sequenza di strutture romboidali, una per ciascuna variabile di ϕ. Prima di arrivare sul lato destro di G, vediamo come il gioco procede sul lato sinistro.

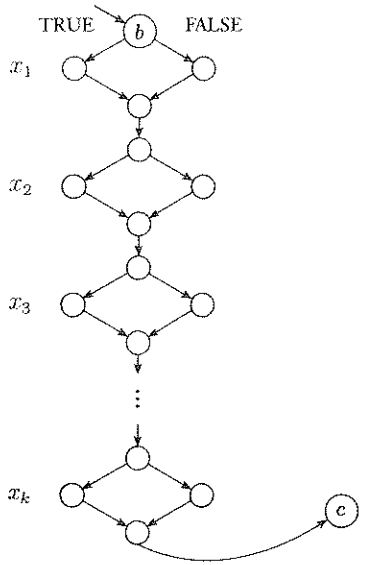


FIGURA 8.15
Struttura parziale del gioco di geografia che simula il gioco della formula

Il gioco comincia da b. Il Giocatore I deve selezionare uno dei due archi uscenti da b. Questi archi corrispondono alle scelte possibili per il Giocatore E all'inizio del gioco della formula. La scelta sul lato sinistro per il Giocatore I corrisponde a VERO per il Giocatore E nel gioco della formula e la scelta sul lato destro a FALSO. Dopo che il Giocatore I ha selezionato uno di questi archi - diciamo, quello sul lato sinistro - il Giocatore II muove. Solo un arco uscente è presente, quindi la mossa è forzata. Analogamente, la mossa successiva del Giocatore I è forzata e il gioco continua dalla parte superiore del secondo rombo. Ora di nuovo due archi sono presenti, ma il Giocatore II ottiene il turno. Questa scelta corrisponde alla prima mossa per il Giocatore A nel gioco della formula. Il gioco continua in questo modo, ed i Giocatori I e II scelgono un cammino verso destra o verso sinistra attraverso ciascuno dei rombi.

Dopo che il gioco ha attraversato tutti i rombi, la testa del cammino si trova sul nodo in basso dell'ultimo rombo, ed è il turno del Giocatore I perché abbiamo assunto che l'ultimo quantificatore è $\exists$. La mossa successiva del Giocatore I è forzata. Successivamente, si trovano sul nodo c in Figura 8.15 ed il Giocatore II effettua la prossima mossa.

Questo punto nel gioco di geografia corrisponde alla fine del gioco nel gioco della formula. Il cammino scelto attraverso i rombi corrisponde ad un assegnamento alle variabili di ϕ. Rispetto a questo assegnamento, se ψ è VERO, il Giocatore E vince il gioco della formula; se ψ è FALSO, vince il Giocatore A. La struttura sul lato destro della figura seguente garantisce che il Giocatore I può vincere se il Giocatore E ha vinto, e che il Giocatore II può vincere se il Giocatore A ha vinto.

Sul nodo c, il Giocatore II può scegliere un nodo corrispondente ad una delle clausole di ψ. Successivamente, il Giocatore I può scegliere un nodo corrispondente ad un letterale in quella clausola. I nodi corrispondenti a letterali non in forma negata sono connessi ai lati sinistri (VERO) del rombo per le variabili associate, e analogamente per i letterali in forma negata e i lati destri (FALSO), come mostrato in Figura 8.16.

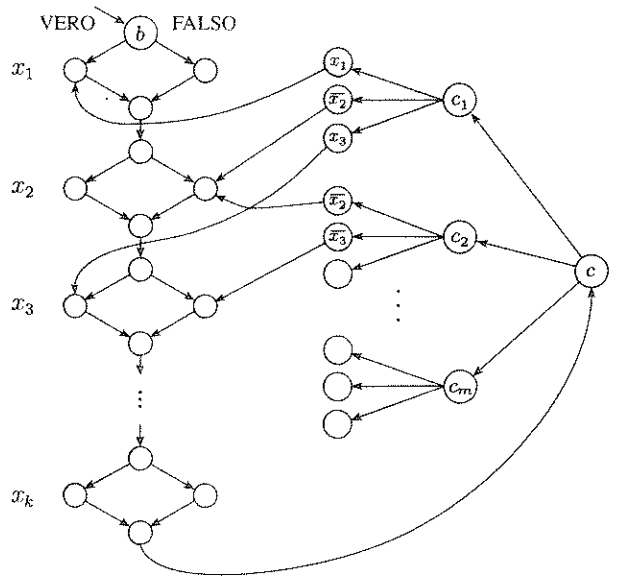


FIGURA 8.16
Struttura completa del gioco di geografia che simula il gioco della formula, dove
$\phi = \exists x_1 \forall x_2 \cdots \exists x_k\ [(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3} \vee \cdots) \wedge \cdots \wedge (\quad)]$

Se ψ è FALSO, il Giocatore II può vincere selezionando la clausola non soddisfatta. Qualsiasi letterale che il Giocatore I può successivamente scegliere è FALSO ed è connesso al lato del rombo che non è stato ancora giocato. Quindi, il Giocatore II può giocare il nodo nel rombo, ma poi il Giocatore I è impossibilitato a muoversi e perde. Se ψ è VERO, qualsiasi clausola che il Giocatore II sceglie contiene un letterale VERO. Il Giocatore I seleziona quel letterale dopo la mossa del Giocatore II. Poiché il letterale è VERO, è connesso al lato del rombo che è stato già giocato, pertanto il Giocatore II non può muoversi e perde.

Nel Teorema 8.14 abbiamo mostrato che non esistono algoritmi di tempo polinomiale per giocare in modo ottimale il gioco di geografia generalizzato a meno che P = PSPACE. Ci piacerebbe provare un risultato analogo per quanto riguarda la difficoltà di giocare in modo ottimale i giochi da tavolo come gli scacchi, ma sorge un ostacolo. Possono presentarsi soltanto un numero finito di posizioni di gioco differenti su una scacchiera standard 8×8. In linea di principio tutte queste posizioni potrebbero esser collocate in una tabella, insieme alla mossa migliore in ogni posizione. La tabella sarebbe troppo grande per entrare nella nostra galassia ma, essendo finita, potrebbe venire memorizzata nel controllo di una macchina di Turing (o addirittura in quella di un automa finito!). Così, la macchina sarebbe in grado di giocare in modo ottimale in tempo lineare, usando la ricerca nella tabella. Forse in futuro saranno sviluppati metodi che permetteranno di quantificare la complessità di problemi finiti. Ma i metodi attuali sono asintotici e pertanto si applicano solo al tasso di crescita della complessità al crescere della dimensione del problema - ma a nessuna dimensione fissata. Ciò nondimeno, possiamo fornire una qualche evidenza per la difficoltà di calcolare strategie di gioco ottimali per molti giochi da tavolo, generalizzandoli a tavoli di dimensioni $n \times n$. È stato dimostrato che queste generalizzazioni degli scacchi, della dama e di GO sono PSPACE-difficili o difficili anche per classi di complessità più ampie, a seconda dei dettagli della generalizzazione.

8.4 LE CLASSI L ED NL

Fino ad ora abbiamo considerato soltanto limitazioni alle complessità di spazio e di tempo che risultano almeno lineari - cioè, limitazioni dove $f(n)$ è almeno n. Ne esaminiamo ora di più piccole, ovvero limitazioni di spazio *quasilineari*. Nella complessità di tempo limitazioni quasilineari sono insufficienti per leggere l'input intero, e quindi non le prendiamo in considerazione. Con una complessità di spazio quasilineare, la macchina è in

grado di leggere l'intero input ma non ha abbastanza spazio per memorizzarlo. Per trattare questa situazione in maniera significativa, dobbiamo modificare il nostro modello computazionale.

Introduciamo pertanto macchine di Turing con due nastri: un nastro di input a sola lettura, e un nastro di lavoro di lettura/scrittura. Sul nastro a sola lettura, la testina può leggere i simboli ma non può cambiarli. Doteremo la macchina di un modo per capire quando la testina è sull'estremità sinistra o su quella destra dell'input. La testina deve rimanere sulla porzione del nastro che contiene l'input. Il nastro di lavoro può essere letto o scritto come al solito. Soltanto le celle esaminate sul nastro di lavoro contribuiscono alla complessità di spazio di questo tipo di macchina di Turing.

Si pensi ad un nastro di input a sola lettura come ad un CD-ROM, un dispositivo usato per l'input in molti personal computer. Spesso il CD-ROM contiene molti più dati di quanti il computer ne possa memorizzare nella propria memoria principale. Algoritmi con spazio quasilineare permettono al computer di manipolare i dati senza memorizzarli totalmente in memoria principale.

Per limitazioni di spazio che sono almeno lineari, il modello di TM a due nastri è equivalente al modello standard con un singolo nastro (vedi Esercizio 8.1). Per limitazioni quasilineari, usiamo soltanto il modello a due nastri.

DEFINIZIONE **8.17**

L è la classe di linguaggi che sono decidibili in spazio logaritmico da una macchina di Turing deterministica. In altre parole,

$$\mathrm{L} = \mathrm{SPACE}(\log n).$$

NL è la classe di linguaggi che sono decidibili in spazio logaritmico da una macchina di Turing non deterministica. In altre parole,

$$\mathrm{NL} = \mathrm{NSPACE}(\log n).$$

Focalizziamo la nostra attenzione su spazio $\log n$ invece di, diciamo, $\sqrt{n}$ oppure spazio $\log^2 n$, per molteplici ragioni che sono simili a quelle alla base delle nostre scelte per le limitazioni di spazio e di tempo polinomiale. Uno spazio logaritmico è grande abbastanza per risolvere un buon numero di problemi computazionali interessanti, e mantiene proprietà matematiche attrattive quali la robustezza anche quando il modello di macchina ed il metodo di codifica dell'input cambiano. I puntatori a parti dell'input possono essere rappresentati con spazio logaritmico, quindi un modo per pensare alla potenza di algoritmi che usano spazio logaritmico è quello di pensare al potere di un numero fissato di puntatori all'input.

ESEMPIO 8.18

Il linguaggio $A = \{0^k 1^k | \, k \geq 0\}$ è un elemento di L. In Sezione 7.1, a pagina 293, abbiamo descritto una macchina di Turing che decide A procedendo a zig-zag avanti e indietro attraverso l'input, marcando gli 0 e gli 1 non appena vengono accoppiati. L'algoritmo in questione usa spazio lineare per memorizzare quali posizioni sono state marcate, ma può essere modificato per far sì che usi soltanto spazio logaritmico.

La TM con spazio logaritmico per A non può marcare gli 0 e gli 1 che sono stati accoppiati sul nastro di input perché il nastro di input è a sola lettura. Invece, la macchina conta il numero di 0 e, separatamente, il numero di 1 in binario sul nastro di lavoro. Il solo spazio richiesto è quello utilizzato per memorizzare i due contatori. In binario, ciascun contatore usa soltanto spazio logaritmico e quindi l'algoritmo computa in spazio $O(\log n)$. Pertanto, $A \in$ L.

ESEMPIO 8.19

Si ricordi il linguaggio

$PATH = \{(G, s, t) | G$ è un grafo diretto che ha un cammino diretto da s a $t\}$

definito in Sezione 7.2. Il Teorema 7.14 mostra che $PATH$ appartiene a P, ma che l'algoritmo dato usa spazio lineare. Non sappiamo se $PATH$ possa essere risolto deterministicamente con spazio logaritmico, ma conosciamo un algoritmo non deterministico per $PATH$ che usa spazio logaritmico.

La macchina di Turing non deterministica di spazio logaritmico che decide $PATH$ opera partendo dal nodo s e provando ad individuare non deterministicamente i nodi di un cammino da s a t. La macchina memorizza solo la posizione del nodo corrente ad ogni passo sul nastro di lavoro, non l'intero cammino (che farebbe eccedere il requisito di spazio logaritmico). La macchina seleziona non deterministicamente il prossimo nodo tra quelli puntati dal nodo corrente. Ripete questa azione fino a quando raggiunge il nodo t e restituisce *accetta*, oppure fino a quando è rimasta in esecuzione per m passi e restituisce *rifiuta*, dove m è il numero di nodi nel grafo. Pertanto, $PATH$ appartiene a NL.

La nostra affermazione precedente che ogni macchina di Turing limitata in spazio da $f(n)$ computa anche in tempo $2^{O(f(n))}$ non vale più per limitazioni molto piccole dello spazio. Per esempio, una macchina di Turing che usa spazio $O(1)$ (i.e., costante) può computare per n passi. Per ottenere una limitazione al tempo di esecuzione che si applichi a qualsiasi limitazione $f(n)$ allo spazio, diamo la seguente definizione.

DEFINIZIONE 8.20

Se M è una macchina di Turing che ha un nastro di input separato di sola lettura e w è un input, una ***configurazione di M su w*** consiste in un assegnamento dello stato, del nastro di lavoro, e delle posizioni delle due testine sui nastri. L' input w non è parte della configurazione di M su w.

Se M computa in spazio $f(n)$ e w è un input di lunghezza n, il numero di configurazioni di M su w è $n2^{O(f(n))}$. Per spiegare questo risultato, supponiamo che M abbia c stati e g simboli di nastro. Il numero di stringhe possibili sul nastro di lavoro è $g^{f(n)}$. La testina di input può stare in una delle n posizioni, e la testina del nastro di lavoro può stare in $f(n)$ posizioni. Pertanto, il numero totale di configurazioni di M su w, che rappresenta un limite superiore al tempo di esecuzione di M su w, è $cnf(n)g^{f(n)}$, o anche $n2^{O(f(n))}$.

Focalizzeremo la nostra attenzione quasi esclusivamente su limitazioni di spazio $f(n)$ che risultino almeno $\log n$. La nostra affermazione precedente che la complessità di tempo di una macchina è al più esponenziale nella sua complessità di spazio resta vera per queste limitazioni perché $n2^{O(f(n))}$ è $2^{O(f(n))}$ quando $f(n) \geq \log n$.

Si ricordi che il teorema di Savitch mostra che è possibile convertire TM non deterministiche in TM deterministiche con un incremento della complessità di spazio $f(n)$ soltanto quadratica, posto che $f(n) \geq n$. Possiamo estendere il teorema di Savitch affinché valga per limitazioni di spazio quasilineari con $f(n) \geq \log n$. La prova è identica all'originale fornita a pagina 358, eccetto che usiamo macchine di Turing con un nastro di input a sola lettura; ed invece di far riferimento alle configurazioni di N, facciamo riferimento alle configurazioni di N su w. La memorizzazione di una configurazione di N su w usa spazio $\log(n2^{O(f(n))}) = \log n + O(f(n))$. Se $f(n) \geq \log n$, la memoria usata è $O(f(n))$ ed il resto della dimostrazione rimane immutato.

8.5 NL-COMPLETEZZA

Come menzionato nell'Esempio 8.19, si sa che il problema $PATH$ è in NL ma non si sa se appartiene a L. Si ritiene che $PATH$ non appartenga a L, ma non siamo in grado di provare questa congettura. In realtà, non conosciamo nessun problema in NL che si possa dimostrare non essere in L. Analogamente alla domanda se P = NP, esiste la domanda se L = NL.

Come passo verso la risoluzione della questione L diverso da NL possiamo mostrare alcuni linguaggi che sono NL-completi. Come per i linguaggi completi per altre classi di complessità, i linguaggi NL-completi sono esempi di linguaggi che sono, in un certo senso, i linguaggi più difficili in NL. Se L e NL sono differenti, tutti i linguaggi NL-completi non appartengono a L.

Similmente a quanto fatto per le definizioni precedenti di completezza, definiamo un linguaggio NL-completo se il linguaggio appartiene ad NL ed ogni altro linguaggio in NL è riducibile ad esso. Tuttavia, in questo caso non usiamo la riducibilità in tempo polinomiale perché, come vedremo, tutti i problemi in NL sono risolvibili in tempo polinomiale. Pertanto, ogni coppia di problemi in NL eccetto $\emptyset$ e Σ^* sono riducibili in tempo polinomiale l'uno all'altro (vedi la discussione sulla riducibilità in tempo polinomiale nella definizione di PSPACE-completezza a pagina 362). Quindi, la riducibilità in tempo polinomiale è troppo forte per differenziare l'uno dall'altro problemi in NL . Invece, utilizzeremo un nuovo tipo di riducibilità chiamata *riducibilità in spazio logaritmico.*

DEFINIZIONE **8.21**

Un ***trasduttore di spazio logaritmico*** è una macchina di Turing con un nastro di input di sola lettura, un nastro di output di sola scrittura, ed un nastro di lavoro su cui si può leggere e scrivere. La testina del nastro di output non si può muovere verso sinistra, quindi non può leggere ciò che ha scritto. Il nastro di lavoro può contenere $O(\log n)$ simboli. Un trasduttore di spazio logaritmico M computa una funzione $f\colon \Sigma^* \longrightarrow \Sigma^*$, dove $f(w)$ è la stringa che rimane sul nastro di output dopo che M si è arrestata, dopo aver iniziato la computazione con w sul proprio nastro di input. Diremo che f è una ***funzione computabile in spazio logaritmico***. Il linguaggio A è ***riducibile in spazio logaritmico*** al linguaggio B, denotato da $A \leq_{\mathrm{L}} B$, se A è riducibile mediante funzione a B attraverso una funzione f computabile in spazio logaritmico.

A questo punto siamo pronti per definire la NL-completezza.

DEFINIZIONE **8.22**

Un linguaggio B è ***NL-completo*** se

1. $B \in$ NL, e
2. ogni A in NL è riducibile in spazio logaritmico a B.

Se un linguaggio è riducibile in spazio logaritmico ad un altro linguaggio che è noto appartere a L, anche il linguaggio originale appartiene a L, come dimostra il teorema seguente.

TEOREMA 8.23

Se $A \leq_{\mathrm{L}} B$ e $B \in$ L, allora $A \in$ L.

DIMOSTRAZIONE. Un approccio invitante alla dimostrazione di questo teorema consiste nel seguire il modello presentato nel Teorema 7.31, il risultato analogo per la riducibilità in tempo polinomiale. In quell'approccio un algoritmo di spazio logaritmico per A prima associa $f(w)$ al proprio input w, utilizzando la riduzione in spazio logaritmico f, e poi applica l'algoritmo di spazio logaritmico per B. Tuttavia, lo spazio di memoria richiesto per $f(w)$ potrebbe essere troppo grande per rientrare nel limite di spazio logaritmico, quindi dobbiamo modificare questo approccio.

Invece, la macchina M_A di A computa singoli simboli di $f(w)$ come richiesto dalla macchina M_B di B. Nella simulazione M_A tiene traccia di dove si troverebbe la testina di lettura di M_B su $f(w)$. Ogni volta che M_B si muove, M_A fa ripartire dall'inizio la computazione di f su w e ignora completamente l'output ad eccezione della locazione desiderata di $f(w)$. Procedere in questo modo può richiedere occasionalmente il ricalcolo di parti di $f(w)$ e, pertanto, risulta inefficiente in termini di complessità di tempo. Il vantaggio di questo metodo è che, in ogni istante, soltanto un singolo simbolo di $f(w)$ deve essere memorizzato, barattando in effetti tempo per spazio.

COROLLARIO 8.24

Se un qualsiasi linguaggio NL-completo è in L, allora L = NL.

Ricerca in grafi

TEOREMA 8.25

PATH è NL-completo.

IDEA. L'Esempio 8.19 mostra che *PATH* appartiene a NL, quindi dobbiamo dimostrare soltanto che *PATH* è NL-difficile. In altre parole, dobbiamo dimostrare che qualsiasi linguaggio A appartenente a NL è riducibile in spazio logaritmico a *PATH*.

L'idea che sta dietro alla riduzione in spazio logaritmico di A a *PATH* è costruire un grafo che rappresenti la computazione per A di una macchina di Turing non deterministica con spazio logaritmico. La riduzione associa

a una stringa w un grafo i cui nodi corrispondono alle configurazioni della NTM sull'input w. Un nodo punta ad un secondo nodo se la prima configurazione corrispondente può produrre la seconda configurazione, attraverso un singolo passo della NTM. Pertanto, la macchina accetta w ogni volta che un cammino dal nodo corrispondente alla configurazione iniziale porta al nodo corrispondente alla configurazione di accettazione.

DIMOSTRAZIONE. Faremo vedere come fornire una riduzione in spazio logaritmico di un qualsiasi linguaggio A appartenente a NL a $PATH$. Supponiamo che la NTM M decida A in spazio $O(\log n)$. Dato un input w, costruiamo $\langle G, s, t\rangle$ in spazio logaritmico, dove G è un grafo diretto che contiene un cammino da s a t se e solo se M accetta w.

I nodi di G sono le configurazioni di M su w. Per le configurazioni c_1 e c_2 di M su w, la coppia (c_1, c_2) è un arco di G se c_2 è una delle successive configurazioni possibili di M partendo da c_1. Più precisamente, se la funzione di transizione di M indica che lo stato di c_1 insieme ai simboli di nastro rispetto alle testine di input e del nastro di lavoro, possono produrre lo stato successivo e movimenti delle testine tali da trasformare c_1 in c_2, allora (c_1, c_2) è un arco di G. Il nodo s rappresenta la configurazione iniziale di M su w. La macchina M viene modificata per avere un'unica configurazione di accettazione, e stabiliamo che questa configurazione è rappresentata dal nodo t.

Questa associazione riduce A a $PATH$ perché ogni volta che M accetta il proprio input, qualche diramazione della sua computazione accetta, e ciò corrisponde ad un cammino dalla configurazione iniziale s alla configurazione di accettazione t in G. Viceversa, se esiste un cammino da s a t in G, una diramazione della computazione è accettante quando M computa su input w, ed M accetta w.

Per dimostrare che la riduzione opera in spazio logaritmico, diamo un trasduttore di spazio logaritmico che dà in output $\langle G, s, t\rangle$ su input w. Descriviamo G elencando i suoi nodi ed i suoi archi. Elencare i nodi è facile perché ciascun nodo è una configurazione di M su w e può essere rappresentato in spazio $c \log n$ per qualche costante c. Il trasduttore procede sequenzialmente attraverso tutte le possibili stringhe di lunghezza $c \log n$, verifica che ciascuna di esse sia una configurazione legale di M su w, e dà in output quelle che superano la verifica. Il trasduttore elenca gli archi in modo simile. Una quantità di spazio logaritmica è sufficiente per verificare che una configurazione c_1 di M su w può produrre una configurazione c_2 perché il trasduttore deve esaminare soltanto i contenuti correnti nelle locazioni sotto le testine in c_1 per stabilire che la funzione di transizione di M darebbe come risultato la configurazione c_2. Il trasduttore tenta a sua volta tutte le coppie (c_1, c_2) per trovare quali sono qualificate ad essere archi di G. Quelle che lo sono vengono aggiunte al nastro di output.

Una conseguenza immediata del Teorema 8.25 è il corollario seguente, che afferma che NL è un sottoinsieme di P.

COROLLARIO 8.26

NL $\subseteq$ P.

DIMOSTRAZIONE. Il Teorema 8.25 mostra che ogni linguaggio in NL è riducibile in spazio logaritmico a $PATH$. Ricordiamo che una macchina di Turing che usa spazio $f(n)$ computa in tempo $n\,2^{O(f(n))}$, quindi un trasduttore che computa in spazio logaritmico computa anche in tempo polinomiale. Pertanto, ogni linguaggio in NL è riducibile in tempo polinomiale a $PATH$, che a sua volta appartiene a P, per il Teorema 7.14. Sappiamo che qualsiasi linguaggio che è polinomialmente riducibile ad un linguaggio in P è anche esso in P, e quindi la dimostrazione è completa.

Sebbene la riducibilità in spazio logaritmico sembri essere altamente restrittiva, è adeguata per molte riduzioni in teoria della complessità perché queste sono solitamente computazionalmente semplici. Per esempio, nel Teorema 8.9 abbiamo dimostrato che qualsiasi problema in PSPACE è riducibile in tempo polinomiale a $TQBF$. Le formule altamente ripetitive che queste riduzioni producono possono essere calcolate usando soltanto spazio logaritmico e, pertanto, possiamo concludere che $TQBF$ è PSPACE-completo rispetto alla riducibilità in spazio logaritmico. Questa conclusione è importante perché il Corollario 9.6 dimostra che NL $\subsetneq$ PSPACE. Detta separazione e la riducibilità in spazio logaritmico implicano che $TQBF \notin$ NL.

8.6
NL COINCIDE CON CONL

Questa sezione contiene uno dei risultati noti più sorprendenti in merito alle relazioni tra classi di complessità. Generalmente si ritiene che le classi NP e coNP siano differenti. A prima vista, lo stesso risultato sembra valere per le classi NL e coNL. Il fatto che NL sia uguale a coNL, come proveremo tra un attimo, mostra che la nostra intuizione sulla computazione ha ancora molte lacune.

TEOREMA 8.27

NL = coNL.

IDEA. Faremo vedere che $\overline{PATH}$ è contenuto in NL, e con ciò proveremo che ogni problema in coNL appartiene anche ad NL, perché $PATH$ è NL-completo. L'algoritmo NL M che presentiamo per $\overline{PATH}$ deve avere una computazione accettante ogni volta che il grafo di input G *non* contiene un cammino da s a t.

Prima di tutto, affrontiamo un problema più semplice. Sia c il numero di nodi in G che sono raggiungibili da s. Assumiamo che c venga fornito come input a M e facciamo vedere come usare c per risolvere $\overline{PATH}$. Successivamente mostreremo come calcolare c.

Dati G, s, t, e c, la macchina M opera come segue. Uno per uno, M attraversa tutti gli m nodi di G e non deterministicamente prova a vedere se ciascuno di essi è raggiungibile da s. Ogni volta che un nodo u è supposto raggiungibile, M tenta di verificare la supposizione, cercando di trovare un cammino di lunghezza m o meno da s a u. Se una diramazione della computazione fallisce nel verificare la supposizione, rifiuta. In aggiunta, se una diramazione verifica che t è raggiungibile, rifiuta. La macchina M conta il numero di nodi che ha verificato essere raggiungibili. Quando una diramazione ha attraversato tutti i nodi di G, controlla che il numero di nodi che ha verificato raggiungibili da s sia uguale a c, il numero di nodi che sono realmente raggiungibili, e rifiuta in caso contrario. Altrimenti, questa diramazione accetta.

In altre parole, se M seleziona non deterministicamente esattamente c nodi raggiungibili da s, che non includono t, e prova che ciascuno di essi è raggiungibile da s ricercando l'eventuale cammino, allora M sa che i nodi rimanenti, incluso t, *non* sono raggiungibili, pertanto può accettare.

Mostriamo a questo punto come calcolare c, il numero di nodi raggiungibili da s. Descriviamo una procedura non deterministica di spazio logaritmico per cui almeno una diramazione della computazione ha il valore corretto per c e tutte le altre diramazioni rifiutano.

Per ogni i da 0 a m, sia A_i la collezione di nodi che sono a una distanza pari a i o meno da s (i.e., che hanno un cammino di lunghezza al più i da s). Quindi $A_0 = \{s\}$, ciascun $A_i \subseteq A_{i+1}$, e A_m contiene tutti i nodi che sono raggiungibili da s. Sia c_i il numero di nodi in A_i. Nel seguito descriviamo una procedura che calcola c_{i+1} da c_i. L'applicazione ripetuta di questa procedura dà il valore desiderato di $c = c_m$.

Calcoliamo c_{i+1} da c_i usando un'idea simile a quella presentata prima in questo sketch della prova. L'algoritmo attraversa tutti i nodi di G, determina se ciascuno di essi è un elemento di A_{i+1}, e conta gli elementi.

Per stabilire se un nodo v appartiene ad A_{i+1}, usiamo un ciclo interno per passare attraverso tutti i nodi di G e vedere se ciascun nodo appartiene ad A_i. Ogni ipotesi positiva è verificata cercando di trovare il cammino di lunghezza al più i da s. Per ogni nodo u di cui è stata verificata l'appartenenza ad A_i, l'algoritmo controlla se (u, v) è un arco di G. Se è un arco, v appartiene ad A_{i+1}. In aggiunta, viene contato il numero di nodi di cui è stata verificata l'appartenenza ad A_i. Al completamento del ciclo interno, se il numero totale di nodi di cui è stata verificata l'appartenenza ad A_i non è c_i, non sono stati trovati tutti gli elementi di A_i, quindi questa diramazione della computazione rifiuta. Se il conto uguaglia c_i e non è stato ancora mostrato che v appartiene ad A_{i+1}, concludiamo che non appartiene a A_{i+1}. Quindi, procediamo dal v successivo nel ciclo più esterno.

DIMOSTRAZIONE. Ecco l'algoritmo per $\overline{PATH}$. Sia m il numero di nodi di G.

M = "Su input $\langle G, s, t\rangle$:

1. Sia $c_0 = 1$. ⟦$A_0 = \{s\}$ contiene 1 nodo⟧
2. Per $i = 0$ a $m - 1$: ⟦calcola c_{i+1} da c_i⟧
3. Sia $c_{i+1} = 1$. ⟦c_{i+1} conta i nodi in A_{i+1}⟧
4. Per ogni nodo $v \neq s$ in G: ⟦controlla se $v \in A_{i+1}$⟧
5. Sia $d = 0$. ⟦d riconta A_i⟧
6. Per ogni nodo u in G: ⟦controlla se $u \in A_i$⟧
7. Non deterministicamente esegui o salta questi passi:
8. Non deterministicamente segui un cammino di lunghezza al più i da s e *rifiuta* se non finisci in u.
9. Incrementa d. ⟦ha verificato che $u \in A_i$⟧
10. Se (u, v) è un arco di G, incrementa c_{i+1} e vai al passo 5 con il prossimo v. ⟦ha verificato che $v \in A_{i+1}$⟧
11. Se $d \neq c_i$, allora *rifiuta*. ⟦controlla se ha trovato tutto A_i⟧
12. Sia $d = 0$. ⟦c_m ora noto; d riconta A_m⟧
13. Per ogni nodo u in G: ⟦controlla se $u \in A_m$⟧
14. Non deterministicamente esegui o salta questi passi:
15. Non deterministicamente segui un cammino di lunghezza al più m da s e *rifiuta* se non finisci in u.
16. Se $u = t$, allora *rifiuta*. ⟦ha trovato un cammino da s a t⟧
17. Incrementa d. ⟦ha verificato che $u \in A_m$⟧
18. Se $d \neq c_m$, allora *rifiuta*. ⟦controlla se ha trovato tutto A_m⟧

Altrimenti, *accetta*."

L'algoritmo deve solo memorizzare m, u, v, c_i, c_{i+1}, d, i, ed un puntatore alla testa del cammino in un dato momento. Pertanto, computa in spazio logaritmico. (Si noti che M accetta anche input non formattati propriamente.)

Riassumiamo la nostra conoscenza attuale sulle relazioni tra le diverse classi di complessità come segue:

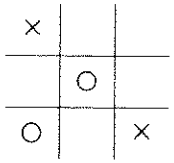
$$\mathrm{L} \subseteq \mathrm{NL} = \mathrm{coNL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}.$$

Non sappiamo se qualcuna di queste inclusioni è propria, sebbene proveremo che NL $\subsetneq$ PSPACE nel Corollario 9.6. Di conseguenza deve valere che o coNL $\subsetneq$ P oppure P $\subsetneq$ PSPACE, ma non sappiamo quale! La maggior parte dei ricercatori congettura che tutte queste inclusioni siano proprie.

ESERCIZI

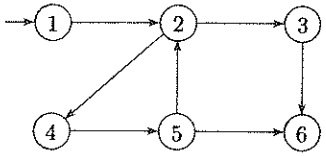
8.1 Si dimostri che per ogni funzione $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, dove $f(n) \geq n$, la classe di complessità SPACE($f(n)$) è la stessa sia che si definisca la classe attraverso il modello di TM a singolo nastro sia che si usi il modello di TM a due nastri con input di sola lettura.

8.2 Si consideri la seguente configurazione del gioco standard del tris.



Diciamo che è il turno del giocatore X di effettuare la mossa successiva. Si descriva una strategia vincente per questo giocatore. (Si ricordi che una strategia vincente non è semplicemente la mossa migliore da fare nella configurazione attuale. Essa include tutte le risposte che questo giocatore deve dare per vincere, comunque l'avversario muova.)

8.3 Si consideri il seguente gioco di geografia generalizzato dove il nodo iniziale è il nodo con l'arco entrante non proveniente da altri nodi. Il Giocatore I ha una strategia vincente? E il Giocatore II? Si motivino le risposte.



8.4 Si dimostri che PSPACE è chiusa rispetto alle operazioni di unione, complemento e star.

A8.5 Si dimostri che $A_{\mathrm{DFA}} \in \mathrm{L}$.

8.6 Si dimostri che ogni linguaggio PSPACE-hard è anche NP-hard.

A8.7 Si dimostri che NL è chiusa rispetto alle operazioni di unione, concatenazione, e star.

PROBLEMI

8.8** Il gioco ***Nim viene giocato con una collezione di mucchi di bastoncini. In una mossa singola, un giocatore può rimuove un numero (diverso da zero) di bastoncini da un singolo mucchio. I giocatori si alternano nell'effettuare le mosse. Il giocatore che rimuove l'ultimo bastoncino perde. Supponiamo di avere una disposizione in Nim con k mucchi contenenti $s_1, \ldots, s_k$ bastoncini. Chiamiamo la disposizione ***bilanciata*** se ciascuna colonna di bit contiene un numero pari di 1 quando ciascuno dei numeri s_i è scritto in binario, e i numeri in binario sono scritti come righe di una matrice allineate rispetto ai bit meno significativi. Si provino i due fatti seguenti.

a. Partendo da una disposizione non bilanciata, esiste una mossa singola che cambia la disposizione in bilanciata.

b. Partendo da una disposizione bilanciata, ogni mossa singola rende la disposizione non bilanciata.

Sia $NIM = \{\langle s_1, \ldots, s_k\rangle |$ ciascun s_i è un numero in binario e il Giocatore I ha una strategia vincente nel gioco Nim partendo da questa disposizione$\}$. Si usino i fatti precedenti circa le disposizioni bilanciate per dimostrare che $NIM \in \mathrm{L}$.

8.9 Sia $MULT = \{a\#b\#c | \ a, b, c$ sono numeri naturali in binario e $a \times b = c\}$. Si dimostri che $MULT \in \mathrm{L}$.

8.10 Per ogni intero positivo x, sia $x^{\mathcal{R}}$ l'intero la cui rappresentazione binaria è l'inversa della rappresentazione binaria di x. (Sia assuma che non ci sono 0 all'inizio della rappresentazione binaria di x.) Si definisca la funzione $\mathcal{R}^+: \mathcal{N} \longrightarrow \mathcal{N}$ dove $\mathcal{R}^+(x) = x + x^{\mathcal{R}}$.

a. Sia $A_2 = \{\langle x, y\rangle | \ \mathcal{R}^+(x) = y\}$. Si dimostri che $A_2 \in \mathrm{L}$.

b. Sia $A_3 = \{\langle x, y\rangle | \ \mathcal{R}^+(\mathcal{R}^+(x)) = y\}$. Si dimostri che $A_3 \in \mathrm{L}$.

8.11 **a.** Sia $ADD = \{\langle x, y, z\rangle | \ x, y, z > 0$ sono interi in binario e $x + y = z\}$. Si dimostri che $ADD \in \mathrm{L}$.

b. Sia $PAL\text{-}ADD = \{\langle x, y\rangle | \ x, y > 0$ sono interi in binario dove $x+y$ è un intero la cui rappresentazione binaria è palindroma$\}$. (Si noti che la rappresentazione binaria della somma che si assume non presenti zeri all'inizio. Una stringa palindroma è una stringa che coincide con la sua inversa.) Si dimostri che $PAL\text{-}ADD \in \mathrm{L}$.

***8.12** Sia $UCYCLE = \{\langle G\rangle | \ G$ è un grafo non orientato che contiene un ciclo semplice$\}$. Si dimostri che $UCYCLE \in \mathrm{L}$. (Nota: G potrebbe essere un grafo non connesso.)

***8.13** Per ogni n, si esibiscano due espressioni regolari, R ed S, di lunghezza $poly(n)$, dove $L(R) \neq L(S)$, ma dove la prima stringa su cui differiscono ha lunghezza esponenziale. In altre parole, $L(R)$ ed $L(S)$ devono essere differenti, seppur coincidenti su tutte le stringhe di lunghezza fino a 2^{cn}, per qualche costante $c > 0$.

8.14 Un grafo non orientato è ***bipartito*** se i suoi nodi possono essere divisi in due insiemi tali che tutti gli archi vanno da un nodo in un insieme verso un nodo nell'altro insieme. Si dimostri che un grafo è bipartito se e solo se non contiene un ciclo che ha un numero dispari di nodi. Sia $BIPARTITE = \{\langle G\rangle | \ G$ è un grafo bipartito$\}$. Si dimostri che $BIPARTITE \in \mathrm{NL}$.

8.15 Sia $UPATH$ la controparte di $PATH$ per grafi indiretti. Si dimostri che $\overline{BIPARTITE} \leq_{\mathrm{L}} UPATH$. (Nota: In realtà, possiamo provare che $UPATH \in \mathrm{L}$,

e pertanto *BIPARTITE* ∈ L, ma l' algoritmo [62] è troppo difficile da presentare qui.)

8.16 Ricordiamo che un grafo diretto è ***fortemente connesso*** se per ogni coppia di nodi esiste un cammino diretto in ciascuna direzione che li connette. Sia

$$STRONGLY\text{-}CONNECTED = \{\langle G\rangle \mid G \text{ è un grafo fortemente connesso}\}.$$

Si dimostri che *STRONGLY-CONNECTED* è NL-completo.

8.17 Sia $BOTH_{\text{NFA}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ ed } M_2 \text{ sono NFA dove } L(M_1) \cap L(M_2) \neq \emptyset\}$. Si dimostri che $BOTH_{\text{NFA}}$ è NL-completo.

8.18 Si dimostri che A_{NFA} è NL-completo.

8.19 Si dimostri che E_{DFA} è NL-completo.

***8.20** Si dimostri che *2SAT* è NL-completo.

8.21 Sia $CNF_{\text{H1}} = \{\langle\phi\rangle \mid \phi$ è una formula in forma cnf soddisfacibile dove ciascuna clausola contiene qualsiasi numero di letterali in forma vera ed al più un letterale in forma negata. Inoltre, ciascun letterale negato ha al più una occorrenza in $\phi\}$. Si dimostri che CNF_{H1} è NL-completo.

***8.22** Si fornisca un esempio di un linguaggio context-free NL-completo.

A***8.23** Sia $CYCLE = \{\langle G\rangle \mid G$ è un grafo diretto che contiene un ciclo diretto$\}$. Si dimostri che *CYCLE* è NL-completo.

8.24 Sia $EQ_{\text{REX}} = \{\langle R, S\rangle \mid R \text{ e } S$ sono espressioni regolari equivalenti$\}$. Si dimostri che $EQ_{\text{REX}} \in$ PSPACE.

8.25 Una *scala* è una sequenza di stringhe $s_1, s_2, \ldots, s_k$, dove ogni stringa differisce dalla precedente in esattamente un carattere. Per esempio, ciò che segue è una scala di parole inglesi, che comincia con "head" e finisce con "free":

head, hear, near, fear, bear, beer, deer, deed, feed, feet, fret, free.

Sia $LADDER_{\text{DFA}} = \{\langle M, s, t\rangle \mid M$ è un DFA ed $L(M)$ contiene una scala di stringhe, che comincia con s e finisce con $t\}$. Si dimostri che $LADDER_{\text{DFA}}$ è in PSPACE.

8.26 Il gioco giapponese *go-moku* viene giocato da due giocatori, "X" e "O," su una griglia 19 × 19. I giocatori sistemano a turno dei marcatori, e il primo dei giocatori che riesce a piazzare cinque dei suoi marcatori consecutivamente in una riga, colonna o diagonale è il vincitore. Si consideri questo gioco generalizzato ad una scacchiera di dimensioni $n \times n$. Sia

$$GM = \{\langle B\rangle \mid B \text{ è una disposizione nel go-moku generalizzato, dove il giocatore "X" ha una strategia vincente}\}.$$

Con il termine *disposizione* intendiamo una scacchiera con marcatori collocati su di essa, come potrebbe accadere nel mezzo dell'esecuzione del gioco, insieme con una indicazione di quale giocatore muove di seguito. Si dimostri che $GM \in$ PSPACE.

8.27 Si dimostri che se ogni linguaggio NP-hard è anche PSPACE-hard, allora PSPACE = NP.

8.28 Si dimostri che *TQBF* ristretto alle formule dove la parte che segue i quantificatori è in forma normale congiuntiva è ancora PSPACE-completo.

8.29 Sia $A_{\text{LBA}} = \{\langle M, w\rangle \mid M$ è un LBA che accetta input $w\}$. Si dimostri che A_{LBA} è PSPACE-completo.

***8.30** Il gioco del gatto e del topo viene giocato da due giocatori, "Cat" e "Mouse," su un grafo non orientato arbitrario. In ogni istante ciascun giocatore occupa un nodo del grafo. I giocatori a turno si muovono verso un nodo adiacente a quello che occupano nell'istante corrente. Un nodo speciale del grafo è chiamato "Hole." Cat vince se i due giocatori occupano continuativamente lo stesso nodo. Mouse vince se raggiunge il nodo Hole prima che la precedente condizione accada. Il gioco finisce in pareggio se una situazione si ripete (i.e., i due giocatori occupano simultaneamente posizioni che essi simultaneamente occupavano precedentemente, ed è il turno di mossa dello stesso giocatore).

$$HAPPY\text{-}CAT = \{\langle G, c, m, h\rangle \mid G, c, m, h \text{ sono rispettivamente un grafo e le posizioni di Cat, Mouse, ed Hole, tali che Cat ha una strategia vincente se Cat muove prima}\}.$$

Si dimostri che *HAPPY-CAT* è in P. (Suggerimento: La soluzione non è complicata e non dipende da dettagli sottili legati al modo in cui il gioco è definito. Si consideri l'intero albero del gioco. Ha taglia esponenziale, ma è possibile esplorarlo in tempo polinomiale.)

8.31 Si consideri la versione seguente per due persone del linguaggio *PUZZLE* che è stata descritta nel Problema 7.39. Ciascun giocatore inizia con una pila ordinata di carte per il puzzle. I giocatori a turno dispongono le carte in ordine nella scatola e possono scegliere quale lato rivolgere verso l'alto. Il Giocatore I vince se tutte le posizioni vuote sono ostruite nella pila finale, mentre il Giocatore II vince se qualche posizione vuota resta non bloccata. Si dimostri che il problema di determinare quale giocatore ha una strategia vincente per una data configurazione iniziale delle carte è PSPACE-completo.

8.32 Si rilegga la definizione di *MIN-FORMULA* nel Problema 7.21.

a. Si dimostri che *MIN-FORMULA* ∈ PSPACE.

b. Si spieghi perché questo argomento fallisce nel provare che *MIN-FORMULA* ∈ coNP: Se $\phi \notin$ *MIN-FORMULA*, allora ϕ ha una formula più piccola equivalente. Una NTM può verificare che $\phi \in \overline{\textit{MIN-FORMULA}}$ indovinando quella formula.

8.33 Sia A il linguaggio delle parentesi tonde propriamente annidate. Per esempio, `(())` e `(()(()))()` sono in A, ma `)(` no. Si dimostri che A è in L.

***8.34** Sia B il linguaggio delle parentesi tonde e quadre propriamente annidate. Per esempio, `([()()]()[])` è in B ma `(])` no. Si dimostri che B è in L.

SOLUZIONI SELEZIONATE

8.5 Si costruisca una TM M per decidere A_{DFA}. Quando M riceve in input $\langle A, w\rangle$, un DFA ed una stringa, M simula A su w tenendo traccia dello stato corrente di A e della posizione corrente della testina, aggiornandoli in modo appropriato. Lo spazio richiesto per effettuare questa simulazione è $O(\log n)$ perché M può registrare ciascuno di questi valori memorizzando un puntatore nel suo input.

8.7 Siano A_1 e A_2 linguaggi decidibili dalle macchine NL N_1 ed N_2. Si costruiscano tre macchine di Turing: $N_\cup$ che decide $A_1 \cup A_2$; $N_\circ$ che decide $A_1 \circ A_2$; ed N_* che decide A_1^*. Ciascuna di queste macchine opera come segue.

La macchina $N_\cup$ si dirama non deterministicamente per simulare N_1 o per simulare N_2. In ogni caso, $N_\cup$ accetta se la macchina simulata accetta.

La macchina $N_\circ$ seleziona non deterministicamente una posizione dell'input per dividerlo in due sottostringhe. Soltanto un puntatore a detta posizione viene memorizzato sul nastro di lavoro – non c'è spazio a sufficienza per memorizzare le sottostringhe stesse. Successivamente, $N_\circ$ simula N_1 sulla prima sottostringa, diramandosi non deterministicamente per simulare il non determininismo di N_1. Su ogni diramazione che raggiunge lo stato di accettazione di N_1, $N_\circ$ simula N_2 sulla seconda sottostringa. Su ogni diramazione che raggiunge lo stato di accettazione di N_2, $N_\circ$ accetta.

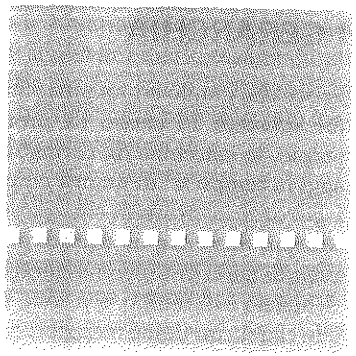
La macchina N_* ha un algoritmo più complesso, pertanto lo descriviamo per fasi.

N_* = "Su input w:

1. Inizializza a 0, la posizione immediatamente precedente il primo simbolo di input, due puntatori p_1 e p_2 a posizioni di input.
2. *Accetta* se non c'è nessun simbolo di input dopo p_2.
3. Muovi in avanti p_2 verso una posizione selezionata non deterministicamente.
4. Simula N_1 sulla sottostringa di w dalla posizione che segue p_1 alla posizione indicata da p_2, diramando non deterministicamente per simulare il non determinismo di N_1.
5. Se questa diramazione della simulazione raggiunge lo stato di accettazione di N_1, copia p_2 in p_1 e vai alla fase 2. Se N_1 rifiuta su questa diramazione, *rifiuta*."

8.23 Si riduce *PATH* a *CYCLE*. L'idea che sta dietro alla riduzione è modificare l'istanza $\langle G, s, t\rangle$ del problema *PATH* aggiungendo un arco da t a s in G. Se esiste un cammino da s a t in G, esisterà un ciclo diretto in G modificato. D'altronde, altri cicli potrebbero esistere in G modificato perché essi potrebbero già essere presenti in G. Per gestire il problema, prima di tutto si modifica G in modo tale che non contenga cicli. Un ***grafo diretto a livelli*** è un grafo dove i nodi sono divisi in gruppi, $A_1, A_2, \ldots, A_k$, chiamati *livelli*, e sono permessi soli archi da un livello al livello successivo più in alto. Si noti che un grafo a livelli è aciclico. Il problema *PATH* per grafi livellati è ancora NL-completo, come mostra la riduzione seguente dal problema *PATH* non ristretto. Dato un grafo G con due nodi s e t, ed m nodi in totale, si generi il grafo a livelli G' i cui livelli sono m copie dei nodi di G. Si disegni un arco dal nodo i a ciascun livello verso il nodo j nel livello successivo se G contiene un arco da i a j. Inoltre, si disegni un arco dal nodo i in ciascun livello verso il nodo i nel livello prossimo. Sia s' il nodo s nel primo livello e sia t' il nodo t nell'ultimo livello. Il grafo G contiene un cammino da s a t se e solo se G' contiene un cammino da s' a t'. Se si modifica G' aggiungendo un arco da t' a s', si ottiene una riduzione da *PATH* a *CYCLE*. La riduzione è semplice computazionalmente, e la sua implementazione in spazio logaritmico è routine. Per di più, una procedura non complicata mostra che $CYCLE \in$ NL. Pertanto, *CYCLE* è NL-completo.

9



INTRATTABILITÀ

Diversi problemi computazionali sono risolvibili in linea di principio, ma le soluzioni richiedono talmente tanto tempo o spazio che in pratica non possono essere utilizzate. Questi problemi vengono detti ***intrattabili***.

Nei Capitoli 7 e 8 abbiamo introdotto diversi problemi ritenuti intrattabili ma nessuno che è stato dimostrato essere tale. Per esempio, molte persone credono che il problema *SAT* e tutti gli altri problemi NP-completi siano intrattabili, nonostante non si sappia come dimostrare che lo sono. In questo capitolo daremo esempi di problemi di che possiamo provare essere intrattabili.

Al fine di presentare questi esempi, svilupperemo alcuni teoremi che rapportano la potenza delle macchine di Turing alla quantità di tempo o spazio disponibile per la computazione.

Concluderemo il capitolo con una discussione sulla possibilità di provare che problemi in NP risultano intrattabili e con ciò di risolvere la questione se P è diverso da NP. Prima di tutto introdurremo la tecnica della relativizzazione e la useremo per argomentare che certi metodi non ci permetteranno di raggiungere l'obiettivo. Successivamente discuteremo la teoria della complessità dei circuiti, un approccio perseguito dai ricercatori che sembra promettente.

9.1
I TEOREMI DI GERARCHIA

Il senso comune suggerisce che dando ad una macchina di Turing più spazio o più tempo dovrebbe crescere la classe di problemi che la macchina può risolvere. Per esempio, le macchine di Turing dovrebbero essere capaci di decidere più linguaggi in tempo n^3 di quanti ne possano decidere in tempo n^2. I ***teoremi di gerarchia*** provano che questa intuizione è corretta, posto che alcune condizioni descritte di seguito siano soddisfatte. Utilizziamo il termine *teorema di gerarchia* perché questi teoremi provano che le classi di complessità di spazio e di tempo non sono tutte uguali - ma formano una gerarchia dove le classi con limiti maggiori contengono più linguaggi di quanti ne posseggano le classi con limiti minori.

Il teorema di gerarchia per la complessità di spazio è leggermente più semplice di quello per la complessità di tempo, pertanto lo presentiamo prima. Cominciamo con la definizione tecnica che segue.

DEFINIZIONE **9.1**

Una funzione $f: \mathcal{N} \longrightarrow \mathcal{N}$, dove $f(n)$ è almeno $O(\log n)$, viene detta ***spazio-costruibile*** se la funzione che mappa la stringa 1^n nella rappresentazione binaria di $f(n)$ è computabile in spazio $O(f(n))$.[1]

In altre parole, f è spazio-costruibile se esiste una TM con spazio $O(f(n))$ che si arresta sempre con la rappresentazione binaria di $f(n)$ sul proprio nastro una volta avviata con 1^n in input. Funzioni a valori non interi come $n \log_2 n$ e $\sqrt{n}$ vengono arrotondate all'intero successivo più piccolo ai fini della costruibilità in tempo e spazio.

ESEMPIO 9.2

Tutte le funzioni che si presentano comunemente e che sono almeno $O(\log n)$ sono spazio-costruibili, includendo le funzioni $\log_2 n$, $n \log_2 n$, ed n^2.

Per esempio, n^2 è spazio-costruibile perché una macchina può prendere il proprio input 1^n, ottenere n in binario contando il numero di 1, e dare in output n^2 usando uno qualsiasi dei metodi standard per multiplicare n con se stesso. Lo spazio totale utilizzato è $O(n)$, che è certamente $O(n^2)$.

Nel mostrare che funzioni $f(n)$ che sono $o(n)$ sono spazio-costruibili, utilizziamo un nastro di input separato a sola lettura, come abbiamo fatto

[1] Si ricordi che 1^n indica una stringa di n simboli 1.

quando abbiamo definito la complessità di spazio quasi lineare in Sezione 8.4. Per esempio, una tale macchina può calcolare la funzione che associa a 1^n la rappresentazione binaria di $\log_2 n$ come segue. La macchina prima conta il numero di 1 in input in binario, usando il proprio nastro di lavoro, mentre muove la propria testina lungo il nastro di input. Quindi, con n in binario sul proprio nastro di lavoro, può calcolare $\log_2 n$ contando il numero di bit nella rappresentazione binaria di n.

Il ruolo della costruibilità in spazio nel teorema di gerarchia di spazio può essere compreso attraverso la situazione seguente. Se $f(n)$ e $g(n)$ sono due limitazioni di spazio, dove $f(n)$ è asintoticamente maggiore di $g(n)$, ci aspetteremmo che una macchina risultasse in grado di decidere più linguaggi in spazio $f(n)$ che in spazio $g(n)$. Tuttavia, supponiamo che $f(n)$ ecceda $g(n)$ soltanto per una quantità di spazio molto piccola e difficile da calcolare. Allora, la macchina potrebbe non essere in grado di usare con profitto lo spazio extra perché anche la mera computazione della quantità di spazio extra potrebbe richiedere più spazio di quanto disponibile. In questo caso, una macchina potrebbe non essere in grado di calcolare più linguaggi in spazio $f(n)$ di quanto possa in spazio $g(n)$. Convenendo che $f(n)$ sia spazio-costruibile evita questa situazione e ci permette di provare che una macchina può calcolare più di quanto sarebbe in grado di fare con una limitazione asintoticamente minore, come il teorema seguente dimostra.

TEOREMA 9.3

Teorema di gerarchia di spazio Per ogni funzione $f: \mathcal{N} \longrightarrow \mathcal{N}$ spazio-costruibile, esiste un linguaggio A che è decidibile in spazio $O(f(n))$ ma non in spazio $o(f(n))$.

IDEA. Dobbiamo mostrare un linguaggio A che ha due proprietà. La prima stabilisce che A è decidibile in spazio $O(f(n))$. La seconda stabilisce che A non è decidibile in spazio $o(f(n))$.

Descriviamo A fornendo un algoritmo D che lo decide. L'algoritmo D computa in spazio $O(f(n))$, garantendo con ciò la prima proprietà. Per di più, D garantisce che A è differente da ogni altro linguaggio che risulti decidibile in spazio $o(f(n))$, garantendo quindi la seconda proprietà. Il linguaggio A è differente dai linguaggi discussi in precedenza perché non sussiste per esso una definizione non algoritmica. Pertanto, non possiamo offrirne una semplice immagine mentale.

Al fine di garantire che A non risulti decidibile in spazio $o(f(n))$, costruiamo D implementando il metodo della diagonalizzazione che abbiamo usato nel Teorema 4.11 a pagina 212 per dimostrare che il problema dell'accettazione A_{TM} è indecidibile. Se M è una TM che decide un linguaggio

in spazio $o(f(n))$, D garantisce che A differisce dal linguaggio di M in almeno una posizione. Quale posizione? La posizione corrispondente alla descrizione di M stessa.

Diamo uno sguardo al modo in cui D opera. In parole povere, D prende in input la descrizione di una TM M. (Se l'input non è la descrizione di una TM, allora l'agire di D non è significativo su questo input, per cui possiamo arbitrariamente far sì che D rifiuti.) Successivamente, D esegue M sullo stesso input - ossia, $\langle M \rangle$ - entro il limite di spazio $f(n)$. Se M si arresta entro tale quantità di spazio, D accetta se e solo se M rifiuta. Se M non si arresta, D semplicemente rifiuta. Pertanto, se M computa entro il limite di spazio $f(n)$, D dispone di spazio a sufficienza per garantire che il suo linguaggio risulti differente da quello di M. Se no, D non ha abbastanza spazio per capire cosa fa M. Ma fortunatamente D non deve soddisfare alcun requisito che imponga un comportamento differente da macchine che non computano in spazio $o(f(n))$, pertanto l'agire di D su un tale input non è significativo.

Questa descrizione cattura l'essenza della dimostrazione ma omette diversi dettagli importanti. Se M computa in spazio $o(f(n))$, D deve garantire che il suo linguaggio è differente dal linguaggio di M. Ma anche quando computa in spazio $o(f(n))$, M potrebbe usare una quantità di spazio maggiore di $f(n)$ per valori piccoli di n, quando il comportamento asintotico non è stato raggiunto ancora. Potrebbe succedere che D non ha abbastanza spazio per eseguire M fino al completamento sull'input $\langle M \rangle$, e quindi D perderebbe la propria opportunità di differire dal linguaggio di M. Pertanto, se non procediamo con cura, D potrebbe terminare decidendo lo stesso linguaggio che decide M, ed il teorema non risulterebbe dimostrato.

Possiamo risolvere questo problema modificando D fornendogli opportunità aggiuntive per evitare il linguaggio di M. Invece di eseguire M soltanto quando D riceve input $\langle M \rangle$, essa esegue M ogni volta che riceve un input della forma $\langle M \rangle \mathtt{10}^*$; cioè, un input della forma $\langle M \rangle$ seguita da un 1 ed un certo numero di 0. Successivamente, se M sta computando realmente in spazio $o(f(n))$, D disporrà di abbastanza spazio per eseguirla fino al termine su input $\langle M \rangle \mathtt{10}^k$ per qualche valore grande di k perché il comportamento asintotico alla fine deve essere raggiunto.

Deve essere gestito un ultimo dettaglio tecnico. Quando D esegue M su una qualche stringa, M potrebbe finire in un ciclo infinito anche se sta usando soltanto una quantità di spazio finita. Ma si suppone che D sia un decisore, pertanto dobbiamo garantire che D non entri in ciclo nel corso della simulazione di M. Qualsiasi macchina che computa in spazio $o(f(n))$ usa soltanto tempo $2^{o(f(n))}$. Modifichiamo D in modo tale che conti il numero di passi usati nel simulare M. Laddove questo conto dovesse eccedere $2^{f(n)}$, allora D rifiuterebbe.

DIMOSTRAZIONE. L'algoritmo D di spazio $O(f(n))$ che segue decide un linguaggio A che non è decidibile in spazio $o(f(n))$.

D = "Su input w:

1. Sia n la lunghezza di w.
2. Calcola $f(n)$ usando la costruibilità in spazio e marca questa quantità sul nastro. Se in fasi successive si verifica un tentativo di usarne di più, *rifiuta*.
3. Se w non è della forma $\langle M \rangle \mathtt{10}^*$ per qualche TM M, *rifiuta*.
4. Simula M su w contando allo stesso tempo il numero di passi usati durante la simulazione. Se il conto eccede $2^{f(n)}$, *rifiuta*.
5. Se M accetta, *rifiuta*. Se M rifiuta, *accetta*."

Nella fase 4, dobbiamo fornire dettagli aggiuntivi della simulazione al fine di determinare la quantità di spazio usato. La TM M simulata dispone di un alfabeto di nastro arbitrario e D ha un alfabeto di nastro fissato, pertanto rappresentiamo ciascuna cella del nastro di M con diverse celle del nastro di D. Quindi, la simulazione introduce un fattore costante aggiuntivo nello spazio usato. In altre parole, se M computa in spazio $g(n)$, allora D usa spazio $d\,g(n)$ per simulare M per qualche costante d che dipende da M.

La macchina D è un decisore perché ciascuna delle sue fasi può durare un tempo limitato. Sia A il linguaggio che D decide. Chiaramente, A è decidibile in spazio $O(f(n))$ perché D decide in questo modo. Nel seguito, mostriamo che A non è decidibile in spazio $o(f(n))$.

Si assuma al contrario che una macchina di Turing M decida A in spazio $g(n)$, tale che $g(n)$ è $o(f(n))$. Come menzionato in precedenza, D può simulare M, usando spazio $d\,g(n)$ per qualche costante d. Poiché $g(n)$ è $o(f(n))$, esiste una costante n_0, dove $d\,g(n) < f(n)$ per tutti gli $n \geq n_0$. Pertanto, la simulazione di M da parte di D giungerà a completamento purché l'input abbia lunghezza n_0 o più. Si consideri cosa accade quando D viene eseguito sull'input $\langle M \rangle \mathtt{10}^{n_0}$. Questo input è più lungo di n_0, e quindi la simulazione nella fase 4 giungerà a termine. Pertanto, D farà l'opposto di M sullo stesso input. Quindi M non decide A, contraddicendo la nostra assunzione. In conclusione, A non è decidibile in spazio $o(f(n))$.

COROLLARIO 9.4

Per ogni coppia di funzioni $f_1, f_2 : \mathcal{N} \longrightarrow \mathcal{N}$, dove $f_1(n)$ è $o(f_2(n))$ ed f_2 è spazio-costruibile, $\mathrm{SPACE}(f_1(n)) \subsetneq \mathrm{SPACE}(f_2(n))$.[2]

[2] Si ricordi che $A \subsetneq B$ significa che A è un sottoinsieme proprio (i.e., non uguale) di B.

Questo corollario ci permette di separare varie classi di complessità di spazio. Per esempio, possiamo mostrare che la funzione n^c è spazio-costruibile per ogni numero naturale c. Pertanto, per ogni coppia di numeri naturali $c_1 < c_2$, possiamo dimostrare che $\text{SPACE}(n^{c_1}) \subsetneq \text{SPACE}(n^{c_2})$. Con poco lavoro in più possiamo dimostrare che n^c è spazio-costruibile per ogni numero razionale $c > 0$ e con ciò estendere la precedente inclusione in modo tale che valga per ogni coppia di numeri razionali $0 \leq c_1 < c_2$. Notando che esistono sempre due numeri razionali c_1 e c_2 tra ogni coppia di numeri reali $\epsilon_1 < \epsilon_2$ tali che $\epsilon_1 < c_1 < c_2 < \epsilon_2$, otteniamo il corollario aggiuntivo seguente, che mostra una sottile gerarchia all'interno della classe PSPACE.

COROLLARIO 9.5

Per ogni coppia di numeri reali $0 \leq \epsilon_1 < \epsilon_2$,

$$\text{SPACE}(n^{\epsilon_1}) \subsetneq \text{SPACE}(n^{\epsilon_2}).$$

Possiamo usare il teorema di gerarchia di spazio anche per separare due classi di complessità di spazio che abbiamo incontrato precedentemente.

COROLLARIO 9.6

NL $\subsetneq$ PSPACE.

DIMOSTRAZIONE. Il teorema di Savitch mostra che $\text{NL} \subseteq \text{SPACE}(\log^2 n)$, ed il teorema della gerarchia di spazio mostra che $\text{SPACE}(\log^2 n) \subsetneq \text{SPACE}(n)$. Pertanto, il corollario è provato.

Come abbiamo osservato a pagina 381, questa separazione mostra che *TQBF* $\notin$ NL perché *TQBF* è PSPACE-completo rispetto alla riducibilità in spazio logaritmico.

A questo punto raggiungiamo l'obiettivo principale di questo capitolo: mostrare l'esistenza di problemi che sono decidibili in linea di principio ma non in pratica - cioè, problemi che sono decidibili ma intrattabili. Ciascuna delle classi $\text{SPACE}(n^k)$ è contenuta nella classe $\text{SPACE}(n^{\log n})$, che a sua volta è strettamente contenuta nella classe $\text{SPACE}(2^n)$. Pertanto, otteniamo il corollario aggiuntivo che segue, che separa PSPACE da $\text{EXPSPACE} = \bigcup_k \text{SPACE}(2^{n^k})$.

COROLLARIO 9.7

PSPACE $\subsetneq$ EXPSPACE.

Questo corollario stabilisce l'esistenza di problemi decidibili che sono intrattabili, nel senso che le procedure per la loro decisione devono usare una quantità di spazio più che polinomiale. I linguaggi stessi sono in qualche modo artificiali - interessanti soltanto per il fine di separare le classi di complessità. Utilizzeremo questi linguaggi per provare l'intrattabilità di altri linguaggi più naturali, dopo aver discusso il teorema di gerarchia di tempo.

DEFINIZIONE 9.8

Una funzione $t: \mathcal{N} \longrightarrow \mathcal{N}$, dove $t(n)$ è almeno $O(n \log n)$, viene detta ***tempo-costruibile*** se la funzione che associa alla stringa 1^n la rappresentazione binaria di $t(n)$ è calcolabile in tempo $O(t(n))$.

In altre parole, t è tempo-costruibile se esiste una TM di tempo $O(t(n))$ che si arresta sempre con la rappresentazione binaria di $t(n)$ sul suo nastro, una volta avviata con 1^n in input.

ESEMPIO 9.9

Tutte le funzioni che si presentano comunemente che sono almeno $n \log n$ sono costruibili in tempo, includendo le funzioni $n \log n$, $n\sqrt{n}$, n^2, e 2^n.

Per esempio, per mostrare che $n\sqrt{n}$ è tempo-costruibile, prima progettiamo una TM per contare il numero di 1 in binario. Per far ciò, la TM realizza un contatore binario lungo il nastro, incrementandolo di 1 per ogni posizione dell'input, fino a quando raggiunge la fine dell'input. Questa parte richiede $O(n \log n)$ passi, perché $O(\log n)$ passi vengono effettuati per ognuna delle n posizioni dell'input. Successivamente, calcoliamo $\lfloor n\sqrt{n} \rfloor$ in binario a partire dalla rappresentazione binaria di n. Qualsiasi metodo ragionevole per far ciò richiederà tempo $O(n \log n)$ poiché la lunghezza dei numeri coinvolti è $O(\log n)$.

Il teorema di gerarchia di tempo è un analogo per la complessità di tempo del Teorema 9.3. Per ragioni tecniche che saranno chiare nel corso della dimostrazione, il teorema di gerarchia di tempo è leggermente più debole di quello provato per lo spazio. Laddove nella costruibilità in spazio *qualsiasi* incremento asintotico al limite spaziale allarga la classe dei linguaggi ivi decidibili, per il tempo dobbiamo ulteriormente incrementare il limite al tempo di un fattore logaritmico, al fine di garantire che possiamo ottenere linguaggi aggiuntivi. Plausibilmente un teorema di gerarchia di tempo più stretto è vero; ma al momento non sappiamo come provarlo. Questa peculiarità del teorema di gerarchia di tempo nasce dal fatto che misuriamo

la complessità di tempo con macchine di Turing a singolo nastro. Possiamo provare teoremi di gerarchia di tempo più stretti per altri modelli di computazione.

TEOREMA 9.10

Teorema di gerarchia di tempo Per ogni funzione tempo-costruibile $t: \mathcal{N} \longrightarrow \mathcal{N}$, esiste un linguaggio A che è decidibile in tempo $O(t(n))$ ma non è decidibile in tempo $o(t(n)/\log t(n))$.

IDEA. Questa dimostrazione è simile alla dimostrazione del Teorema 9.3. Costruiamo una TM D che decide un linguaggio A in tempo $O(t(n))$, dove A non può essere deciso in tempo $o(t(n)/\log t(n))$. In questo contesto, D prende un input w della forma $\langle M \rangle 1 0^*$ e simula M sull'input w, assicurandosi di non usare un tempo maggiore di $t(n)$. Se M si arresta entro questo limite temporale, D restituisce l'output opposto.

La differenza importante nella dimostrazione riguarda il costo per simulare M contando, allo stesso tempo, il numero di passi che la simulazione sta usando. La macchina D deve effettuare questa simulazione temporizzata in modo efficiente, tale da garantire che D computi in tempo $O(t(n))$ e realizzando l'obiettivo di evitare tutti i linguaggi che risultano decidibili in tempo $o(t(n)/\log t(n))$. Per la complessità di spazio, come abbiamo osservato nella dimostrazione del Teorema 9.3, la simulazione ha introdotto un fattore aggiuntivo costante. Per la complessità di tempo, la simulazione introduce un fattore aggiuntivo logaritmico. Il fattore aggiuntivo maggiore per il tempo è la ragione della presenza del fattore $1/\log t(n)$ nell'enunciato del teorema. Se disponessimo di un modo per simulare una TM a singolo nastro attraverso un'altra TM a singolo nastro per un numero di passi specificato in precedenza, usando soltanto un fattore aggiuntivo costante in tempo, saremmo in grado di rafforzare il teorema sostituendo $o(t(n)/\log t(n))$ con $o(t(n))$. Una simulazione efficiente di questo tipo non è nota.

DIMOSTRAZIONE. L'algoritmo D di tempo $O(t(n))$ che segue decide un linguaggio A che non è decidibile in tempo $o(t(n)/\log t(n))$.

D = "Su input w:

1. Sia n la lunghezza di w.
2. Calcola $t(n)$ usando la costruibilità in tempo e memorizza il valore $\lceil t(n)/\log t(n) \rceil$ in un contatore binario. Decrementa questo contatore prima di ogni passo effettuato per realizzare le fasi 4 e 5. Se il contatore dovesse raggiungere 0, *rifiuta*.
3. Se w non è della forma $\langle M \rangle 1 0^*$ per qualche TM M, *rifiuta*.
4. Simula M su w.
5. Se M accetta, allora *rifiuta*. Se M rifiuta, allora *accetta*."

Esaminiamo ciascuna delle fasi dell'algoritmo per determinare il tempo di esecuzione. Le fasi 1, 2, e 3 possono essere effettuate entro tempo $O(t(n))$.

Durante la fase 4, ogni volta che D simula un passo di M, prende lo stato corrente di M insieme con il simbolo del nastro sotto la testina di M determina la mossa successiva di M in base alla funzione di transizione in modo tale da poter aggiornare il nastro di M in modo appropriato. Tutti e tre questi oggetti (stato, simbolo di nastro, e funzione di transizione) sono memorizzati da qualche parte sul nastro di D. Se fossero memorizzati lontano l'uno dall'altro, D avrebbe necessità di diversi passi per raccogliere queste informazioni ogni volta che simula uno dei passi di M. Invece D mantiene sempre queste informazioni vicine.

Possiamo pensare al singolo nastro di D come organizzato in *tracce*. Un modo per ottenere due tracce è attraverso la memorizzazione di una traccia nelle posizioni dispari e dell'altra nelle posizioni pari. Alternativamente, l'effetto delle due tracce può essere ottenuto allargando l'alfabeto del nastro di D per includere ciascuna coppia di simboli: uno dalla traccia superiore ed il secondo dalla traccia inferiore. In modo simile possiamo ottenere l'effetto di tracce aggiuntive. Nota che tracce multiple introducono soltanto un fattore aggiuntivo costante al tempo, posto che vengano utilizzate soltanto un numero fissato di tracce. In questo caso D ha tre tracce.

Una delle tracce contiene l'informazione sul nastro di M, e la seconda contiene il suo stato corrente ed una copia della funzione di transizione di M. Durante la simulazione, D mantiene l'informazione sulla seconda traccia vicino alla posizione corrente della testina di M sulla prima traccia. Ogni volta che la posizione della testina di M cambia, D fa slittare tutta l'informazione sulla seconda traccia per mantenerla vicino alla testina. Poiché la taglia dell'informazione sulla seconda traccia dipende soltanto da M e non dalla lunghezza dell'input di M, lo slittamento aggiunge soltanto un fattore costante al tempo della simulazione. Inoltre, poiché l'informazione richiesta viene mantenuta insieme e vicina, il costo per determinare la mossa successiva di M dalla sua funzione di transizione e per aggiornare il suo nastro è soltanto costante. Pertanto, se M computa in tempo $g(n)$, D può simularne l'esecuzione in tempo $O(g(n))$.

Ad ogni passo in fase 4, D deve decrementare il contatore di passi, inizializzato originariamente in fase 2. A tal proposito, D può agire senza aggiungere eccessivamente tempo alla simulazione, mantenendo il contatore in binario su una terza traccia e spostandolo per mantenerlo vicino alla posizione attuale della testina. Questo contatore ha una ampiezza massima di circa $t(n)/\log t(n)$, pertanto la sua lunghezza è $\log(t(n)/\log t(n))$, che è $O(\log t(n))$. Quindi il costo di aggiornamento e spostamento ad ogni passo aggiunge un fattore $\log t(n)$ al tempo della simulazione, portando dunque il tempo totale di esecuzione a $O(t(n))$. Pertanto, A è decidibile in tempo $O(t(n))$.

Per mostrare che A non è decidibile in tempo $o(t(n)/\log t(n))$, usiamo un argomento simile a quello usato nella dimostrazione del Teorema 9.3.

Si assuma al contrario che una TM M decida A in tempo $g(n)$, dove $g(n)$ è $o(t(n)/\log t(n))$. In tal caso, D può simulare M, usando tempo $d\,g(n)$ per qualche costante d. Se il tempo totale della simulazione (non contando il tempo per aggiornare il contatore di passi) è al più $t(n)/\log t(n)$, la simulazione verrà eseguita fino al completamento. Poiché $g(n)$ è $o(t(n)/\log t(n))$, esiste una costante n_0 tale che $d\,g(n) < t(n)/\log t(n)$ per tutti gli $n \geq n_0$. Pertanto, la simulazione di M da parte di D verrà eseguita fino al completamento purché l'input abbia lunghezza n_0 o maggiore. Si consideri cosa accade quando D viene eseguito sull'input $\langle M\rangle 10^{n_0}$. Questo input è più lungo di n_0, pertanto la simulazione nella fase 4 giungerà a termine. Pertanto, D farà l'opposto di M sullo stesso input. Quindi M non decide A, contraddicendo la nostra assunzione. In conclusione, A non è decidibile in tempo $o(t(n)/\log t(n))$.

Enunciamo gli analoghi dei Corollari 9.4, 9.5 e 9.7 per la complessità di tempo.

COROLLARIO 9.11

Per ogni coppia di funzioni $t_1, t_2 : \mathcal{N} \longrightarrow \mathcal{N}$, dove $t_1(n)$ è $o(t_2(n)/\log t_2(n))$ e t_2 è tempo-costruibile, $\text{TIME}(t_1(n)) \subsetneq \text{TIME}(t_2(n))$.

COROLLARIO 9.12

Per ogni coppia di numeri reali $1 \leq \epsilon_1 < \epsilon_2$, risulta $\text{TIME}(n^{\epsilon_1}) \subsetneq \text{TIME}(n^{\epsilon_2})$.

COROLLARIO 9.13

P $\subsetneq$ EXPTIME.

Completezza in spazio esponenziale

Possiamo usare i risultati precedenti per mostrare che un linguaggio specifico è realmente intrattabile. Lo facciamo in due passi. Innanzitutto i teoremi di gerarchia ci dicono che una macchina di Turing può decidere più linguaggi in EXPSPACE di quanto possa in PSPACE. Successivamente facciamo vedere che un particolare linguaggio che riguarda espressioni regolari generalizzate risulta completo per EXPSPACE e, quindi, non può essere deciso in tempo polinomiale o anche in spazio polinomiale.

Prima di soffermarci sulla loro generalizzazione, rivediamo brevemente il modo in cui abbiamo introdotto le espressioni regolari nella Definizione 1.52. Esse sono costruite a partire dalle espressioni atomiche $\emptyset$, ε, e da simboli dell'alfabeto, usando le operazioni regolari di unione, concatenazione e star, denotate con $\cup$, $\circ$, e *, rispettivamente. Dal Problema 8.24, sappiamo che possiamo verificare l'equivalenza di due espressioni regolari in spazio polinomiale.

Facciamo vedere che permettendo espressioni regolari con più operazioni delle usuali operazioni regolari, la complessità dell'analisi delle espressioni può crescere in modo drammatico.

Sia $\uparrow$ l'***operatore di esponenziazione***. Se R è un'espressione regolare e k è un intero non negativo, scrivere $R \uparrow k$ è equivalente a concatenare R con se stessa k volte. Scriviamo anche R^k come un'abbreviazione per $R \uparrow k$. In altre parole,

$$R^k = R \uparrow k = \overbrace{R \circ R \circ \cdots \circ R}^{k}.$$

Le espressioni regolari generalizzate permettono l'operatore di esponenziazione in aggiunta alle usuali operazioni regolari. Ovviamente, queste espressioni regolari generalizzate, generano ancora la stessa classe di linguaggi regolari delle espressioni regolari standard, poiché possiamo eliminare l'operatore di esponenziazione ripetendo l'espressione di base. Sia

$$EQ_{\text{REX}\uparrow} = \{\langle Q, R\rangle \mid Q \text{ ed } R \text{ sono espressioni regolari equivalenti con esponenziazione}\}.$$

Per mostrare che $EQ_{\text{REX}\uparrow}$ è intrattabile, dimostriamo che è completo per la classe EXPSPACE. Qualsiasi problema EXPSPACE-completo non può stare in PSPACE, men che meno in P. Altrimenti, EXPSPACE sarebbe uguale a PSPACE, contraddicendo il Corollario 9.7.

DEFINIZIONE 9.14

Un linguaggio B è ***EXPSPACE-completo*** se

1. $B \in$ EXPSPACE, e
2. ogni A in EXPSPACE è riducibile in tempo polinomiale a B.

TEOREMA 9.15

$EQ_{\text{REX}\uparrow}$ è EXPSPACE-completo.

IDEA. Nel misurare la complessità di decisione di $EQ_{\mathsf{REX}\uparrow}$, assumiamo che tutti gli esponenti siano scritti come interi in binario. La lunghezza di una espressione è il numero totale di simboli che essa contiene.

Uno schema di un algoritmo EXPSPACE per $EQ_{\mathsf{REX}\uparrow}$ è il seguente. Per verificare se due espressioni con esponenziazione sono equivalenti, prima usiamo la ripetizione per eliminare l'esponenziazione, poi convertiamo le espressioni risultanti in NFA. Infine, usiamo una procedura di verifica dell'equivalenza tra NFA simile a quella usata per decidere il complemento di ALL_{NFA} nell'Esempio 8.4.

Per mostrare che un linguaggio A in EXPSPACE è riducibile in tempo polinomiale a $EQ_{\mathsf{REX}\uparrow}$, utilizziamo la tecnica delle riduzioni attraverso le storie della computazione che abbiamo introdotto in Sezione 5.1. La costruzione è simile alla costruzione fornita nella prova del Teorema 5.13.

Data una TM M per A, progettiamo una riduzione di tempo polinomiale che associa a un input w una coppia di espressioni, R_1 ed R_2, che sono equivalenti esattamente quando M accetta w. Le espressioni R_1 ed R_2 simulano la computazione di M su w. L'espressione R_1 genera semplicemente tutte le stringhe sull'alfabeto che consiste dei simboli che possono esser presenti in storie di computazione. L'espressione R_2 genera tutte le stringhe che non corrispondono a storie di computazione di rifiuto. Quindi, se la TM accetta il proprio input, non esiste alcuna storia di computazione di rifiuto, e le espressioni R_1 ed R_2 generano lo stesso linguaggio. Si ricordi che una storia di computazione di rifiuto è la sequenza di configurazioni in cui la macchina transita durante una computazione su un input che termina con un rifiuto. Si consulti pagina 233 in Sezione 5.1 per un ripasso delle storie di computazione.

La difficoltà in questa prova è che la dimensione delle espressioni costruite deve essere polinomiale in n (in modo tale che la riduzione possa essere eseguita in tempo polinomiale), mentre la computazione simulata potrebbe avere lunghezza esponenziale. L'operazione di esponenziazione è utile qui per rappresentare una computazione lunga con una espressione relativamente breve.

DIMOSTRAZIONE. Prima di tutto presentiamo un algoritmo non deterministico per verificare se due NFA non sono equivalenti.

N = "Su input $\langle N_1, N_2\rangle$, dove N_1 ed N_2 sono NFA:

1. Colloca un marcatore su ciascuno degli stati iniziali di N_1 ed N_2.
2. Ripeti $2^{q_1+q_2}$ volte, dove q_1 e q_2 rappresentano il numero degli stati in N_1 ed N_2:
3. Non deterministicamente seleziona un simbolo di input e cambia le posizioni dei marcatori sugli stati di N_1 ed N_2 per simulare la lettura del simbolo.

4. Se in qualsiasi momento un marcatore era stato collocato sullo stato di accettazione di uno degli automi finiti e non sullo stato di accettazione dell'altro automa finito, *accetta*. Altrimenti, *rifiuta*."

Se gli automi N_1 ed N_2 sono equivalenti, N chiaramente rifiuta perché accetta solo quando appura che una macchina accetta una stringa che l'altra non accetta. Se gli automi non sono equivalenti, qualche stringa viene accettata da una macchina ma non dall'altra. Una tale stringa deve avere una lunghezza pari al più a $2^{q_1+q_2}$. Altrimenti, si supponga di usare la stringa più corta di questo tipo associata alla sequenza non deterministica di scelte. Esistono soltanto $2^{q_1+q_2}$ modi diversi di collocare i marcatori sugli stati di N_1 ed N_2; quindi, in una stringa più lunga, le posizioni dei marcatori debbono ripetersi. Rimuovendo la porzione della stringa tra le ripetizioni, si otterrebbe una stringa più corta. Pertanto, l'algoritmo N individuerebbe questa stringa tra le sue scelte non deterministiche e accetterebbe. Di conseguenza, N opera correttamente.

L'algoritmo N computa non deterministicamente in spazio lineare. Pertanto, il teorema di Savitch fornisce un algoritmo deterministico che usa spazio $O(n^2)$ per questo problema. Di seguito utilizzeremo la forma deterministica di questo algoritmo per progettare l'algoritmo seguente E che decide $EQ_{\mathsf{REX}\uparrow}$.

E = "Su input $\langle R_1, R_2\rangle$, dove R_1 ed R_2 sono espressioni regolari con esponenziazione:

1. Converti R_1 ed R_2 in espressioni regolari equivalenti B_1 e B_2 che usano la ripetizione invece dell'esponenziazione.
2. Converti B_1 e B_2 in NFA equivalenti N_1 ed N_2, usando la procedura di conversione data nella prova del Lemma 1.55.
3. Usa la versione deterministica dell'algoritmo N per stabilire se N_1 ed N_2 sono equivalenti."

L'algoritmo E è ovviamente corretto. Per analizzare la sua complessità di spazio, osserviamo che l'uso delle ripetizioni per rimuovere l'esponenziazione può incrementare la lunghezza di un'espressione di un fattore pari a 2^l, dove l è la somma delle lunghezze degli esponenti. Pertanto, le espressioni B_1 e B_2 hanno una lunghezza pari al più a $n2^n$, dove n è la lunghezza dell'input. La procedura di conversione del Lemma 1.55 fa crescere la dimensione linearmente, e quindi gli NFA N_1 ed N_2 hanno al più $O(n2^n)$ stati. Discende che, con input di dimensione $O(n2^n)$, la versione deterministica dell'algoritmo N usa spazio $O((n2^n)^2) = O(n^2 2^{2n})$. Pertanto, $EQ_{\mathsf{REX}\uparrow}$ è decidibile in spazio esponenziale.

A questo punto, mostriamo che $EQ_{\mathsf{REX}\uparrow}$ è EXPSPACE-hard. Sia A un linguaggio deciso dalla TM M che computa in spazio $2^{(n^k)}$ per qualche costante k. La riduzione associa a un input w una coppia di espressioni

regolari, R_1 ed R_2. L'espressione R_1 è Δ^* dove, se Γ e Q sono l'alfabeto del nastro e gli stati di M, $\Delta = \Gamma \cup Q \cup \{\#\}$ è l'alfabeto che consiste di tutti i simboli che possono comparire in una storia di computazione. Costruiamo l'espressione R_2 per generare tutte le stringhe che non sono storie di computazione di rifiuto di M su w. Naturalmente, M accetta w se e solo se M su w non ha storie di computazione di rifiuto. Pertanto, le due espressioni sono equivalenti se e solo se M accetta w. La costruzione procede come segue.

Una storia di computazione di rifiuto per M su w è una sequenza di configurazioni separate da simboli #. Utilizziamo la nostra codifica standard delle configurazioni laddove un simbolo corrispondente allo stato corrente viene collocato alla sinistra della posizione corrente della testina. Assumiamo che tutte le configurazioni abbiano lunghezza $2^{(n^k)}$ e siano completate sulla destra attraverso simboli blank nel caso fossero più corte. La prima configurazione in una storia di computazione di rifiuto è la configurazione iniziale di M su w. L'ultima configurazione è la configurazione di rifiuto. Ciascuna configurazione deve seguire dalla precedente in accordo alle regole specificate dalla funzione di transizione.

Una stringa può non essere una computazione di rifiuto in diversi modi: può non iniziare o terminare propriamente, o può essere scorretta da qualche parte nel mezzo. L'espressione R_2 risulta uguale a $R_{\text{bad-start}} \cup R_{\text{bad-window}} \cup R_{\text{bad-reject}}$, dove ciascuna sottoespressione corrisponde ad uno dei tre modi in cui una stringa può fallire.

Costruiamo l'espressione $R_{\text{bad-start}}$ per generare tutte le stringhe che non iniziano con la configurazione iniziale C_1 di M su w, come segue. La configurazione C_1 somiglia a $q_0 w_1 w_2 \cdots w_n \sqcup \cdots \sqcup \#$. Scriviamo $R_{\text{bad-start}}$ come l'unione di diverse sottoespressioni per gestire ciascuna parte di C_1:

$$R_{\text{bad-start}} = S_0 \cup S_1 \cup \cdots \cup S_n \cup S_b \cup S_\#.$$

L'espressione S_0 genera tutte le stringhe che non cominciano con q_0. Sia quindi S_0 l'espressione $\Delta_{-q_0}\,\Delta^*$. La notazione Δ_{-q_0} è un'abbreviazione per scrivere l'unione di tutti i simboli in Δ eccetto q_0.

L'espressione S_1 genera tutte le stringhe che non contengono w_1 nella seconda posizione. Sia quindi S_1 l'espressione $\Delta\,\Delta_{-w_1}\,\Delta^*$. In generale, per $1 \le i \le n$, l'espressione S_i è $\Delta^i\,\Delta_{-w_i}\,\Delta^*$. Quindi, S_i genera tutte le stringhe che contengono simboli qualsiasi nelle prime i posizioni, qualsiasi simbolo eccetto w_i in posizione $i+1$, e qualsiasi stringa di simboli dopo la posizione $i+1$. Si noti che in questo caso abbiamo usato l'operazione di esponenziazione. In realtà, a questo punto, l'esponenziazione è più una convenienza che una necessità perché avremmo invece potuto ripetere i volte il simbolo Δ senza incrementare eccessivamente la lunghezza dell'espressione. Ma nella sottoespressione seguente, l'esponenziazione è cruciale al fine di mantenere la dimensione polinomiale.

L'espressione S_b genera tutte le stringhe che non contengono un simbolo blank in qualche posizione da $n+2$ a $2^{(n^k)}$. Potremmo introdurre sottoespressioni S_{n+2} fino a $S_{2^{(n^k)}}$ per tale scopo, Ma poi l'espressione $R_{\text{bad-start}}$ avrebbe lunghezza esponenziale. Invece, definiamo

$$S_b = \Delta^{n+1}\,(\Delta \cup \varepsilon)^{2^{(n^k)}-n-2}\,\Delta_{-\sqcup}\,\Delta^*.$$

Pertanto, S_b genera tutte le stringhe che contengono simboli qualsiasi nelle prime $n+1$ posizioni, simboli qualsiasi nelle successive t posizioni, dove t può variare da 0 a $2^{(n^k)} - n - 2$, e qualsiasi simbolo eccetto blank nella posizione successiva.

Infine, $S_\#$ genera tutte le stringhe che non hanno un simbolo # in posizione $2^{(n^k)}+1$. Sia quindi $S_\#$ uguale a $\Delta^{(2^{(n^k)})}\,\Delta_{-\#}\,\Delta^*$.

Ora che abbiamo completato la costruzione di $R_{\text{bad-start}}$, rivolgiamo la nostra attenzione al pezzo successivo, $R_{\text{bad-reject}}$. Essa genera tutte le stringhe che non terminano propriamente; cioè, stringhe che non contengono una configurazione di rifiuto. Qualsiasi configurazione di rifiuto contiene lo stato q_{reject}, pertanto poniamo

$$R_{\text{bad-reject}} = \Delta^*_{-q_{\text{reject}}}.$$

Quindi, $R_{\text{bad-reject}}$ genera tutte le stringhe che non contengono q_{reject}.

Infine, costruiamo $R_{\text{bad-window}}$, l'espressione che genera tutte le stringhe per cui una configurazione non conduce correttamente alla configurazione successiva. Ricordiamo che nella dimostrazione del Teorema di Cook e Levin, abbiamo stabilito che una configurazione dà legalmente un'altra ogni volta che ogni tre simboli consecutivi nella prima configurazione danno correttamente i corrispondenti tre simboli nella seconda configurazione in accordo alla funzione di transizione. Quindi, se una configurazione non ne produce un'altra, l'errore sarà chiaro ad un'analisi dei sei simboli appropriati. Utilizziamo questa idea per costruire $R_{\text{bad-window}}$:

$$R_{\text{bad-window}} = \bigcup_{\text{bad}(abc,def)} \Delta^*\,abc\,\Delta^{(2^{(n^k)}-2)}\,def\,\Delta^*,$$

dove bad(abc, def) significa che abc non produce def in accordo alla funzione di transizione. L'unione è presa solo su tali simboli a, b, c, d, e, ed f in Δ. La figura che segue illustra la posizione di questi simboli nella storia di computazione.

Per calcolare la lunghezza di R_2 determiniamo la lunghezza degli esponenti che sono presenti in essa. Sono presenti diversi esponenti di grandezza circa $2^{(n^k)}$, e la loro lunghezza totale in binario è $O(n^k)$. Pertanto, la lunghezza di R_2 è polinomiale in n.

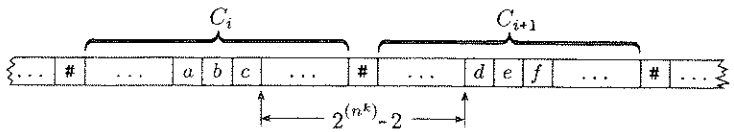


FIGURA 9.16
Posizioni corrispondenti in configurazioni adiacenti

9.2
RELATIVIZZAZIONE

La dimostrazione che $EQ_{\mathsf{REX}\uparrow}$ è intrattabile poggia sul metodo della diagonalizzazione. Perché non proviamo che *SAT* è intrattabile allo stesso modo? Forse potremmo usare la diagonalizzazione per mostrare che una TM non deterministica di tempo polinomiale può decidere un linguaggio di cui si può dimostrare la non appartenenza a P. In questa sezione introduciamo il metodo della *relativizzazione* per dare un'evidenza forte contro la possibilità di risolvere la questione P diverso da NP utilizzando una dimostrazione basata sulla diagonalizzazione.

Nel metodo della relativizzazione modifichiamo il nostro modello di computazione fornendo alla macchina di Turing alcune informazioni essenziali "gratuitamente." A seconda di quale informazione viene realmente fornita, la TM può essere in grado di risolvere alcuni problemi più facilmente di prima.

Per esempio, si supponga di concedere alla TM la capacità di risolvere il problema della soddisfacibilità in un singolo passo, per formule booleane di qualsiasi dimensione. Non importa come quest'azione venga realizzata - si immagini una "scatola nera" collegata che dà alla macchina questa capacità. Chiamiamo la scatola nera un *oracolo* per enfatizzare che essa non corrisponde necessariamente ad un dispositivo fisico. Ovviamente, la macchina potrebbe usare l'oracolo per risolvere qualsiasi problema in NP in tempo polinomiale, indipendentemente da se P è uguale a NP, perché ogni problema in NP è riducibile in tempo polinomiale al problema della soddisfacibilità. Una tale TM si dice che computa *relativamente al* problema della soddisfacibilità; da cui il termine *relativizzazione*.

In generale, un oracolo può corrispondere a qualsiasi linguaggio particolare, non soltanto al problema della soddisfacibilità. L'oracolo permette alla TM di verificare l'appartenenza al linguaggio senza dover realmente calcolare la risposta.

Formalizzeremo questa nozione tra breve. Probabilmente il lettore ricorda che abbiamo introdotto gli oracoli in Sezione 6.3. Lì li abbiamo definiti al fine di classificare i problemi in accordo al grado di irrisolvibilità. In questo caso utilizziamo gli oracoli per comprendere meglio il potere del metodo della diagonalizzazione.

DEFINIZIONE **9.17**

Un ***oracolo*** per un linguaggio A è un dispositivo che è capace di segnalare se una qualsiasi stringa w è un elemento di A. Una ***macchina di Turing con oracolo*** M^A è una macchina di Turing modificata che dispone della capacità aggiuntiva di interrogare un oracolo circa A. Ogni volta che M^A scrive una stringa sullo speciale ***nastro dell'oracolo***, la macchina sa se la stringa è un elemento di A in un singolo passo di computazione.

Sia P^A la classe dei linguaggi decidibili con una macchina di Turing di tempo polinomiale che usa l'oracolo A. Allo stesso modo definiamo la classe NP^A.

ESEMPIO 9.18

Come anticipato precedentemente, la computazione relativa al problema della soddisfacibilità contiene tutti i problemi in NP. In altre parole, $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$. Inoltre, $\mathrm{coNP} \subseteq \mathrm{P}^{SAT}$ perché P^{SAT}, essendo una classe di complessità deterministica, è chiusa rispetto alla complementazione.

ESEMPIO 9.19

Proprio come P^{SAT} contiene linguaggi che crediamo non siano in P, la classe NP^{SAT} contiene linguaggi che crediamo non siano in NP. Il complemento del linguaggio *MIN-FORMULA* che abbiamo definito nel Problema 7.21 a pagina 348 fornisce uno di tali esempi.

$\overline{\textit{MIN-FORMULA}}$ non sembra appartenere a NP (anche se la sua effettiva appartenenza a NP non è nota). Tuttavia, $\overline{\textit{MIN-FORMULA}}$ è in NP^{SAT}, perché una macchina di Turing non deterministica di tempo polinomiale con un oracolo per *SAT* può verificare se ϕ è un elemento come segue. Prima di tutto, il problema della non equivalenza di due formule booleane è risolvibile in NP e, quindi, il problema dell'equivalenza è in coNP perché una macchina non deterministica può trovare l'assegnamento su cui le due formule assumono valori differenti. Poi, la macchina non deterministica per $\overline{\textit{MIN-FORMULA}}$ individua non deterministicamente la più piccola formula equivalente, verifica se essa è realmente equivalente usando l'oracolo per *SAT* e accetta se lo è.

I limiti del metodo della diagonalizzazione

Il prossimo teorema presenta due oracoli, A e B, per i quali si può dimostrare che P^A ed NP^A sono differenti, mentre P^B ed NP^B sono uguali. Questi due oracoli sono importanti perché la loro esistenza indica che è improbabile risolvere la questione P diverso da NP attraverso il metodo della diagonalizzazione.

Il cuore del metodo della diagonalizzazione sta nella simulazione di una macchina di Turing tramite un'altra. La simulazione viene fatta in modo tale che la macchina simulante possa determinare il comportamento dell'altra macchina e poi comportarsi differentemente. Si supponga di dare a queste due macchine di Turing oracoli identici. Successivamente, ogni volta che la macchina simulata interroga l'oracolo, il simulatore fa lo stesso; e pertanto, la simulazione può procedere come in precedenza. Di conseguenza, qualsiasi teorema dimostrato per macchine di Turing utilizzando soltanto il metodo della diagonalizzazione continuerebbe ancora a valere qualora ad entrambe le macchine fosse dato lo stesso oracolo.

In particolare, se potessimo provare che P ed NP sono diversi attraverso la diagonalizzazione, potremmo concludere che sono differenti anche relativamente a qualsiasi oracolo. Ma P^B ed NP^B sono uguali, quindi una tale conclusione è falsa. Pertanto la diagonalizzazione non è sufficiente per separare queste due classi. Analogamente, nessuna dimostrazione che si basa su una semplice simulazione potrebbe dimostrare che le due classi sono le stesse perché tale risultato mostrerebbe che esse sono le stesse relativamente a qualsiasi oracolo; ma in realtà, P^A ed NP^A sono differenti.

TEOREMA 9.20

1. Esiste un oracolo A per cui $\mathrm{P}^A \neq \mathrm{NP}^A$.
2. Esiste un oracolo B per cui $\mathrm{P}^B = \mathrm{NP}^B$.

IDEA. L'oracolo B è facile da esibire. Sia B un qualsiasi problema PSPACE-completo, come per esempio $TQBF$.

L'oracolo A lo esibiamo per costruzione. Progettiamo A in modo tale che un determinato linguaggio L_A in NP^A richieda (in un modo che sia dimostrabile) una ricerca esaustiva, e quindi L_A non può stare in P^A. Pertanto possiamo concludere che $\mathrm{P}^A \neq \mathrm{NP}^A$. La costruzione prende in considerazione ogni macchina con oracolo di tempo polinomiale e garantisce che ciascuna di esse fallisca nel decidere il linguaggio L_A.

DIMOSTRAZIONE. Sia B il linguaggio $TQBF$. Sussistono le seguenti inclusioni

$$\mathrm{NP}^{TQBF} \overset{1}{\subseteq} \mathrm{NPSPACE} \overset{2}{\subseteq} \mathrm{PSPACE} \overset{3}{\subseteq} \mathrm{P}^{TQBF}.$$

L'inclusione 1 vale perché possiamo convertire la TM con oracolo non deterministica di tempo polinomiale in una macchina non deterministica di spazio polinomiale che computa le risposte alle interrogazioni riguardo a $TQBF$ invece di usare l'oracolo. L'inclusione 2 segue dal teorema di Savitch. L'inclusione 3 vale perché $TQBF$ è PSPACE-completo. Pertanto, risulta $\mathrm{P}^{TQBF} = \mathrm{NP}^{TQBF}$.

Nel seguito mostriamo come costruire l'oracolo A. Per ogni oracolo A, sia L_A la collezione di tutte le stringhe per cui esiste una stringa di eguale lunghezza che è presente in A. Quindi,

$$L_A = \{w \mid \exists x \in A \; [|x| = |w|]\}.$$

Ovviamente, per qualsiasi A, il linguaggio L_A appartiene ad NP^A.

Per mostrare che L_A non appartiene a P^A, progettiamo A come segue. Sia $M_1, M_2, \ldots$ una lista di tutte le TM con oracolo di tempo polinomiale. Possiamo assumere per semplicità che M_i computi in tempo n^i. La costruzione procede per fasi, dove la fase i costruisce una parte di A che garantisce che M_i^A non decide L_A. Costruiamo A dichiarando che determinate stringhe appartengono ad A mentre altre non vi appartengono. Ciascuna fase determina lo stato soltanto di un numero finito di stringhe. All'inizio non abbiamo alcuna informazione su A. Cominciamo con la fase 1.

Fase i. Fino ad ora, un numero finito di stringhe sono state dichiarate appartenenti o non appartenenti ad A. Scegliamo n maggiore della lunghezza di ciascuna di queste stringhe e abbastanza grande da far sì che 2^n risulti maggiore di n^i, il tempo di esecuzione di M_i. Mostriamo come estendere le nostre informazioni su A in modo tale che M_i^A accetti 1^n ogni volta che questa stringa non appartiene a L_A.

Mandiamo in esecuzione M_i sull'input 1^n e rispondiamo alle sue interrogazioni all'oracolo come segue. Se M_i usa per l'interrogazione una stringa y il cui stato già è stato determinato, rispondiamo consistentemente. Se lo stato di y è indeterminato, rispondiamo NO all'interrogazione e dichiariamo y non appartenente ad A. Continuiamo la simulazione di M_i fino a quando essa si arresta.

Guardiamo ora la situazione dal punto di vista di M_i. Se trova una stringa di lunghezza n in A, dovrebbe accettare perché sa che 1^n appartiene ad L_A. Se M_i stabilisce che tutte le stringhe di lunghezza n non appartengono ad A, dovrebbe rifiutare perché sa che 1^n non appartiene a L_A. Tuttavia, essa non dispone di tempo a sufficienza per chiedere su tutte le stringhe di lunghezza n, ed abbiamo risposto NO ad ognuna delle interrogazioni che ha fatto. Pertanto quando M_i si ferma e deve decidere se accettare o rifiutare, non possiede informazione a sufficienza per esser certa che la sua decisione risulti corretta.

Il nostro obiettivo è garantire che la sua decisione *non* sia corretta. Facciamo ciò osservando la sua decisione e poi estendendo A in modo tale che il contrario sia vero. Specificamente, se M_i accetta 1^n, dichiariamo tutte le stringhe rimanenti di lunghezza n non appartenenti ad A e quindi determiniamo che 1^n non appartiene a L_A. Se M_i rifiuta 1^n, troviamo una stringa di lunghezza n che M_i non ha usato per un'interrogazione e dichiariamo questa stringa appartenente ad A per garantire che 1^n appartenga ad L_A. Una tale stringa deve esistere perché M_i resta in esecuzione per n^i passi, che è meno di 2^n, il numero totale di stringhe di lunghezza n. In entrambi i casi abbiamo assicurato che M_i^A non decide L_A.

Completiamo la fase i dichiarando arbitrariamente che qualsiasi stringa di lunghezza al più n, il cui stato rimane indeterminato a questo punto, non appartiene ad A. La fase i è completa e procediamo con la fase $i+1$.

Abbiamo mostrato che nessuna TM con oracolo di tempo polinomiale decide L_A con l'oracolo A, provando con ciò il teorema.

In conclusione, il metodo della relativizzazione ci dice che per risolvere la questione P diverso da NP dobbiamo *analizzare* le computazioni, non meramente simularle. In Sezione 9.3 introdurremo un approccio che potrebbe condurre ad un'analisi di questo tipo.

9.3 COMPLESSITÀ DEI CIRCUITI

I calcolatori vengono costruiti a partire da congegni elettronici cablati assieme in un progetto chiamato *circuito digitale*. Possiamo anche simulare modelli teorici, come le macchine di Turing, con la controparte teorica dei circuiti digitali, chiamata *circuiti booleani*. Stabilendo la connessione tra TM e circuiti booleani si raggiungono due scopi. In primo luogo, i ricercatori credono che i circuiti forniscano un modello computazionale conveniente per attaccare la questione P diverso da NP e questioni collegate. In secondo luogo, i circuiti forniscono una prova alternativa del teorema di Cook e Levin che *SAT* è NP-completo. Copriremo entrambi gli argomenti in questa sezione.

DEFINIZIONE **9.21**

Un ***circuito booleano*** è una collezione di ***porte*** e ***input*** connessi da ***fili***. I cicli non sono permessi. Le porte assumono tre forme: porta AND, porta OR, e porta NOT, come mostrato schematicamente nella figura seguente.

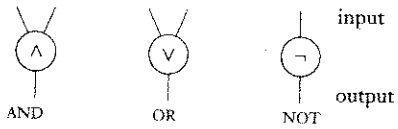


FIGURA 9.22
Una porta AND, una porta OR, ed una porta NOT

I fili in un circuito booleano trasportano i valori booleani 0 e 1. Le porte sono semplici processori che computano le funzioni booleane AND, OR, e NOT. La funzione AND dà in output 1 se entrambi i suoi input sono 1 e dà in output 0 altrimenti. La funzione OR dà in output 0 se entrambi i suoi input sono 0 e dà in output 1 altrimenti. La funzione NOT dà in output l'opposto del suo input; in altre parole, dà in output un 1 se il suo input è 0 ed uno 0 se il suo input è 1. Gli input sono etichettati con $x_1, \ldots, x_n$. Una delle porte viene designata come la ***porta di output***. La figura seguente rappresenta un circuito booleano.

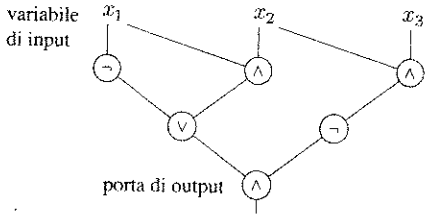


FIGURA 9.23
Un esempio di circuito booleano

Un circuito booleano calcola un valore di output a partire da una specifica degli input, propagando i valori lungo i fili e calcolando la funzione associata alle rispettive porte fino a quando viene assegnato un valore alla porta di output. La figura seguente mostra un circuito booleano che calcola un valore a partire da una specifica dei suoi input.

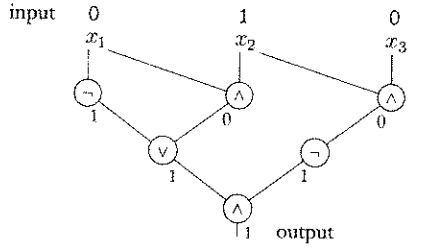


FIGURA 9.24
Un esempio di calcolo di un circuito booleano

Utilizziamo le funzioni per descrivere il comportamento input/output dei circuiti booleani. Associamo ad un circuito booleano C con n variabili di input, una funzione $f_C: \{0,1\}^n \longrightarrow \{0,1\}$ dove, se C dà in output b quando i suoi input $x_1, \ldots, x_n$ assumono i valori $a_1, \ldots, a_n$, scriviamo $f_C(a_1, \ldots, a_n) = b$. Diciamo che C calcola la funzione f_C. Qualche volta considereremo circuiti booleani che hanno porte di output multiple. Una funzione con k bit di output calcola una funzione il cui range è $\{0,1\}^k$.

ESEMPIO 9.25

La ***funzione di parità*** con n input $parity_n: \{0,1\}^n \longrightarrow \{0,1\}$ dà in output 1 se un numero dispari di 1 compare nelle variabili di input. Il circuito in Figura 9.26 calcola $parity_4$, la funzione di parità su 4 variabili.

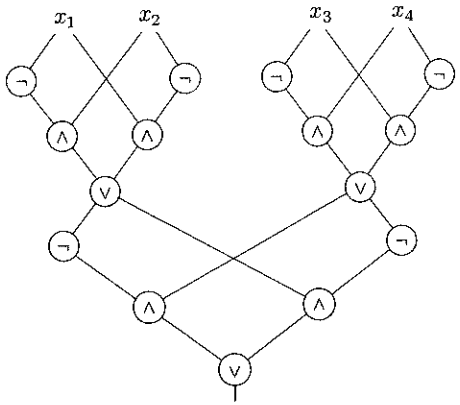


FIGURA 9.26
Un circuito booleano che calcola la funzione di parità su 4 variabili

Utilizzeremo di usare i circuiti per verificare l'appartenenza ai linguaggi, una volta che siano stati opportunamente codificati in $\{0,1\}$. Un problema che sorge è che ogni particolare circuito può gestire soltanto input di una qualche lunghezza fissata, mentre un linguaggio può contenere stringhe di differenti lunghezze. Pertanto, per adempiere a questo compito, invece di usare un singolo circuito per verificare l'appartenenza ad un linguaggio, utilizzeremo una intera ***famiglia*** di circuiti, uno per ogni lunghezza dell'input. Formalizziamo questa nozione nella definizione che segue.

DEFINIZIONE **9.27**

Una ***famiglia di circuiti*** C è una lista infinita di circuiti, $(C_0, C_1, C_2, \ldots)$, dove C_n ha n variabili di input. Diciamo che C decide un linguaggio A su $\{0,1\}$ se, per ogni stringa w,

$$w \in A \quad \text{se e solo se} \quad C_n(w) = 1,$$

dove n è la lunghezza di w.

La ***taglia*** di un circuito è il numero di porte che esso contiene. Due circuiti sono equivalenti se hanno lo stesso numero di variabili di input e danno in output lo stesso valore per tutti gli assegnamenti alle variabili di input. Un circuito è di ***taglia minimale*** se nessun circuito più piccolo è equivalente ad esso. Il problema della minimizzazione dei circuiti ha ovvie applicazioni ingegneristiche ma è molto difficile da risolvere in generale. Anche il problema di verificare se un circuito particolare è minimale non sembra essere risolvibile in P o in NP. Una famiglia di circuiti è minimale se ogni C_i nella lista è un circuito minimale. La ***complessità di taglia*** di una famiglia di circuiti $(C_0, C_1, C_2, \ldots)$ è la funzione $f: \mathcal{N} \longrightarrow \mathcal{N}$, dove $f(n)$ è la taglia di C_n. Possiamo far riferimento semplicemente alla complessità di una famiglia di circuiti, invece che alla complessità di taglia, quando risulta chiaro che stiamo parlando della taglia.

La ***profondità*** di un circuito è la lunghezza (numero di fili) del cammino più lungo da una variabile di input alla porta di output. Definiamo circuiti e famiglie di circuiti di ***minima profondità*** e la ***complessità di profondità*** di famiglie di circuiti, così come abbiamo fatto con la taglia dei circuiti. La complessità di profondità dei circuiti è di particolare interesse nella Sezione 10.5 che riguarda il calcolo parallelo.

DEFINIZIONE **9.28**

La ***complessità di circuito*** di un linguaggio è la complessità di taglia di una famiglia di circuiti minimale per quel linguaggio. La ***complessità di profondità di circuito*** di un linguaggio viene definita analogamente, utilizzando la profondità invece della taglia.

ESEMPIO 9.29

Possiamo facilmente generalizzare l'Esempio 9.25 per produrre circuiti che computano la funzione di parità su n variabili con $O(n)$ porte. Un modo per far ciò è costruire un albero binario di porte che calcola la funzione XOR, dove la funzione XOR coincide con la funzione $parity_2$, e poi implementare

ciascuna porta XOR con due NOT, due AND, ed un OR, come abbiamo fatto nell'esempio precedente.

Sia A il linguaggio di stringhe che contengono un numero dispari di 1. Allora A ha complessità di circuito $O(n)$.

La complessità di circuito di un linguaggio è connessa alla sua complessità temporale. Qualsiasi linguaggio con una complessità di tempo piccola ha anche una complessità di circuito piccola, come stabilisce il teorema seguente.

TEOREMA 9.30

Sia $t: \mathcal{N} \longrightarrow \mathcal{N}$ una funzione, dove $t(n) \geq n$. Se $A \in \text{TIME}(t(n))$, allora A ha complessità di circuito $O(t^2(n))$.

Questo teorema fornisce un approccio per provare che P $\neq$ NP, attraverso il quale cerchiamo di dimostrare che un qualche linguaggio in NP ha complessità di circuito più che polinomiale.

IDEA. Sia M una TM che decide A in tempo $t(n)$. (Per semplicità, ignoriamo il fattore costante in $O(t(n))$, il tempo di esecuzione reale di M.) Per ogni n, costruiamo un circuito C_n che simula M su input di lunghezza n. Le porte di C_n sono organizzate per righe, una per ciascuno dei $t(n)$ passi nel calcolo di M su un input di lunghezza n. Ogni riga di porte rappresenta la configurazione di M al passo corrispondente. Ogni riga è collegata alla riga precedente affinché possa calcolare la sua configurazione a partire dalla configurazione della riga precedente. Modifichiamo M in modo tale che l'input sia codificato in {0,1}. Inoltre, quando M sta per accettare, facciamo sì che essa muova la sua testina sulla cella del nastro più a sinistra e scriva il simbolo ␣ su detta cella prima di entrare nello stato di accettazione. In questo modo possiamo designare una porta nella riga finale del circuito come porta di output.

DIMOSTRAZIONE. Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ una macchina che decide A in tempo $t(n)$, e sia w un input di lunghezza n per M. Definiamo un ***tableau*** per M su w come una tabella di dimensione $t(n) \times t(n)$, le cui righe sono configurazioni di M. La riga superiore del tableau contiene la configurazione iniziale di M su w. La i-esima riga contiene la configurazione al passo i-esimo del calcolo.

Per comodità in questa dimostrazione modifichiamo il formato di rappresentazione delle configurazioni. Invece del vecchio formato, descritto a pagina 176, dove lo stato compare alla sinistra del simbolo che la testina sta leggendo, rappresentiamo sia lo stato che il simbolo di nastro sotto la testina del nastro attraverso un singolo carattere composto. Per esempio, se M si trovasse nello stato q ed il suo nastro contenesse la stringa 1011 con la testina che legge il secondo simbolo da sinistra, il vecchio formato

sarebbe 1q011 mentre il nuovo sarebbe 1[q0]11 - dove il carattere composto [q0] rappresenta sia q, lo stato, sia 0, il simbolo sotto la testina.

Ogni cella del tableau può contenere un simbolo di nastro (appartenente a Γ) o una combinazione di uno stato e di un simbolo di nastro (appartenente a $Q \times \Gamma$). La cella definita dalla riga i-esima e dalla colonna j-esima del tableau è $cell[i,j]$. La riga superiore del tableau è allora $cell[1,1], \ldots, cell[1,t(n)]$ e contiene la configurazione iniziale.

Facciamo due assunzioni sulla TM M nel definire la nozione di tableau. Prima di tutto, come anticipato nell'idea della prova, M accetta solo quando la sua testina si trova sulla cella più a sinistra del nastro e detta cella contiene il simbolo ␣. In secondo luogo, una volta che M si è arrestata, resta nella stessa configurazione per tutti i passi futuri. Pertanto, guardando la cella più a sinistra nella riga finale del tableau, $cell[t(n),1]$, possiamo stabilire se M ha accettato. La figura seguente mostra parte del tableau per M sull'input 0010.

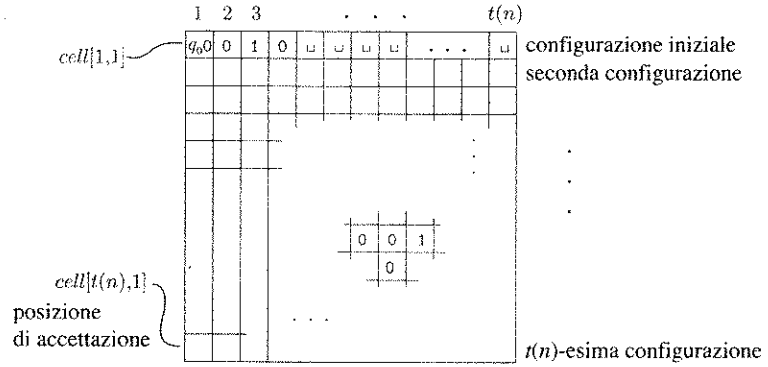
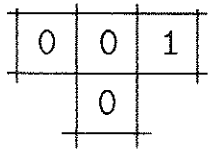


FIGURA 9.31
Un tableau per M su input 0010

Il contenuto di ciascuna cella viene determinato da celle specifiche della riga precedente. Se conosciamo i valori di $cell[i-1,j-1]$, $cell[i-1,j]$, e di $cell[i-1,j+1]$, possiamo ottenere il valore di $cell[i,j]$ attraverso la funzione di transizione di M. Per esempio, la figura che segue ingrandisce una porzione del tableau in Figura 9.31. I tre simboli superiori, 0, 0, ed 1, sono simboli di nastro senza stati, pertanto il simbolo centrale deve rimanere uno 0 nella riga successiva, come mostrato.



A questo punto possiamo iniziare a costruire il circuito C_n. Esso dispone di diverse porte per ogni cella nel tableau. Queste porte calcolano il valore di una cella dai valori delle tre celle che hanno influenza su di essa.

Al fine di rendere la costruzione più semplice da descrivere, aggiungiamo luci che mostrano l'output di alcune delle porte nel circuito. Le luci hanno solo scopi illustrativi e non influenzano le operazioni del circuito.

Sia k il numero di elementi in $\Gamma \cup (Q \times \Gamma)$. Creiamo k luci per ogni cella nel tableau - una luce per ogni elemento di Γ, ed una luce per ogni elemento di $(Q \times \Gamma)$ - ovvero un totale di $kt^2(n)$ luci. Chiamiamo queste luci $light[i, j, s]$, dove $1 \leq i, j \leq t(n)$ ed $s \in \Gamma \cup (Q \times \Gamma)$. La condizione delle luci in una cella indica il contenuto di quella cella. Se $light[i, j, s]$ è accesa, $cell[i, j]$ contiene il simbolo s. Naturalmente, se il circuito viene costruito opportunamente, per ogni cella dovrebbe essere accesa una sola luce.

Prendiamo una delle luci - diciamo, $light[i, j, s]$ in $cell[i, j]$. Questa luce dovrebbe essere accesa se la cella contiene il simbolo s. Consideriamo le tre celle che influenzano $cell[i, j]$ e determiniamo quali tra i loro possibili assegnamenti portano $cell[i, j]$ a contenere s. Questa individuazione può essere effettuata esaminando la funzione di transizione δ.

Supponiamo che se le celle $cell[i-1, j-1]$, $cell[i-1, j]$, e $cell[i-1, j+1]$ contengono a, b, e c, rispettivamente, $cell[i, j]$ contiene s, in accordo a δ. Cabliamo il circuito in modo tale che se $light[i-1, j-1, a]$, $light[i-1, j, b]$, e $light[i-1, j+1, c]$ sono accese, allora è accesa anche $light[i, j, s]$. Facciamo ciò connettendo le tre luci al livello $i-1$ ad una porta AND il cui output è connesso a $light[i, j, s]$.

In generale, molteplici assegnamenti differenti (a_1, b_1, c_1), (a_2, b_2, c_2), ..., (a_l, b_l, c_l) per $cell[i-1, j-1]$, $cell[i-1, j]$, e $cell[i-1, j+1]$ possono portare $cell[i, j]$ a contenere s. In questo caso, cabliamo il circuito in modo tale che per ogni assegnamento a_i, b_i, c_i, le rispettive luci siano connesse ad una porta AND e tutte le porte AND siano connesse ad una porta OR. Questa circuiteria è illustrata nella figura seguente.

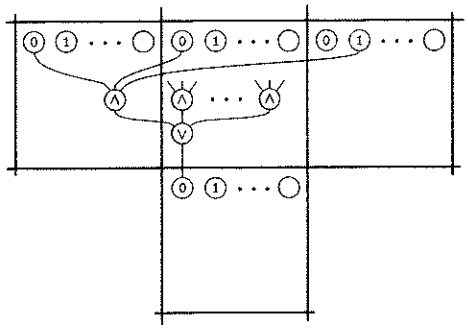


FIGURA 9.32
Circuiteria per una luce

La circuiteria appena descritta viene ripetuta per ogni luce, con poche eccezioni ai confini. Ogni cella al confine sinistro del tableau - cioè, $cell[i, 1]$ per $1 \leq i \leq t(n)$ - ha solo due celle precedenti che influenzano il suo contenuto. Le celle al confine destro sono simili. In questi casi modifichiamo la circuiteria per simulare il comportamento della TM M in questa situazione.

Le celle nella prima riga non hanno predecessori e vengono gestite in modo speciale. Queste celle contengono la configurazione iniziale e le loro luci sono cablate con le variabili di input. Pertanto, $light[1, 1, \boxed{q_0 1}]$ è connessa all'input w_1 perché la configurazione iniziale comincia con il simbolo dello stato iniziale q_0 e con la testina su w_1. Analogamente, $light[1, 1, \boxed{q_0 0}]$ è connessa tramite una porta NOT all'input w_1. Inoltre, $light[1, 2, 1], \ldots, light[1, n, 1]$ sono connesse agli input $w_2, \ldots, w_n$, e $light[1, 2, 0], \ldots, light[1, n, 0]$ sono connessi attraverso porte NOT agli input $w_2, \ldots, w_n$ poiché la stringa di input w determina questi valori. In aggiunta, $light[1, n+1, \sqcup], \ldots, light[1, t(n), \sqcup]$ sono accese perché le celle rimanenti nella prima riga corrispondono alle posizioni sul nastro che inizialmente sono $\sqcup$. Infine, tutte le altre luci nella prima riga sono spente.

Finora abbiamo costruito un circuito che simula M fino al suo passo $t(n)$-esimo. Ciò che resta da fare è designare una delle porte come porta di output del circuito. Sappiamo che M accetta w se si trova in uno stato di accettazione q_{accept} su una cella contenente $\sqcup$ all'estremità sinistra del nastro al passo $t(n)$. Pertanto, designiamo come porta di output la porta connessa a $light[t(n), 1, \boxed{q_{\text{accept}}\sqcup}]$. La dimostrazione del teorema è così completa.

Oltre a legare la complessità di circuito alla complessità di tempo, il Teorema 9.30 dà una dimostrazione alternativa del Teorema 7.27, il teorema di Cook e Levin, come segue. Diciamo che un circuito booleano è ***soddisfacibile*** se esiste un assegnamento per l'input che porta il circuito a dare in output 1. Il problema della ***soddisfacibilità dei circuiti*** decide se un circuito è soddisfacibile. Sia

$$CIRCUIT\text{-}SAT = \{\langle C \rangle \mid C \text{ è un circuito booleano soddisfacibile}\}.$$

Il Teorema 9.30 mostra che i circuiti booleani hanno la capacità di simulare le macchine di Turing. Usiamo questo risultato per provare che *CIRCUIT-SAT* è NP-completo.

TEOREMA 9.33

CIRCUIT-SAT è NP-completo.

DIMOSTRAZIONE. Per dimostrare il teorema dobbiamo mostrare che *CIRCUIT-SAT* appartiene a NP, e che qualsiasi linguaggio A appartenente a NP è riducibile a *CIRCUIT-SAT*. La prima affermazione è ovvia. Per la seconda, dobbiamo fornire una riduzione di tempo polinomiale f che associa a stringhe circuiti, dove

$$f(w) = \langle C \rangle$$

implica che

$$w \in A \iff \text{il circuito booleano } C \text{ è soddisfacibile.}$$

Poiché appartiene a NP, A possiede un verificatore di tempo polinomiale V il cui input ha la forma $\langle x, c \rangle$, dove c può essere il certificato che mostra che x appartiene ad A. Per costruire f, ricaviamo il circuito che simula V usando il metodo del Teorema 9.30. Associamo agli input del circuito che corrispondono a x i simboli di w. I soli input rimanenti del circuito corrispondono al certificato c. Chiamiamo questo circuito C e lo diamo in output.

Se C è soddisfacibile, esistee un certificato, e quindi w appartiene ad A. Al contrario, se w appartiene ad A, un certificato esiste, e quindi C è soddisfacibile.

Per far vedere che questa riduzione ha tempo di esecuzione polinomiale, osserviamo che nella dimostrazione del Teorema 9.30 la costruzione del circuito può essere fatta in tempo polinomiale in n. Il tempo di esecuzione del verificatore è n^k per qualche k, pertanto la taglia del circuito costruito è $O(n^{2k})$. La struttura del circuito è abbastanza semplice (in realtà essa è altamente ripetitiva), pertanto il tempo di esecuzione della riduzione è $O(n^{2k})$.

Mostriamo ora che *3SAT* è NP-completo, completando la dimostrazione alternativa del teorema di Cook e Levin.

TEOREMA 9.34

3SAT è NP-completo.

IDEA. *3SAT* appartiene ovviamente a NP. Mostriamo che tutti i linguaggi appartenenti a NP si riducono a *3SAT* in tempo polinomiale. Lo facciamo riducendo *CIRCUIT-SAT* a *3SAT* in tempo polinomiale. La riduzione converte un circuito C in una formula ϕ, per cui C è soddisfacibile se e solo se ϕ è soddisfacibile. La formula contiene una variabile per ciascuna variabile e per ciascuna porta del circuito.

Concettualmente la formula simula il circuito. Un assegnamento che soddisfa ϕ contiene un assegnamento che soddisfa C. Esso contiene anche i valori di ciascuna delle porte di C durante il calcolo di C su un suo assegnamento. In effetti, l'assegnamento che soddisfa ϕ "individua" l'intera computazione di C sul suo assegnamento, e le clausole di ϕ verificano la correttezza della computazione. In aggiunta, ϕ contiene una clausola che impone che l'output di C sia 1.

DIMOSTRAZIONE. Diamo una riduzione di tempo polinomiale f da *CIRCUIT-SAT* a *3SAT*. Sia C un circuito contenente gli input $x_1, \ldots, x_l$ e le porte $g_1, \ldots, g_m$. La riduzione costruisce a partire da C una formula ϕ con variabili $x_1, \ldots, x_l, g_1, \ldots, g_m$. Ogni variabile di ϕ corrisponde ad un filo in C. Le variabili x_i corrispondono ai fili di input, e le variabili g_i corrispondono ai fili di output delle porte. Rinominiamo le variabili di ϕ come $w_1, \ldots, w_{l+m}$.

Descriviamo ora le clausole di ϕ. Scriviamo le clausole di ϕ più intuitivamente usando le implicazioni. Ricordiamo che possiamo convertire l'operatore di implicazione $(P \to Q)$ nella clausola $(\overline{P} \vee Q)$. Ogni porta NOT in C con filo di input w_i e filo di output w_j è equivalente all'espressione

$$(\overline{w_i} \to w_j) \wedge (w_i \to \overline{w_j}),$$

che a sua volta dà le due clausole

$$(w_i \vee w_j) \wedge (\overline{w_i} \vee \overline{w_j}).$$

Si osservi che entrambe le clausole sono soddisfatte se e solo se viene fatto un assegnamento alle variabili w_i e w_j che corrisponde alla funzionalità corretta della porta NOT.

Ogni porta AND in C con input w_i e w_j ed output w_k è equivalente a

$$((\overline{w_i} \wedge \overline{w_j}) \to \overline{w_k}) \wedge ((\overline{w_i} \wedge w_j) \to \overline{w_k}) \wedge ((w_i \wedge \overline{w_j}) \to \overline{w_k}) \wedge ((w_i \wedge w_j) \to w_k),$$

che a sua volta dà le quattro clausole

$$(w_i \vee w_j \vee \overline{w_k}) \wedge (w_i \vee \overline{w_j} \vee \overline{w_k}) \wedge (\overline{w_i} \vee w_j \vee \overline{w_k}) \wedge (\overline{w_i} \vee \overline{w_j} \vee w_k).$$

Analogamente, ogni porta OR in C con input w_i e w_j ed output w_k è equivalente a

$$((\overline{w_i} \wedge \overline{w_j}) \to \overline{w_k}) \wedge ((\overline{w_i} \wedge w_j) \to w_k) \wedge ((w_i \wedge \overline{w_j}) \to w_k) \wedge ((w_i \wedge w_j) \to w_k),$$

che a sua volta dà le quattro clausole

$$(w_i \vee w_j \vee \overline{w_k}) \wedge (w_i \vee \overline{w_j} \vee w_k) \wedge (\overline{w_i} \vee w_j \vee w_k) \wedge (\overline{w_i} \vee \overline{w_j} \vee w_k).$$

In entrambi i casi, tutte e quattro le clausole sono soddisfatte quando viene effettuato un assegnamento alle variabili w_i, w_j, e w_k, che corrisponde alla corretta funzionalità della porta. Inoltre, aggiungiamo la clausola (w_m) a ϕ, dove w_m è la porta di output di C.

Alcune delle clausole descritte contengono meno di tre letterali. Espandiamo queste clausole alla dimensione desiderata ripetendo letterali. Per esempio, espandiamo la clausola (w_m) alla clausola equivalente $(w_m \vee w_m \vee w_m)$. Ciò completa la costruzione.

Proviamo brevemente che la costruzione funziona. Se esiste un assegnamento che soddisfa C, otteniamo un assegnamento che soddisfa ϕ determinando il valore delle variabili g_i in accordo alla computazione di C su questo assegnamento. Viceversa, se esiste un assegnamento che soddisfa ϕ, esso fornisce un assegnamento per C perché descrive l'intera computazione di C dove il valore di output è 1. La riduzione può esser fatta in tempo polinomiale perché è semplice da calcolare e la dimensione dell'output è polinomiale (in realtà lineare) nella taglia dell'input.

ESERCIZI

[A]9.1 Si provi che $\text{TIME}(2^n) = \text{TIME}(2^{n+1})$.

[A]9.2 Si provi che $\text{TIME}(2^n) \subsetneq \text{TIME}(2^{2n})$.

[A]9.3 Si provi che $\text{NTIME}(n) \subsetneq \text{PSPACE}$.

9.4 Si faccia vedere come il circuito riportato in Figura 9.26 computa sull'input 0110, mostrando i valori calcolati da tutte le porte, come fatto in Figura 9.24.

9.5 Si fornisca un circuito che calcola la funzione di parità su tre variabili di input e si mostri come computa sull'input 011.

9.6 Si provi che se $A \in \text{P}$, allora $\text{P}^A = \text{P}$.

9.7 Si forniscano espressioni regolari con esponenziazione che generano i seguenti linguaggi sull'alfabeto $\{0,1\}$.

- **[A]a.** Tutte le stringhe di lunghezza 500
- **[A]b.** Tutte le stringhe di lunghezza 500 o meno
- **[A]c.** Tutte le stringhe di lunghezza 500 o più
- **[A]d.** Tutte le stringhe di lunghezza diversa da 500
- **e.** Tutte le stringhe che contengono esattamente 500 simboli 1
- **f.** Tutte le stringhe che contengono almeno 500 simboli 1
- **g.** Tutte le stringhe che contengono al più 500 simboli 1
- **h.** Tutte le stringhe di lunghezza 500 o più che contengono uno 0 nella 500-esima posizione
- **i.** Tutte le stringhe che contengono due 0 che hanno almeno 500 simboli tra loro

9.8 Se R è un'espressione regolare, sia $R^{\{m,n\}}$ la rappresentazione dell'espressione

$$R^m \cup R^{m+1} \cup \cdots \cup R^n.$$

Si mostri come implementare l'operatore $R^{\{m,n\}}$ usando l'operatore ordinario per l'esponenziazione, ma senza "$\cdots$".

9.9 Si mostri che se $\text{NP} = \text{P}^{SAT}$, allora $\text{NP} = \text{coNP}$.

9.10 Il Problema 8.29 ha mostrato che A_{LBA} è PSPACE-completo.

- **a.** Sappiamo se $A_{\text{LBA}} \in \text{NL}$? Si motivi la risposta.
- **b.** Sappiamo se $A_{\text{LBA}} \in \text{P}$? Si motivi la risposta.

9.11 Si mostri che il linguaggio *MAX-CLIQUE* dal Problema 7.19 appartiene a P^{SAT}.

PROBLEMI

9.12 Ricordiamo che è possibile considerare circuiti che danno in output stringhe su $\{0,1\}$ utilizzando molteplici porte di output. Si assuma che $add_n\colon \{0,1\}^{2n} \longrightarrow \{0,1\}^{n+1}$ prenda due interi binari di n bit e produca la somma di $n+1$ bit. Si mostri che è possibile calcolare la funzione add_n con circuiti di taglia $O(n)$.

9.13 Si definisca la funzione $majority_n\colon \{0,1\}^n \longrightarrow \{0,1\}$ come

$$majority_n(x_1, \ldots, x_n) = \begin{cases} 0 & \sum x_i < n/2; \\ 1 & \sum x_i \geq n/2. \end{cases}$$

Quindi, la funzione $majority_n$ restituisce il voto di maggioranza degli input. Si mostri che $majority_n$ può essere calcolata con:

- **a.** Circuiti di taglia $O(n^2)$.
- **b.** Circuiti di taglia $O(n \log n)$. (Suggerimento: si divida ricorsivamente il numero di input a metà e si usi il risultato del Problema 9.12.)

***9.14** Si definisca la funzione $majority_n$ come nel Problema 9.13. Si mostri come potrebbe essere calcolata con circuiti di taglia $O(n)$.

9.15 Siano A e B due oracoli. Uno di essi è un oracolo per $TQBF$, ma non si sa quale dei due. Si fornisca un algoritmo che abbia accesso sia ad A che a B, e che garantisca di risolvere $TQBF$ in tempo polinomiale.

9.16 Si provi che esiste un oracolo C per il quale $\text{NP}^C \neq \text{coNP}^C$.

9.17 Una ***macchina di Turing con oracolo a k-query*** è una macchina di Turing con oracolo a cui è permesso di effettuare al più k interrogazioni su ogni input. Una macchina di Turing a k-query M con un oracolo per A è denotata con $M^{A,k}$. Si definisca $\text{P}^{A,k}$ come la collezione di linguaggi che sono decidibili da macchine di Turing a k-query con un oracolo per A di tempo polinomiale.

- **a.** Si mostri che $\text{NP} \cup \text{coNP} \subseteq \text{P}^{SAT,1}$.
- **b.** Si assuma che $\text{NP} \neq \text{coNP}$. Si mostri che $\text{NP} \cup \text{coNP} \subsetneq \text{P}^{SAT,1}$.

9.18 Si definisca il problema ***unique-sat*** come

$$USAT = \{\langle\phi\rangle \mid \phi \text{ è una formula booleana che ha un singolo assegnamento che la soddisfa}\}.$$

Si mostri che $USAT \in \mathrm{P}^{SAT}$.

9.19 Sia $E_{\mathsf{REX}\uparrow} = \{\langle R\rangle \mid R \text{ è una espressione regolare con esponenziazione ed } L(R) = \emptyset\}$. Si mostri che $E_{\mathsf{REX}\uparrow} \in \mathrm{P}$.

9.20 Si descriva l'errore nella "dimostrazione" fallace seguente che P $\neq$ NP. Si assume che P = NP e si ottiene una contraddizione. Se P = NP, allora $SAT \in$ P e, quindi, per qualche k, $SAT \in \mathrm{TIME}(n^k)$. Poiché ogni linguaggio in NP è riducibile in tempo polinomiale a SAT, risulta NP $\subseteq \mathrm{TIME}(n^k)$. Quindi, P $\subseteq \mathrm{TIME}(n^k)$. Ma, per il teorema di gerarchia di tempo, $\mathrm{TIME}(n^{k+1})$ contiene un linguaggio che non appartiene a $\mathrm{TIME}(n^k)$, il che contraddice P $\subseteq \mathrm{TIME}(n^k)$. Pertanto, P $\neq$ NP.

9.21 Si consideri la funzione $pad\colon \Sigma^* \times \mathcal{N} \longrightarrow \Sigma^*\#^*$ che è definita come segue. Sia $pad(s, l) = s\#^j$, dove $j = \max(0, l - m)$ ed m è la lunghezza di s. Pertanto, $pad(s, l)$ aggiunge semplicemente abbastanza copie del nuovo simbolo # alla fine di s, in modo tale che la lunghezza del risultato sia almeno l. Per ogni linguaggio A e funzione $f\colon \mathcal{N} \longrightarrow \mathcal{N}$, si definisca il linguaggio $pad(A, f)$ come

$$pad(A, f) = \{pad(s, f(m)) \mid \text{dove } s \in A \text{ ed } m \text{ è la lunghezza di } s\}.$$

Si provi che se $A \in \mathrm{TIME}(n^6)$, allora $pad(A, n^2) \in \mathrm{TIME}(n^3)$.

9.22 Si provi che se NEXPTIME $\neq$ EXPTIME, allora P $\neq$ NP. Potrebbe essere utile la funzione *pad*, definita nel Problema 9.21.

A9.23 Si definisca *pad* come nel Problema 9.21.

a. Si provi che per ogni A e numero naturale k, $A \in$ P se e solo se $pad(A, n^k) \in$ P.

b. Si provi che P $\neq$ SPACE(n).

9.24 Si provi che $TQBF \notin \mathrm{SPACE}(n^{1/3})$.

***9.25** Si legga la definizione di 2DFA (automa finito a due testine) data nel Problema 5.14. Si provi che P contiene un linguaggio che non è riconoscibile da un 2DFA.

SOLUZIONI SELEZIONATE

9.1 Le classi di complessità sono definite in termini della notazione O grande, quindi i fattori costanti non hanno effetti. La funzione 2^{n+1} è $O(2^n)$ e pertanto $A \in \mathrm{TIME}(2^n)$ se e solo se $A \in \mathrm{TIME}(2^{n+1})$.

9.2 L'inclusione $\mathrm{TIME}(2^n) \subseteq \mathrm{TIME}(2^{2n})$ vale perché $2^n \leq 2^{2n}$. L'inclusione è propria in virtù del teorema di gerarchia di tempo. La funzione 2^{2n} è tempo-costruibile perché una TM può scrivere il numero 1 seguito da $2n$ 0 in tempo $O(2^{2n})$. Quindi il teorema garantisce che esiste un linguaggio A che può essere deciso in tempo $O(2^{2n})$ ma non in tempo $o(2^{2n}/\log 2^{2n}) = o(2^{2n}/2n)$. Pertanto, $A \in \mathrm{TIME}(2^{2n})$ ma $A \notin \mathrm{TIME}(2^n)$.

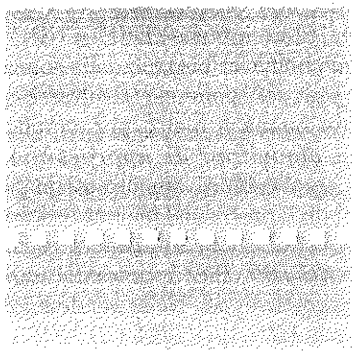
9.3 NTIME(n) $\subseteq$ NSPACE(n) perché qualsiasi macchina di Turing che opera in tempo $t(n)$ su ogni diramazione della computazione può usare al più $t(n)$ celle del nastro su ogni diramazione. Inoltre, NSPACE(n) $\subseteq$ SPACE(n^2) per il teorema di Savitch. D'altronde, SPACE(n^2) $\subsetneq$ SPACE(n^3) per il teorema di gerarchia di spazio. Il risultato segue perché SPACE(n^3) $\subseteq$ PSPACE.

9.7 **(a)** Σ^{500}; **(b)** $(\Sigma \cup \varepsilon)^{500}$; **(c)** $\Sigma^{500}\Sigma^*$; **(d)** $(\Sigma \cup \varepsilon)^{499} \cup \Sigma^{501}\Sigma^*$.

9.23 **(a)** Sia A un qualsiasi linguaggio e sia $k \in \mathcal{N}$. Se $A \in$ P, allora $pad(A, n^k) \in$ P perché è possibile stabilire se $w \in pad(A, n^k)$ scrivendo w come $s\#^t$, dove s non contiene il simbolo #, verificando poi se $|w| = |s|^k$; ed, infine, verificando se $s \in A$. L'implementazione del primo test in tempo polinomiale è immediata. Il secondo test viene eseguito in tempo poly($|s|$), e poiché $|s| \leq |w|$, il test viene eseguito in tempo poly($|w|$) e quindi in tempo polinomiale. Se $pad(A, n^k) \in$ P, allora $A \in$ P perché è possibile stabilire se $w \in A$ estendendo w con simboli # fino a quando ha lunghezza $|w|^k$ e poi verificando se il risultato appartiene a $pad(A, n^k)$. Entrambe queste azioni richiedono soltanto tempo polinomiale.

(b) Si assuma che P = SPACE(n). Sia A un linguaggio appartenente a SPACE(n^2) ma non appartenente a SPACE(n), la cui esistenza è stata mostrata nel teorema di gerarchia di spazio. Il linguaggio $pad(A, n^2) \in$ SPACE(n) perché è disponibile spazio a sufficienza per eseguire l'algoritmo di spazio $O(n^2)$ per A, usando spazio che è lineare nel linguaggio esteso. Per via dell'assunzione $pad(A, n^2) \in$ P, quindi $A \in$ P per la parte (a), e pertanto $A \in$ SPACE(n), ancora una volta per l'assunzione fatta. Ma questa è una contraddizione.

10



ARGOMENTI AVANZATI NELLA TEORIA DELLA COMPLESSITÀ

In questo capitolo introduciamo brevemente alcuni argomenti aggiuntivi in teoria della complessità. Questi argomenti sono parte di un attivo campo di ricerca, che presenta una letteratura molto vasta. Il capitolo è un estratto di alcuni degli sviluppi più avanzati, ma non fornisce un quadro completo. In particolare, due argomenti importanti che sono al di là degli scopi di questo testo sono la computazione quantistica e le prove verificabili probabilisticamente. *The Handbook of Theoretical Computer Science* [77] presenta una panoramica dei lavori precedenti in teoria della complessità.

Il capitolo contiene sezioni sugli algoritmi di approssimazione, sugli algoritmi probabilistici, sui sistemi di prova interattivi, sul calcolo parallelo e sulla crittografia. Le sezioni sono indipendenti ad eccezione del fatto che gli algoritmi probabilistici vengono usati nelle sezioni sui sistemi di prova interattivi e sulla crittografia.

10.1
ALGORITMI DI APPROSSIMAZIONE

In certi problemi, chiamati ***problemi di ottimizzazione***, cerchiamo la soluzione migliore tra una collezione di soluzioni possibili. Per esempio, potremmo voler trovare una clique massimale in un grafo, un vertex cover minimale, o uno dei cammini più brevi che collegano due nodi. Quando

un problema di ottimizzazione è NP-hard, come accade con i primi due di questi tipi di problemi, non esistono algoritmi di tempo polinomiale che trovino la soluzione migliore a meno che P = NP.

In pratica potremmo non aver bisogno della soluzione migliore o ***ottimale*** di un problema. Una soluzione che risultasse quasi ottimale potrebbe essere sufficientemente buona e molto più semplice da trovare. Come il nome sottintende, un ***algoritmo di approssimazione*** è progettato per trovare tali soluzioni approssimativamente ottimali.

Per esempio, si prenda il problema del vertex cover che abbiamo introdotto in Sezione 7.5. In quella sezione abbiamo presentato il problema come il linguaggio *VERTEX-COVER*, che rappresenta un ***problema di decisione*** - uno che ammette risposta si/no. Nella versione di ottimizzazione di questo problema, chiamata *MIN-VERTEX-COVER*, miriamo a produrre una delle coperture mediante vertici più piccole tra tutte le possibili coperture mediante vertici nel grafo di input. L'algoritmo di tempo polinomiale che segue risolve in modo approssimato il problema di ottimizzazione. Produce un vertex cover che ha al più taglia doppia rispetto a quella di un vertex cover minimale.

A = "Su input $\langle G \rangle$, dove G è un grafo non orientato:

1. Ripeti i passi che seguono fino a quando tutti gli archi in G toccano un arco marcato:
2. Trova un arco in G non toccato da nessun arco marcato.
3. Marca l'arco.
4. Dai in output tutti i nodi che sono estremi di archi marcati."

TEOREMA 10.1

A è un algoritmo di tempo polinomiale che produce un vertex cover di G al più doppio rispetto al più piccolo vertex cover.

DIMOSTRAZIONE. L'algoritmo A ovviamente computa in tempo polinomiale. Sia X l'insieme di nodi che dà in output. Sia H l'insieme degli archi che l'algoritmo marca. Sappiamo che X è un vertex cover perché H contiene o tocca ogni arco in G, e quindi X tocca tutti gli archi in G.

Per dimostrare che X ha al più dimensione doppia rispetto a un vertex cover minimale Y, proviamo due cose: che X ha dimensione doppia rispetto ad H, e che H non è più grande di Y. Prima di tutto, ogni arco in H contribuisce con due nodi alla composizione di X, pertanto X ha dimensione doppia rispetto ad H. In secondo luogo, Y è un vertex cover, pertanto ogni arco in H è toccato da qualche nodo in Y. Nessuno di questi nodi tocca due

archi in H perché gli archi in H non si toccano. Pertanto, il vertex cover Y è almeno tanto grande quanto H perché, per ogni arco in H, Y contiene un nodo diverso che lo tocca. Quindi X ha dimensione al più doppia rispetto alla taglia di Y.

MIN-VERTEX-COVER è un esempio di ***problema di minimizzazione*** perché puntiamo a trovare una soluzione *minimale* nella collezione di soluzioni possibili. In un ***problema di massimizzazione*** cerchiamo una soluzione *massimale*. Un algoritmo di approssimazione per un problema di minimizzazione è ***k-ottimale*** se trova sempre una soluzione che è al più k volte la soluzione ottimale. L'algoritmo precedente è 2-ottimale per il problema del vertex cover. Per un problema di massimizzazione, un algoritmo di approssimazione k-ottimale trova sempre una soluzione che è almeno $\frac{1}{k}$ volte la taglia di una soluzione ottimale.

Il seguente è un algoritmo di approssimazione per un problema di massimizzazione chiamato *MAX-CUT*. Un ***taglio (cut)*** in un grafo non orientato è una separazione dei vertici di V in due sottoinsiemi disgiunti S e T. Un ***arco del taglio*** è un arco tra un nodo in S ed un nodo in T. Un ***arco non del taglio*** è un arco che non è un arco del taglio. La cardinalità di un taglio è il numero di archi del taglio. Il problema *MAX-CUT* chiede di trovare un taglio massimale nel grafo G. Come abbiamo mostrato nel Problema 7.54, il problema è NP-completo. L'algoritmo seguente approssima *MAX-CUT* nei limiti di un fattore 2.

B = "Su input $\langle G \rangle$, dove G è un grafo non orientato con nodi V:

1. Sia $S = \emptyset$ e $T = V$.
2. Se spostando un singolo nodo, da S a T o da T a S, cresce la cardinalità del taglio, effettua la mossa e ripeti il passo.
3. Se non esiste un nodo di questo tipo, dai in output il taglio corrente e fermati."

L'algoritmo comincia l'esecuzione (presumibilmente) con un taglio cattivo e genera miglioramenti locali fino a quando nessun ulteriore miglioramento risulta possibile. Anche se questa procedura non produrrà in generale un taglio ottimale, facciamo vedere che ne produce uno con cardinalità almeno uguale alla metà della cardinalità di uno ottimale.

TEOREMA 10.2

B è un algoritmo di approssimazione 2-ottimale di tempo polinomiale per *MAX-CUT*.

DIMOSTRAZIONE. B computa in tempo polinomiale perché ogni esecuzione della fase 2 incrementa la cardinalità del taglio fino ad un massimo pari al numero totale di archi in G.

Mostriamo ora che il taglio prodotto da B è almeno la metà di uno ottimale. In realtà facciamo vedere qualcosa di più forte: gli archi del taglio di B sono almeno la metà di tutti gli archi in G. Si osservi che per ogni nodo di G, il numero di archi del taglio è almeno tanto grande quanto il numero di archi non del taglio, altrimenti B avrebbe spostato il nodo dall'altra parte. Sommiamo i numeri degli archi del taglio incidenti su ogni nodo. Questa somma è due volte il numero totale degli archi del taglio perché ognuno viene contato due volte, una per ciascuno dei suoi due punti terminali. Dalla precedente osservazione, questa somma deve essere almeno la somma corrispondente del numero degli archi non del taglio incidenti su ogni nodo. Quindi, G ha almeno tanti archi del taglio quanti archi non del taglio. Pertanto, il taglio contiene almeno la metà di tutti gli archi.

10.2
ALGORITMI PROBABILISTICI

Un ***algoritmo probabilistico*** è un algoritmo progettato per utilizzare il risultato di un processo casuale. Tipicamente, un tale algoritmo dovrebbe contenere un'istruzione per "lanciare una moneta", ed il risultato del lancio della moneta dovrebbe influenzare l'esecuzione seguente e l'output dell'algoritmo. Certi tipi di problemi sembrano essere risolvibili più facilmente tramite algoritmi probabilistici che attraverso algoritmi deterministici.

Come può essere meglio prendere una decisione eseguendo il lancio di una moneta rispetto a calcolare realmente, o anche a stimare, la scelta migliore in una particolare situazione? Qualche volta calcolare la scelta migliore può richiedere tempo eccessivo, e la stima può introdurre alterazioni che ne invalidano il risultato. Per esempio, gli esperti di statistica usano l'estrazione causale per determinare informazione circa gli individui in una popolazione numerosa, come sui loro gusti o sulle rispettive preferenze politiche. Interrogare tutti gli individui richiederebbe troppo tempo, e interrogare un sottoinsieme non scelto in modo casuale potrebbe fornire risultati errati.

La classe BPP

Cominceremo la nostra trattazione formale della computazione probabilistica definendo un modello di macchina di Turing probabilistica. Successivamente forniremo una classe di complessità associata alla computazione probabilistica efficiente ed alcuni esempi.

DEFINIZIONE **10.3**

Una ***macchina di Turing probabilistica*** M è un tipo di macchina di Turing non deterministica in cui ogni passo non deterministico è chiamato ***passo con lancio della moneta*** ed ha due successive mosse legittime. Assegniamo una probabilità ad ogni diramazione b della computazione di M su input w come segue. Definiamo la probabilità della diramazione b come

$$\Pr[b] = 2^{-k},$$

dove k è il numero di passi con lancio della moneta che sono presenti sulla diramazione b. Definiamo la probabilità che M accetti w come

$$\Pr[M \text{ accetta } w] = \sum_{\substack{b \text{ è una} \\ \text{diramazione accettante}}} \Pr[b].$$

In altre parole, la probabilità che M accetti w è la probabilità di raggiungere una configurazione accettante se simulassimo M su w lanciando una moneta per stabilire quale mossa eseguire in corrispondenza di ogni passo con lancio della moneta. Poniamo

$$\Pr[M \text{ rifiuta } w] = 1 - \Pr[M \text{ accetta } w].$$

Quando una macchina di Turing probabilistica decide un linguaggio, essa deve accettare tutte le stringhe nel linguaggio e deve rifiutare tutte le stringhe al di fuori del linguaggio come al solito, salvo che ora concediamo alla macchina una piccola probabilità di errore. Per $0 \leq \epsilon < \frac{1}{2}$, diciamo che ***$M$ decide il linguaggio A con probabilità di errore ϵ*** se

1. $w \in A$ implica $\Pr[M \text{ accetta } w] \geq 1 - \epsilon$, e
2. $w \notin A$ implica $\Pr[M \text{ rifiuta } w] \geq 1 - \epsilon$.

In altri termini, la probabilità di ottenere la risposta sbagliata simulando M è al più ϵ. Consideriamo anche limitazioni alla probabilità di errore che dipendono dalla lunghezza dell'input n. Per esempio, la probabilità di errore $\epsilon = 2^{-n}$ indica una probabilità di errore esponenzialmente piccola.

Siamo interessati ad algoritmi probabilistici che computano efficientemente in tempo e/o in spazio. Misuriamo la complessità di tempo e di

spazio di una macchina di Turing probabilistica come facciamo per le macchine di Turing non deterministiche: usando la diramazione peggiore della computazione su ciascun input.

DEFINIZIONE **10.4**

BPP è la classe di linguaggi che sono decisi da macchine di Turing probabilistiche di tempo polinomiale con una probabilità di errore pari a $\frac{1}{3}$.

Abbiamo definito questa classe con una probabilità di errore di $\frac{1}{3}$, ma qualsiasi probabilità di errore costante produrrebbe una definizione equivalente purché sia strettamente compresa tra 0 e $\frac{1}{2}$ in virtù del seguente ***lemma di amplificazione***. Esso fornisce un modo semplice per rendere la probabilità di errore esponenzialmente piccola. Si noti che è molto più verosimile che un algoritmo probabilistico con una probabilità di errore pari a 2^{-100} dia un risultato errato perché sul computer su cui è in esecuzione si verifica un problema hardware che non a causa di un lancio sfortunato della moneta.

LEMMA 10.5

Sia ϵ una costante fissata strettamente compresa tra 0 e $\frac{1}{2}$. Allora per ogni polinomio $p(n)$, ad ogni macchina di Turing probabilistica di tempo polinomiale M_1 che opera con probabilità di errore ϵ corrisponde una equivalente macchina di Turing probabilistica di tempo polinomiale M_2, che opera con una probabilità di errore pari a $2^{-p(n)}$.

IDEA. M_2 simula M_1 mandandola in esecuzione un numero polinomiale di volte e prendendo il voto di maggioranza dei risultati ottenuti. La probabilità di errore decresce esponenzialmente con il numero di esecuzioni di M_1 effettuate.

Si consideri il caso in cui $\epsilon = \frac{1}{3}$. Esso corrisponde ad una scatola che contiene molte palline rosse e blu. Sappiamo che $\frac{2}{3}$ delle palline sono di un colore e che il $\frac{1}{3}$ rimanente sono dell'altro colore, ma non sappiamo quale colore risulta predominante. Possiamo cercare di individuare questo colore attraverso diverse estrazioni casuali - diciamo 100 - di palline, per stabilire quale colore viene fuori più frequentemente. Quasi sicuramente il colore predominante nella scatola sarà il colore più frequente nel campione.

Le palline corrispondono alle diramazioni della computazione di M_1: rosso per le computazioni accettanti, blu per quelle di rifiuto. M_2 determina il colore mandando in esecuzione M_1. Un calcolo mostra che M_2 sbaglia con una probabilità di errore esponenzialmente piccola se esegue M_1 un numero polinomiale di volte e dà in output il risultato che viene fuori più spesso.

DIMOSTRAZIONE. Data una TM M_1 che decide un linguaggio con una probabilità di errore pari a $\epsilon < \frac{1}{2}$ ed un polinomio $p(n)$, costruiamo una TM M_2 che decide lo stesso linguaggio con probabilità di errore $2^{-p(n)}$.

M_2 = "Su input x:

1. Calcola k (vedi l'analisi che segue).
2. Esegui $2k$ simulazioni indipendenti di M_1 su input x.
3. Se la maggior parte delle esecuzioni di M_1 sono di accettazione, allora *accetta*; altrimenti, *rifiuta*."

Diamo un limite[1] alla probabilità che M_2 fornisca la risposta sbagliata su un input x. Il passo 2 produce una sequenza di $2k$ risultati dalla simulazione di M_1, dove ciascun risultato è corretto o scorretto. Se la maggior parte di questi risultati sono corretti, M_2 fornisce la risposta corretta. Limitiamo la probabilità che almeno la metà di questi risultati siano scorretti.

Sia S una qualsiasi sequenza di risultati che M_2 può ottenere al passo 2. Sia P_S la probabilità che M_2 ottenga S. Supponiamo che S abbia c risultati corretti e w risultati scorretti, quindi $c + w = 2k$. Se $c \leq w$ ed M_2 ottiene S, allora M_2 dà un output scorretto. Chiamiamo una tale S una *sequenza cattiva*. Sia ϵ_x la probabilità che M_1 sbagli su x. Se S è una sequenza cattiva, allora $P_S \leq (\epsilon_x)^w(1-\epsilon_x)^c$, che è al più $\epsilon^w(1-\epsilon)^c$ perché $\epsilon_x \leq \epsilon < \frac{1}{2}$ e quindi $\epsilon_x(1-\epsilon_x) \leq \epsilon(1-\epsilon)$, e perché $c \leq w$. Inoltre, $\epsilon^w(1-\epsilon)^c$ è al più $\epsilon^k(1-\epsilon)^k$ perché $k \leq w$ ed $\epsilon < 1-\epsilon$.

Sommando P_S per tutte le sequenze cattive S si ottiene la probabilità che M_2 dia un output scorretto. Abbiamo al più 2^{2k} sequenze cattive perché 2^{2k} è il numero di tutte le sequenze. Pertanto

$$\begin{aligned}&\Pr[\,M_2 \text{ dia un output scorretto sull'input } x\,]\\ &= \sum_{S \text{ cattiva}} P_S \leq 2^{2k} \cdot \epsilon^k(1-\epsilon)^k = (4\epsilon(1-\epsilon))^k.\end{aligned}$$

Abbiamo assunto $\epsilon < \frac{1}{2}$, quindi $4\epsilon(1-\epsilon) < 1$. Pertanto, la probabilità precedente decresce esponenzialmente in k, così come la probabilità di errore di M_2. Per calcolare un valore specifico di k che ci permette di limitare la probabilità di errore di M_2 con 2^{-t} per qualsiasi $t \geq 1$, poniamo $\alpha = -\log_2(4\epsilon(1-\epsilon))$ e scegliamo $k \geq t/\alpha$. In questo modo otteniamo una probabilità di errore pari a $2^{-p(n)}$ in tempo polinomiale.

[1] L'analisi della probabilità di errore usa la ***limitazione di Chernoff***, un risultato standard in teoria della probabilità. Qui ci serviamo di un calcolo alternativo e autosufficiente che evita qualsiasi dipendenza da quel risultato.

Primalità

Un ***numero primo*** è un intero maggiore di 1 che non è divisibile da interi positivi diversi da se stesso e da 1. Un numero maggiore di 1 che non è primo è detto ***composto***. L'antico problema di verificare se un intero è primo o composto è stato oggetto di ricerche approfondite. Un algoritmo di tempo polinomiale per questo problema è ora noto [4], ma è troppo difficile per essere presentato in questo testo.

Descriviamo invece un algoritmo probabilistico di tempo polinomiale molto più semplice per la verifica della primalità.

Un modo per stabilire se un numero è primo è tentare tutti i possibili interi più piccoli del numero stesso e vedere se sono divisori, detti anche ***fattori***. Un algoritmo di questo tipo ha complessità di tempo esponenziale, perché il valore di un numero è esponenziale nella sua lunghezza. L'algoritmo probabilistico di verifica della primalità che descriviamo opera in modo completamente diverso. Non cerca i fattori. Infatti, non si conosce alcun algoritmo di tempo polinomiale per trovare i fattori.

Prima di presentare l'algoritmo, ricordiamo un po' di notazione dalla teoria dei numeri. Tutti i numeri in questa sezione sono interi. Per ogni p maggiore di 1, diciamo che due numeri sono ***equivalenti modulo p*** se differiscono per un multiplo di p. Se i numeri x ed y sono equivalenti modulo p, scriviamo $x \equiv y \pmod{p}$. Sia $x \bmod p$ il più piccolo y non negativo tale che $x \equiv y \pmod{p}$. Ogni numero è equivalente modulo p a qualche elemento dell'insieme $\mathcal{Z}_p = \{0, \ldots, p-1\}$. Per comodità, poniamo $\mathcal{Z}_p^+ = \{1, \ldots, p-1\}$. Possiamo far riferimento agli elementi di questi insiemi attraverso numeri equivalenti modulo p, come quando ci riferiamo a $p-1$ con -1.

L'idea principale che sottende all'algoritmo deriva dal risultato che segue, chiamato ***piccolo teorema di Fermat***.

TEOREMA 10.6

Se p è primo ed $a \in \mathcal{Z}_p^+$, allora $a^{p-1} \equiv 1 \pmod{p}$.

Per esempio, se $p = 7$ ed $a = 2$, il teorema dice che $2^{(7-1)} \bmod 7$ dovrebbe essere 1 perché 7 è primo. Semplici conti

$$2^{(7-1)} = 2^6 = 64 \quad \text{e} \quad 64 \bmod 7 = 1$$

confermano il risultato. Si supponga invece di tentare $p = 6$. Allora

$$2^{(6-1)} = 2^5 = 32 \quad \text{e} \quad 32 \bmod 6 = 2$$

danno un risultato diverso da 1, implicando in accordo al teorema che 6 non risulta primo. Naturalmente lo sapevamo già. Tuttavia, questo metodo

prova che 6 è composto senza trovare i suoi fattori. Il Problema 10.16 chiede di fornire una dimostrazione di questo teorema.

Si pensi al teorema precedente come ad uno strumento che fornisce un tipo di "test" per la primalità, detto un ***test di Fermat***. Quando diciamo che p supera il test di Fermat in a, intendiamo che $a^{p-1} \equiv 1 \pmod{p}$. Il teorema stabilisce che i primi superano tutti i test di Fermat per $a \in \mathcal{Z}_p^+$. Abbiamo osservato che 6 fallisce un test di Fermat, quindi 6 non è primo.

Possiamo usare i test di Fermat per dare un algoritmo che determina la primalità? Più o meno. Chiameremo un numero ***pseudoprimo*** se supera i test di Fermat per tutti gli a più piccoli relativamente primi ad esso. Con l'eccezione dei poco frequenti ***numeri di Carmichael***, che sono composti anche se superano tutti i test di Fermat, i numeri pseudoprimi sono identici ai numeri primi. Cominceremo descrivendo un algoritmo probabilistico di tempo polinomiale molto semplice che distingue primi da composti con l'eccezione dei numeri di Carmichael. Successivamente presenteremo ed analizzeremo l'algoritmo completo per la verifica della primalità.

Un algoritmo per la pseudoprimalità che effettuasse tutti i test di Fermat richiederebbe tempo esponenziale. L'idea chiave alla base dell'algoritmo probabilistico di tempo polinomiale è che se il numero non è pseudoprimo, allora fallisce almeno sulla metà di tutti i test. (Al momento si dia per vera questa affermazione. Il Problema 10.14 chiede di provarla.) L'algoritmo opera tentando diversi test scelti in modo casuale. Se uno di essi fallisce, il numero deve essere composto. L'algoritmo contiene un parametro k che determina la probabilità di errore.

PSEUDOPRIME = "Su input p:

1. Scegli $a_1, \ldots, a_k$ in modo casuale in $\mathcal{Z}_p^+$.
2. Calcola $a_i^{p-1} \bmod p$ per ciascun i.
3. Se tutti i valori calcolati sono 1, *accetta*; altrimenti, *rifiuta*."

Se p è pseudoprimo, allora supera tutti i test e l'algoritmo accetta con certezza. Se p non è pseudoprimo, allora supera al più la metà di tutti i test. In questo caso, supera ciascun test scelto in modo casuale con probabilità al più $\frac{1}{2}$. La probabilità che superi tutti i k test scelti in modo casuale è pertanto al più 2^{-k}. L'algoritmo opera in tempo polinomiale perché l'esponenziazione modulare è calcolabile in tempo polinomiale (si veda il Problema 7.40).

Per convertire l'algoritmo precedente in un algoritmo per la primalità, introduciamo un test più sofisticato che evita il problema rappresentato dai numeri di Carmichael. Il principio di fondo è che il numero 1 ha esattamente due radici quadrate, 1 e -1, modulo qualsiasi primo p. Per molti numeri composti, inclusi tutti i numeri di Carmichael, 1 ha quattro o più radici quadrate. Per esempio, ± 1 e ± 8 sono le quattro radici quadrate di 1, modulo 21. Se un numero supera il test di Fermat in a, l'algoritmo trova una delle radici quadrate di 1 in modo casuale e controlla

se la radice quadrata è 1 o -1. Se non lo è, sappiamo che il numero non è primo.

Possiamo ottenere le radici quadrate di 1 se p supera il test di Fermat in a perché $a^{p-1} \bmod p = 1$, e quindi $a^{(p-1)/2} \bmod p$ è una radice quadrata di 1. Se questo valore è ancora 1, possiamo ripetutamente dividere l'esponente per 2, fino a quando l'esponente risultante resta un intero, e vedere se il primo numero che è differente da 1 è -1 o qualche altro numero. Forniremo una dimostrazione formale della correttezza dell'algoritmo subito dopo la sua descrizione. Si scelga $k \geq 1$ come un parametro che fissa la massima probabilità di errore a 2^{-k}.

PRIME = "Su input p:

1. Se p è pari, *accetta* se $p = 2$; altrimenti, *rifiuta*.
2. Scegli $a_1, \ldots, a_k$ in modo casuale in $\mathcal{Z}_p^+$.
3. Per ciascun i da 1 a k:
4. Calcola $a_i^{p-1} \bmod p$ e *rifiuta* se diverso da 1.
5. Sia $p - 1 = s \cdot 2^l$ dove s è dispari.
6. Calcola la sequenza $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. Se qualche elemento di questa sequenza non è 1, trova l'ultimo elemento che non è 1 e *rifiuta* se l'elemento non è -1.
8. Tutti i test sono stati superati a questo punto, pertanto *accetta*."

I due lemmi seguenti mostrano che l'algoritmo *PRIME* funziona correttamente. Ovviamente l'algoritmo è corretto quando p è pari, quindi consideriamo solo il caso in cui p sia dispari. Diciamo che a_i è un ***testimone (della compostezza)*** se l'algoritmo rifiuta al passo 4 o al passo 7, usando a_i.

LEMMA 10.7

Se p è un numero primo dispari, $\Pr[\, PRIME \text{ accetta } p \,] = 1$.

DIMOSTRAZIONE. Prima mostriamo che se p è primo, non esiste nessun testimone e quindi nessuna diramazione dell'algoritmo rifiuta. Se a fosse un testimone al passo 4, allora $(a^{p-1} \bmod p) \neq 1$ e il piccolo teorema di Fermat implicherebbe che p è composto. Se a fosse un testimone al passo 7, esisterebbe un qualche b in $\mathcal{Z}_p^+$, per cui $b \not\equiv \pm 1 \pmod p$ e $b^2 \equiv 1 \pmod p$. Pertanto, $b^2 - 1 \equiv 0 \pmod p$. La fattorizzazione di $b^2 - 1$ dà

$$(b-1)(b+1) \equiv 0 \pmod p,$$

che implica

$$(b-1)(b+1) = cp$$

per qualche intero positivo c. Poiché $b \not\equiv \pm 1 \pmod p$, entrambi $b - 1$ e $b + 1$ sono strettamente compresi tra 0 e p. Pertanto, p è composto perché un multiplo di un numero primo non può essere espresso come prodotto di numeri più piccoli di esso.

Il lemma seguente mostra che l'algoritmo identifica i numeri composti con alta probabilità. Prima di tutto presentiamo un risultato elementare importante della teoria dei numeri. Due numeri sono relativamente primi se non posseggono divisori comuni diversi da 1. Il ***teorema cinese del resto*** dice che esiste una corrispondenza uno-a-uno tra $\mathcal{Z}_{pq}$ e $\mathcal{Z}_p \times \mathcal{Z}_q$ se p e q sono relativamente primi. Ciascun numero $r \in \mathcal{Z}_{pq}$ corrisponde alla coppia (a, b), dove $a \in \mathcal{Z}_p$ e $b \in \mathcal{Z}_q$, in modo tale che

$$r \equiv a \pmod p \text{ e}$$
$$r \equiv b \pmod q.$$

LEMMA 10.8

Se p è un numero composto dispari, $\Pr[\, PRIME \text{ accetta } p \,] \leq 2^{-k}$.

DIMOSTRAZIONE. Mostriamo che se p è un numero composto dispari ed a viene scelto in modo casuale in $\mathcal{Z}_p^+$,

$$\Pr[\, a \text{ è un testimone} \,] \geq \tfrac{1}{2}.$$

Lo facciamo facendo vedere che in $\mathcal{Z}_p^+$esistono almeno tanti testimoni quanti non testimoni, trovando un unico testimone per ogni non testimone.

Per tutti i non testimoni, la sequenza calcolata al passo 6 è costituita da tutti 1 oppure contiene -1 in qualche posizione, seguito da alcuni 1. Per esempio, 1 stesso è un non testimone del primo tipo, mentre -1 è un non testimone del secondo tipo poiché s è dispari e $(-1)^{s \cdot 2^0} \equiv -1$ e $(-1)^{s \cdot 2^1} \equiv 1$. Tra tutti i non testimoni del secondo tipo, si prenda un non testimone per il quale -1 compare nella posizione più grande nella sequenza. Sia h questo non testimone e sia j la posizione di -1 nella sua sequenza, dove le posizioni della sequenza sono numerate a partire da 0. Quindi $h^{s \cdot 2^j} \equiv -1 \pmod p$.

Poiché p è composto, o p è una potenza di un primo oppure possiamo scrivere p come il prodotto di q ed r - due numeri che sono relativamente primi. Consideriamo prima il secondo caso. Il teorema cinese del resto implica che esiste in $\mathcal{Z}_p$ un numero t per cui

$$t \equiv h \pmod q \quad \text{e}$$
$$t \equiv 1 \pmod r.$$

Pertanto,

$$t^{s\cdot 2^j} \equiv -1 \pmod{q} \quad \text{e}$$
$$t^{s\cdot 2^j} \equiv 1 \pmod{r}.$$

Quindi t è un testimone perché $t^{s\cdot 2^j} \not\equiv \pm 1 \pmod{p}$ ma $t^{s\cdot 2^{j+1}} \equiv 1 \pmod{p}$.

Ora che abbiamo un testimone, possiamo ottenerne molti altri. Dimostriamo che $dt \bmod p$ è un testimone unico per ciascun non testimone d attraverso due osservazioni. La prima è che $d^{s\cdot 2^j} \equiv \pm 1 \pmod{p}$ e $d^{s\cdot 2^{j+1}} \equiv 1 \pmod{p}$ per il modo in cui è stato scelto j. Pertanto, $dt \bmod p$ è un testimone perché $(dt)^{s\cdot 2^j} \not\equiv \pm 1$ e $(dt)^{s\cdot 2^{j+1}} \equiv 1 \pmod{p}$.

La seconda è che se d_1 e d_2 sono non testimoni distinti, $d_1 t \bmod p \neq d_2 t \bmod p$. La motivazione è che $t^{s\cdot 2^{j+1}} \bmod p = 1$. Quindi, $t \cdot t^{s\cdot 2^{j+1}-1} \bmod p = 1$. Pertanto, se fosse $td_1 \bmod p = td_2 \bmod p$, risulterebbe

$$d_1 = t \cdot t^{s\cdot 2^{j+1}-1} d_1 \bmod p = t \cdot t^{s\cdot 2^{j+1}-1} d_2 \bmod p = d_2.$$

Perciò, il numero di testimoni deve essere tanto grande quanto il numero di non testimoni, ed abbiamo completato l'analisi nel caso in cui p non è una potenza di un primo.

Nel caso in cui è una potenza di un primo, risulta $p = q^e$ dove q è primo ed $e > 1$. Sia $t = 1 + q^{e-1}$. Espandendo t^p attraverso il teorema del binomio, otteniamo

$$t^p = (1 + q^{e-1})^p = 1 + p \cdot q^{e-1} + \text{multipli di potenze maggiori di } q^{e-1},$$

il che è equivalente ad 1 mod p. Pertanto t è un testimone al passo 4 perché se $t^{p-1} \equiv 1 \pmod{p}$, allora $t^p \equiv t \not\equiv 1 \pmod{p}$. Come nel caso precedente, usiamo questo testimone per ottenerne molti altri. Se d è un non testimone, risulta $d^{p-1} \equiv 1 \pmod{p}$, ma allora $dt \bmod p$ è un testimone. Per di più, se d_1 e d_2 sono non testimoni distinti, allora $d_1 t \bmod p \neq d_2 t \bmod p$. Altrimenti,

$$d_1 = d_1 \cdot t \cdot t^{p-1} \bmod p = d_2 \cdot t \cdot t^{p-1} \bmod p = d_2.$$

Pertanto, il numero di testimoni deve essere tanto grande quanto il numero di non testimoni e la dimostrazione è completa.

L'algoritmo precedente e la sua analisi provano il teorema che segue. Sia $PRIMES = \{n \mid n \text{ un numero primo in binario}\}$.

TEOREMA 10.9

$PRIMES \in$ BPP.

Si noti che l'algoritmo probabilistico per la primalità ha ***errore unilaterale***. Quando l'algoritmo dà in output *rifiuta*, sappiamo che l'input deve essere un composto. Quando l'output è *accetta*, sappiamo che l'input potrebbe essere un primo o un composto. Perciò, può esserci una risposta scorretta solo quando l'input è un numero composto. La caratteristica dell'errore unilaterale è comune a molti algoritmi probabilistici, tanto che la classe di complessità RP è appositamente introdotta per essa.

DEFINIZIONE **10.10**

RP è la classe dei linguaggi che sono decisi da macchine di Turing probabilistiche di tempo polinomiale dove gli input nel linguaggio sono accettati con probabilità almeno pari a $\frac{1}{2}$, e gli input che non appartengono al linguaggio sono rifiutati con probabilità 1.

Possiamo rendere la probabilità di errore esponenzialmente piccola e mantenere il tempo di esecuzione polinomiale usando una tecnica di amplificazione della probabilità simile a quella (in realtà più semplice) utilizzata nel Lemma 10.5. Il nostro algoritmo precedente prova che $COMPOSITES \in$ RP.

Programmi ramificati a lettura singola

Un *programma ramificato* è un modello di computazione usato in teoria della complessità ed in certi settori applicativi quali la progettazione assistita dal computer. Questo modello rappresenta un processo decisionale che chiede i valori delle variabili di input e stabilisce come procedere in base alle risposte ottenute alle interrogazioni effettuate. Raffiguriamo il processo decisionale attraverso un grafo i cui nodi corrispondono alla specifica variabile interrogata in quel momento nel processo.

In questa sezione indaghiamo sulla complessità di verificare se due programmi ramificati sono equivalenti. In generale il problema è coNP-completo. Se poniamo una certa restrizione naturale alla classe dei programmi ramificati, possiamo fornire un algoritmo probabilistico di tempo polinomiale per verificare l'equivalenza. L'algoritmo è particolarmente interessante per due ragioni. La prima è che non si conosce per questo problema nessun algoritmo di tempo polinomiale, e quindi esso fornisce un esempio di come la probabilità apparentemente espanda la classe dei linguaggi in cui l'appartenenza può essere verifica efficientemente. La seconda è che l'algoritmo introduce la tecnica dell'assegnamento di valori non booleani a variabili normalmente booleane, al fine di analizzare il comportamento di una qualche funzione booleana di quelle variabili. La tecnica è usata con profitto nei sistemi di prova interattivi, come mostreremo nella Sezione 10.4.

DEFINIZIONE **10.11**

Un ***programma ramificato*** è un grafo diretto aciclico[2] dove tutti i nodi sono etichettati con variabili, ad eccezione di due ***nodi di output*** etichettati con 0 o 1. I nodi che sono etichettati con variabili sono chiamati ***nodi di domanda***. Ogni nodo di domanda ha due archi uscenti: uno etichettato con 0 e l'altro etichettatto con 1. Entrambi i nodi di output non hanno archi uscenti. Uno dei nodi in un programma ramificato è il nodo iniziale.

Un programma ramificato determina una funzione booleana come segue. Si prenda un qualsiasi assegnamento alle variabili che si presentano sui suoi nodi di domanda e, cominciando dal nodo iniziale, si segua il cammino individuato prendendo l'arco uscente da ciascun nodo di domanda in accordo al valore assegnato alla variabile indicata, fino a quando non viene raggiunto uno dei nodi di output. L'output è l'etichetta del nodo di output. La Figura 10.12 fornisce due esempi di programmi ramificati.

I programmi ramificati sono in relazione con la classe L attraverso una modalità che è analoga alla relazione tra i circuiti booleani e la classe P. Il Problema 10.13 chiede di mostrare che un programma ramificato con un numero di nodi polinomiale può verificare l'appartenenza per qualsiasi linguaggio su $\{0,1\}$ che appartenga a L.

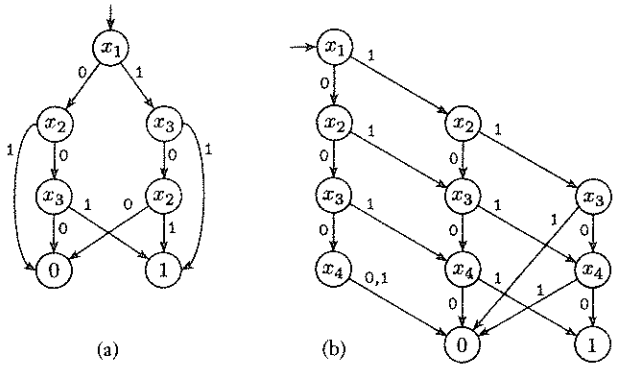


FIGURA 10.12
Due programmi ramificati a lettura singola

[2]Un grafo diretto è ***aciclico*** se non contiene cicli diretti.

Due programmi ramificati sono equivalenti se determinano funzioni uguali. Il Problema 10.9 chiede di mostrare che il problema di verificare l'equivalenza di programmi ramificati è coNP-completo. Nel seguito consideriamo una forma ristretta di programmi ramificati. Un ***programma ramificato a lettura singola*** può interrogare ciascuna variabile al più una volta su ogni cammino diretto dal nodo iniziale al nodo di output. Entrambi i programmi ramificati in Figura 10.12 hanno la caratteristica della lettura singola. Sia

$EQ_{\mathsf{ROBP}} = \{\langle B_1, B_2\rangle \mid B_1 \text{ e } B_2 \text{ sono programmi ramificati a lettura singola equivalenti}\}$.

TEOREMA 10.13

EQ_{ROBP} appartiene a BPP.

IDEA. Per iniziare proviamo ad assegnare valori in modo casuale alle variabili da x_1 fino a x_m, che sono presenti in B_1 ed in B_2, e valutiamo i programmi ramificati sull'assegnamento prodotto. Accettiamo se B_1 e B_2 sono concordi sull'assegnamento e rifiutiamo altrimenti. Purtroppo questa strategia non funziona perché due programmi ramificati a lettura singola che non sono equivalenti potrebbero non essere concordi soltanto su un singolo assegnamento tra i 2^m possibili assegnamenti booleani alle variabili. La probabilità di selezionare questo assegnamento è esponenzialmente piccola. Pertanto, accetteremmo con alta probabilità anche quando B_1 e B_2 non sono equivalenti, e ciò è insoddisfacente.

Invece, modifichiamo la strategia scegliendo in modo casuale un assegnamento non booleano per le variabili, e valutiamo B_1 e B_2 in un modo opportunamente definito. Possiamo mostrare che se B_1 e B_2 non sono equivalenti, le valutazioni casuali verosimilmente saranno differenti.

DIMOSTRAZIONE. Assegniamo polinomi definiti su $x_1, \ldots, x_m$ ai nodi ed agli archi di un programma ramificato a lettura singola B come segue. La funzione costante 1 viene assegnata al nodo iniziale. Se ad un nodo etichettato con x è stato assegnato il polinomio p, assegniamo il polinomio xp al suo arco uscente etichettato con 1, ed assegniamo il polinomio $(1-x)p$ al suo arco uscente etichettato con 0. Se agli archi entranti di un nodo sono stati assegnati polinomi, assegniamo al nodo la somma di quei polinomi. Infine, il polinomio che è stato assegnato al nodo di output etichettato con 1 viene anche assegnato al programma ramificato stesso. A questo punto siamo pronti per presentare l'algoritmo probabilistico di tempo polinomiale per EQ_{ROBP}. Sia $\mathcal{F}$ un campo finito con almeno $3m$ elementi.

D = "Su input $\langle B_1, B_2 \rangle$, due programmi ramificati a lettura singola:

1. Scegli elementi da a_1 fino ad a_m in modo casuale in $\mathcal{F}$.
2. Valuta i polinomi assegnati p_1 e p_2 in a_1 fino ad a_m.
3. Se $p_1(a_1, \ldots, a_m) = p_2(a_1, \ldots, a_m)$, *accetta*; altrimenti, *rifiuta*."

L'algoritmo computa in tempo polinomiale perché possiamo valutare il polinomio corrispondente ad un programma ramificato senza in realtà costruire il polinomio. Mostreremo che l'algoritmo decide EQ_{ROBP} con una probabilità di errore pari al più ad $\frac{1}{3}$.

Esaminiamo la relazione tra un programma ramificato a lettura singola B ed il polinomio p ad esso assegnato. Si osservi che per qualsiasi assegnamento booleano alle variabili di B, tutti i polinomi assegnati ai suoi nodi assumono valore 0 o 1. I polinomi che assumono valore 1 sono quelli sul cammino computazionale per quell'assegnamento. Quindi B e p concordano quando le variabili assumono valori booleani. Analogamente, poiché B è a lettura singola, possiamo scrivere p come una somma di termini prodotto $y_1 y_2 \cdots y_m$, dove ciascun y_i è x_i, $(1 - x_i)$, o 1, e dove ciascun termine prodotto corrisponde ad un cammino in B dal nodo iniziale al nodo di output etichettato con 1. Il caso di $y_i = 1$ si presenta quando un cammino non contiene la variabile x_i.

Si prenda ciascun termine prodotto di p contenente un y_i che vale 1 e si divida nella somma di due termini prodotto, uno dove $y_i = x_i$ ed un altro dove $y_i = (1 - x_i)$. Facendo ciò si ottiene un polinomio equivalente perché $1 = x_i + (1 - x_i)$. Si continui a dividere i termini prodotto fino a quando ciascun y_i è x_i o $(1 - x_i)$. Il risultato finale è un polinomio equivalente q che contiene un termine prodotto per ciascun assegnamento su cui B vale 1. Ora siamo pronti per analizzare il comportamento dell'algoritmo D.

Prima di tutto mostriamo che se B_1 e B_2 sono equivalenti, D accetta sempre. Se i programmi ramificati sono equivalenti, essi assumono valore 1 esattamente sugli stessi assegnamenti. Di conseguenza, i polinomi q_1 e q_2 sono uguali perché contengono termini prodotto identici. Pertanto, p_1 e p_2 sono uguali su tutti gli assegnamenti.

In secondo luogo, mostriamo che se B_1 e B_2 non sono equivalenti, D rifiuta con una probabilità pari almeno a $\frac{2}{3}$. Questa conclusione segue immediatamente dal Lemma 10.15.

La dimostrazione precedente si basa sui lemmi seguenti che riguardano la probabilità di trovare in modo casuale una radice di un polinomio in funzione del numero di variabili che ha, dei gradi delle sue variabili e della dimensione del campo sottostante.

LEMMA 10.14

Per ogni $d \geq 0$, un polinomio p di grado d su una singola variabile x ha al più d radici, oppure è ovunque uguale a 0.

DIMOSTRAZIONE. Procediamo per induzione su d.

Base: Prova per $d = 0$. Un polinomio di grado 0 è costante. Se la costante non è 0, il polinomio chiaramente non ha radici.

Passo induttivo: Si assuma vera per $d - 1$ e si provi vera per d. Se p è un polinomio non nullo di grado d con una radice in a, il polinomio $x - a$ divide p in due parti. Quindi $p/(x - a)$ è un polinomio non nullo di grado $d - 1$, ed ha al più $d - 1$ radici in virtù dell'ipotesi induttiva.

LEMMA 10.15

Sia $\mathcal{F}$ un campo finito con f elementi e sia p un polinomio non nullo sulle variabili da x_1 a x_m, dove ciascuna variabile ha grado al più d. Se a_1 fino ad a_m sono scelti in modo casuale in $\mathcal{F}$, allora $\Pr[p(a_1, \ldots, a_m) = 0] \leq md/f$.

DIMOSTRAZIONE. Procediamo per induzione su m.

Base: Prova per $m = 1$. Dal Lemma 10.14, p ha al più d radici, quindi la probabilità che a_1 sia una di esse è al più d/f.

Passo induttivo: Si assuma vera per $m - 1$ e si provi vera per m. Sia x_1 una delle variabili di p. Per ciascun $i \leq d$, sia p_i il polinomio comprendente i termini di p che contengono x_1^i, ma dove x_1^i è stato fattorizzato. Allora

$$p = p_0 + x_1 p_1 + x_1^2 p_2 + \cdots + x_1^d p_d.$$

Se $p(a_1, \ldots, a_m) = 0$, si presenta uno di due casi. O tutti i p_i assumono valore 0, oppure qualche p_i non assume valore 0 ed a_1 è una radice del polinomio ad una sola variabile ottenuto valutando p_0 fino a p_d su a_2 fino ad a_m.

Per limitare la probabilità che si verifichi il primo caso, si osservi che uno dei p_j deve essere non nullo perché p è non nullo. Quindi la probabilità che tutti i p_i assumano valore 0 è al più la probabilità che p_j assuma valore 0. Per l'ipotesi induttiva questa è al più $(m-1)d/f$, perché p_j ha al più $m-1$ variabili.

Per limitare la probabilità che si verifichi il secondo caso, si osservi che se qualche p_i non assume valore 0, allora sull'assegnamento a_2 fino ad a_m, p si riduce ad un polinomio non nullo nella singola variabile x_1. La base ha già dimostrato che a_1 è una radice di un tale polinomio con probabilità al più d/f.

Pertanto, la probabilità che a_1 fino ad a_m sia una radice del polinomio è al più $(m - 1)d/f + d/f = md/f$.

Concludiamo questa sezione con un punto importante che riguarda l'uso della casualità negli algoritmi probabilistici. Nelle nostre analisi assumiamo che gli algoritmi siano implementati usando casualità vera. La casualità vera può essere difficile (o impossibile) da ottenere, e quindi è solitamente simulata attraverso ***generatori pseudocasuali***, che sono algoritmi deterministici il cui output sembra casuale. Anche se l'output di qualsiasi procedura deterministica non può mai essere veramente casuale, alcune di queste procedure generano risultati che esibiscono alcune caratteristiche dei risultati generati in modo casuale. Algoritmi che sono progettati per usare casualità possono funzionare bene ugualmente con questi generatori pseudocasuali, ma provare che ciò accade è generalmente molto difficile. Infatti a volte gli algoritmi probabilistici possono non funzionare bene con certi generatori pseudocasuali. Sono stati ideati generatori pseudocasuali sofisticati che producono risultati indistinguibili da risultati veramente casuali, mediante qualsiasi test che operi in tempo polinomiale, sotto l'assunzione che esista una funzione one way. (Si veda la Sezione 10.6 per una discussione delle funzioni one way.)

10.3 ALTERNANZA

L'alternanza è una generalizzazione del non determinismo che si è rivelata utile nella comprensione delle relazioni tra le classi di complessità e nella classificazione di problemi specifici in accordo alle rispettive complessità. Usando l'alternanza possiamo semplificare svariate dimostrazioni in teoria della complessità, ed esibire un legame sorprendente tra le misure di complessità di tempo e di spazio.

Un algoritmo alternante può contenere istruzioni per diramare un processo in processi figli multipli, esattamente come in un algoritmo non deterministico. La differenza tra i due sta nel modo in cui l'accettazione viene determinata. Una computazione non deterministica accetta se uno qualsiasi dei processi avviati accetta. Quando una computazione alternante si divide in processi multipli, si presentano due possibilità. L'algoritmo può stabilire che il processo corrente accetta se *uno qualsiasi* dei suoi figli accetta, o può stabilire che il processo corrente accetta se *tutti* i suoi figli accettano.

Raffiguriamo la differenza tra computazione alternante e non deterministica con alberi che rappresentano la struttura delle diramazioni dei processi generati. Ciascun nodo rappresenta una configurazione in un processo. In una computazione non deterministica, ogni nodo calcola l'operazione OR dei suoi figli. Ciò corrisponde alla modalità di accettazione non deterministica usuale in cui un processo è accettante se uno qualsiasi dei suoi figli è accettante. In una computazione alternante i nodi possono calcolare le operazioni AND o OR, come stabilito dall'algoritmo. Ciò corrisponde alla modalità di accettazione alternante, in cui un processo è accettante se tutti o uno qualsiasi dei suoi figli accetta. Definiamo una macchina di Turing alternante come segue.

DEFINIZIONE **10.16**

Una ***macchina di Turing alternante*** è una macchina di Turing non deterministica con una caratteristica aggiuntiva. I suoi stati, ad eccezione di q_{accept} e di q_{reject}, sono divisi in ***stati universali*** e ***stati esistenziali***. Quando mandiamo in esecuzione una macchina di Turing alternante su una stringa di input, etichettiamo ciascun nodo del suo albero di computazione non deterministico con $\wedge$ o $\vee$, a seconda del fatto che la configurazione corrispondente contenga uno stato universale o uno stato esistenziale. Designiamo un nodo come accettante se è etichettato con $\wedge$ e tutti i suoi figli sono accettanti, o se è etichettato con $\vee$ ed uno qualsiasi dei suoi figli è accettante. L'input viene accettato se il nodo iniziale è designato come accettante

La figura che segue mostra alberi di computazione non deterministica ed alternante. Etichettiamo i nodi dell'albero di computazione alternante con $\wedge$ o $\vee$ ad indicare quale funzione dei propri figli essi calcolano.

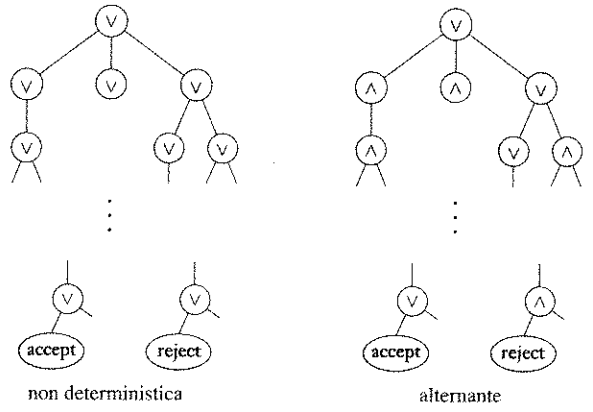


FIGURA 10.17
Alberi di computazione non deterministica ed alternante

Tempo e spazio alternanti

Definiamo la complessità di tempo e di spazio di queste macchine allo stesso modo in cui l'abbiamo fatto per le macchine di Turing non deterministiche: prendendo il tempo o lo spazio massimo usato da una diramazione della computazione. Definiamo le classi di complessità di tempo e di spazio alternanti come segue.

DEFINIZIONE **10.18**

$$\mathbf{ATIME}(t(n)) = \{L \mid L \text{ è deciso da una macchina di Turing alternante di tempo } O(t(n))\}.$$
$$\mathbf{ASPACE}(f(n)) = \{L \mid L \text{ è deciso da una macchina di Turing alternante di spazio } O(f(n))\}.$$

Definiamo , **APSPACE** e **AL** come le classi di linguaggi che sono decise da macchine di Turing alternanti di tempo polinomiale, alternanti di spazio polinomiale ed alternanti di spazio logaritmico, rispettivamente.

ESEMPIO 10.19

Una ***tautologia*** è una formula booleana che assume valore 1 su tutti gli assegnamenti alle sue variabili. Sia $TAUT = \{\langle\phi\rangle \mid \phi \text{ è una tautologia}\}$. L'algoritmo alternante che segue mostra che $TAUT$ è in ·

"Su input $\langle\phi\rangle$:

1. Scegli universalmente tutti gli assegnamenti per le variabili di ϕ.
2. Per un particolare assegnamento, valuta ϕ.
3. Se ϕ vale 1, *accetta*; altrimenti, *rifiuta*."

Il passo 1 di questo algoritmo seleziona non deterministicamente tutti gli assegnamenti per le variabili di ϕ con diramazione universale. Ciò richiede che tutte le diramazioni siano accettanti affinché l'intera computazione risulti accettante.

I passi 2 e 3 verificano deterministicamente se l'assegnamento che è stato selezionato su una particolare diramazione della computazione soddisfa la formula. Pertanto l'algoritmo accetta il proprio input se determina che tutti gli assegnamenti soddisfano la formula.

Si osservi che $TAUT$ è un elemento di coNP. In realtà si può facilmente far vedere che qualsiasi problema in coNP appartiene ad AP utilizzando un algoritmo simile al precedente.

ESEMPIO 10.20

Questo esempio caratterizza un linguaggio in AP che non si sa se appartiene a NP o a coNP. Riprendiamo il linguaggio *MIN-FORMULA* che abbiamo definito nel Problema 7.21 a pagina 348. L'algoritmo seguente mostra che *MIN-FORMULA* appartiene ad AP.

"Su input $\langle\phi\rangle$:

1. Scegli universalmente tutte le formule ψ che sono più corte di ϕ.
2. Scegli esistenzialmente un assegnamento per le variabili di ϕ.
3. Valuta sia ϕ che ψ su questo assegnamento.
4. *Accetta* se le formule assumono valori diversi. *Rifiuta* se esse assumono lo stesso valore."

Questo algoritmo inizia con la diramazione universale per selezionare tutte le formule più corte al passo 1 e poi passa alla diramazione esistenziale per selezionare l'assegnamento al passo 2. Il termine *alternanza* deriva dalla capacità di alternare, o muovere, tra la diramazione universale e quella esistenziale.

L'alternanza ci permette di stabilire una connessione notevole tra le misure di complessità di tempo e di spazio. Grosso modo il teorema seguente mostra un'equivalenza tra tempo alternante e spazio deterministico per limitazioni correlate polinomialmente, e un'altra equivalenza tra spazio alternante e tempo deterministico quando la limitazione di tempo è esponenzialmente più grande della limitazione di spazio.

TEOREMA 10.21

Per $f(n) \geq n$, risulta $\text{ATIME}(f(n)) \subseteq \text{SPACE}(f(n)) \subseteq \text{ATIME}(f^2(n))$.
Per $f(n) \geq \log n$, risulta $\text{ASPACE}(f(n)) = \text{TIME}(2^{O(f(n))})$.

Di conseguenza, AL = P, AP = PSPACE, e APSPACE = EXPTIME. La dimostrazione di questo teorema si trova nei quattro lemmi che seguono.

LEMMA 10.22

Per $f(n) \geq n$, risulta $\text{ATIME}(f(n)) \subseteq \text{SPACE}(f(n))$.

DIMOSTRAZIONE. Convertiamo una macchina alternante M di tempo $O(f(n))$ in una macchina deterministica S di spazio $O(f(n))$ che simula M come segue. Su input w, il simulatore S esegue una visita in pro-

fondità dell'albero di computazione di M per stabilire quali nodi nell'albero sono accettanti. Dopodichè, S accetta se si accerta che la radice dell'albero, corrispondente alla configurazione iniziale di M, è accettante.

La macchina S ha bisogno di spazio per memorizzare la pila per la ricorsione che viene usata nella visita in profondità. Ciascun livello della ricorsione memorizza una configurazione. La profondità della ricorsione rappresenta la complessità di tempo di M. Ciascuna configurazione usa spazio $O(f(n))$, e la complessità di tempo di M è $O(f(n))$. Quindi S usa spazio $O(f^2(n))$.

Possiamo migliorare la complessità di spazio notando che S non ha bisogno di memorizzare l'intera configurazione ad ogni livello della ricorsione. Al contrario, essa memorizza solo la scelta non deterministica che M ha fatto per raggiungere la configurazione a partire dal proprio genitore. Successivamente S può recuperare questa configurazione ripetendo la computazione dall'inizio e seguendo i 'segnaposto' memorizzati. Apportando questa modifica si riduce l'uso dello spazio ad una costante ad ogni livello della ricorsione. Il totale ora utilizzato è pertanto $O(f(n))$.

LEMMA 10.23

Per $f(n) \geq n$, risulta $\text{SPACE}(f(n)) \subseteq \text{ATIME}(f^2(n))$.

DIMOSTRAZIONE. Iniziamo con una macchina deterministica M di spazio $O(f(n))$ e costruiamo una macchina alternante S che usa tempo $O(f^2(n))$ per simularla. L'approccio è simile a quello utilizzato nella dimostrazione del teorema di Savitch (Teorema 8.5), dove abbiamo costruito una procedura generale per il problema della resa.

Nel problema della resa disponiamo delle configurazioni c_1 e c_2 di M e di un numero t. Dobbiamo verificare se M può andare da c_1 a c_2 entro t passi. Una procedura alternante per questo problema prima dirama esistenzialmente per individuare una configurazione c_m a metà strada tra c_1 e c_2. Poi dirama universalmente i due processi: uno che verifica ricorsivamente se si può andare da c_1 a c_m entro $t/2$ passi, e l'altro per verificare se si può andare da c_m a c_2 entro $t/2$ passi.

La macchina S usa questa procedura alternante ricorsiva per verificare se la configurazione iniziale può raggiungere una configurazione accettante entro $2^{df(n)}$ passi. Qui d viene scelto in modo tale che M abbia non più di $2^{df(n)}$ configurazioni entro il proprio limite di spazio.

Il tempo massimo utilizzato su ogni diramazione di questa procedura alternante è $O(f(n))$, per scrivere una configurazione ad ogni livello della ricorsione, per la profondità della ricorsione, che è $\log 2^{df(n)} = O(f(n))$. Quindi l'algoritmo computa in tempo alternante $O(f^2(n))$.

LEMMA 10.24

Per $f(n) \geq \log n$, risulta $\text{ASPACE}(f(n)) \subseteq \text{TIME}(2^{O(f(n))})$.

DIMOSTRAZIONE. Costruiamo un macchina deterministica S di tempo $2^{O(f(n))}$ per simulare una macchina M alternante di spazio $O(f(n))$. Su input w, il simulatore S costruisce il grafo che segue della computazione di M su w. I nodi sono le configurazioni di M su w che usano al più spazio $df(n)$, dove d è il fattore costante appropriato per M. Gli archi vanno da una configurazione a quelle configurazioni che essa può raggiungere in una singola mossa di M. Dopo aver costruito il grafo, S lo scandisce ripetutamente e marca certe configurazioni come accettanti. All'inizio solo le configurazioni accettanti reali di M vengono marcate in questo modo. Una configurazione che rappresenta una diramazione universale viene marcata come accettante se tutti i suoi figli sono così marcati, ed una configurazione esistenziale è marcata come accettante se almeno uno dei suoi figli è marcato. La macchina continua a scandire ed a marcare fino a quando una scansione non marca ulteriori nodi. Infine, S accetta se la configurazione iniziale di M su w è marcata.

Il numero di configurazioni di M su w è $2^{O(f(n))}$ perché $f(n) \geq \log n$. Pertanto, la taglia del grafo di configurazione è $2^{O(f(n))}$ e può essere costruito in tempo $2^{O(f(n))}$. Scandire il grafo una volta richiede essenzialmente lo stesso tempo. Il numero totale di scansioni è al più il numero di nodi nel grafo perché ciascuna scansione, ad eccezione di quella finale, marca almeno un nodo aggiuntivo. Quindi, il tempo totale utilizzato è $2^{O(f(n))}$.

LEMMA 10.25

Per $f(n) \geq \log n$, risulta $\text{ASPACE}(f(n)) \supseteq \text{TIME}(2^{O(f(n))})$.

DIMOSTRAZIONE. Mostriamo come simulare una macchina deterministica M di tempo $2^{O(f(n))}$ con una macchina di Turing alternante S che usa spazio $O(f(n))$. Questa simulazione è complicata perché lo spazio disponibile di S è molto meno rispetto alla taglia della computazione di M. In questo caso, S ha soltanto spazio sufficiente per memorizzare i puntatori ad un tableau per M su w, come raffigurato nella figura che segue.

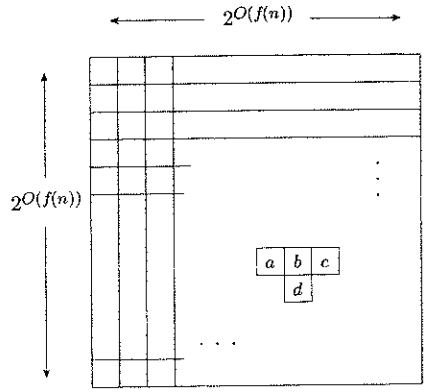


FIGURA 10.26
Un tableau per M su w

Utilizziamo la rappresentazione per le configurazioni fornita nella dimostrazione del Teorema 9.30, con cui un singolo simbolo può rappresentare sia lo stato della macchina che il contenuto della cella del nastro sotto la testina. I contenuti della cella d in Figura 10.26 sono poi determinati dai contenuti dei suoi genitori a, b e c. (Una cella sul bordo sinistro o destro ha soltanto due genitori.)

Il simulatore S opera ricorsivamente per ipotizzare e poi verificare i contenuti delle celle individuali del tableau. Per verificare i contenuti di una cella d al di fuori della prima riga, il simulatore S ipotizza esistenzialmente i contenuti dei genitori, verifica se i loro contenuti produrrebbero il contenuto di d in accordo alla funzione di transizione di M, e poi dirama universalmente per verificare queste previsioni ricorsivamente. Se d si trovasse nella prima riga, S verificherebbe la risposta direttamente perché conosce la configurazione iniziale di M. Assumiamo che M muova la testina all'estremità sinistra del nastro in caso di accettazione, per cui S può stabilire se M accetta w controllando i contenuti della cella più a sinistra in basso del tableau. Quindi S non necessita mai di memorizzare più di un singolo puntatore ad una cella nel tableau, pertanto usa spazio $\log 2^{O(f(n))} = O(f(n))$.

La gerarchia di tempo polinomiale

Le macchine alternanti forniscono un modo per definire una gerarchia naturale di classi all'interno della classe PSPACE.

DEFINIZIONE **10.27**

Sia i un numero naturale. Una ***macchina di Turing Σ_i-alternante*** è una macchina di Turing alternante che su tutti gli input e su tutte le diramazioni della computazione contiene al più i esecuzioni di passi universali o esistenziali, iniziando con passi esistenziali. Una ***macchina di Turing Π_i-alternante*** è simile eccetto che comincia con passi universali.

Si definisca $\mathbf{\Sigma_i TIME}(f(n))$ come la classe dei linguaggi che una TM Σ_i-alternante può decidere in tempo $O(f(n))$. Analogamente, si definisca la classe $\mathbf{\Pi_i TIME}(f(n))$ per macchine di Turing Π_i-alternanti, e si definiscano le classi $\mathbf{\Sigma_i SPACE}(f(n))$ e $\mathbf{\Pi_i SPACE}(f(n))$ per macchine di Turing alternanti limitate in spazio. Definiamo la ***gerarchia di tempo polinomiale*** come la collezione di classi

$$\mathbf{\Sigma_i P} = \bigcup_k \Sigma_i \mathrm{TIME}(n^k) \text{ e}$$
$$\mathbf{\Pi_i P} = \bigcup_k \Pi_i \mathrm{TIME}(n^k).$$

Si definisca la classe $\mathbf{PH} = \bigcup_i \Sigma_i \mathrm{P} = \bigcup_i \Pi_i \mathrm{P}$. Chiaramente, $\mathrm{NP} = \Sigma_1 \mathrm{P}$ e $\mathrm{coNP} = \Pi_1 \mathrm{P}$. In aggiunta *MIN-FORMULA* $\in \Pi_2 \mathrm{P}$.

10.4 SISTEMI DI PROVA INTERATTIVI

I sistemi di prova interattivi forniscono un modo per definire un analogo probabilistico della classe NP, così come gli algoritmi probabilistici di tempo polinomiale forniscono un analogo per P. Lo sviluppo dei sistemi di prova interattivi ha influenzato profondamente la teoria della complessità ed ha portato progressi importanti nei campi della crittografia e degli algoritmi di approssimazione. Per cominciare a comprendere questo concetto nuovo, rivisitiamo la nostra intuizione di NP.

I linguaggi in NP sono quelli i cui elementi posseggono tutti certificati di appartenenza corti che possono essere verificati facilmente. Qualora se ne sentisse il bisogno, si vada a pagina 314 e si riveda questa formulazione di NP. Riesprimiamo tale formulazione introducendo due entità: un Provatore che trova le prove di appartenenza, ed un Verificatore che le verifica. Si pensi al Provatore come se dovesse *convincere* il Verificatore dell'appartenenza di w ad A. Richiediamo che il Verificatore sia una macchina limitata

di tempo polinomiale; altrimenti potrebbe trovare la risposta da sola. Non imponiamo nessun limite computazionale al Provatore perché trovare la prova può richiedere tempo.

Si prenda il problema *SAT* per esempio. Un Provatore può convincere un Verificatore di tempo polinomiale che una formula ϕ è soddisfacibile fornendo un assegnamento che soddisfa la formula. Può un Provatore in modo analogo convincere un Verificatore computazionalmente limitato che una formula *non* è soddisfacibile? Non si sa se il complemento di *SAT* appartiene ad NP, quindi non possiamo contare sull'idea del certificato. Ciò nonostante, la risposta è sorprendentemente sì, posto che diamo al Provatore ed al Verificatore due caratteristiche aggiuntive. La prima, che diamo loro la possibilità di dar luogo ad un dialogo in *entrambi i sensi*. La seconda, che il Verificatore sia una macchina *probabilistica* di tempo polinomiale che acquisisce la risposta corretta con un grado molto alto, ma non assoluto, di certezza. Tali Provatore e Verificatore constituiscono un sistema di prova interattivo.

Non isomorfismo di grafi

Illustriamo il concetto di sistema di prova attraverso l'esempio elegante del problema dell'isomorfismo di grafi. Chiameremo i grafi G ed H ***isomorfi*** se i nodi di G possono essere riordinati in modo tale che G risulti identico ad H. Sia

$$ISO = \{\langle G, H\rangle \mid G \text{ ed } H \text{ sono grafi isomorfi}\}.$$

Sebbene *ISO* sia ovviamente in NP, fino ad ora ricerche approfondite hanno fallito sia nel fornire un algoritmo di tempo polinomiale per questo problema sia nel fornire una dimostrazione che esso risulta NP-completo. Si tratta di uno dei relativamente pochi linguaggi che si presentano in modo naturale in NP ma che non sono stati posizionati in una delle due categorie.

Qui consideriamo il linguaggio che è complementare a *ISO* - cioè, il linguaggio $NONISO = \{\langle G, H\rangle \mid G \text{ ed } H \text{ sono grafi } \textit{non} \text{ isomorfi}\}$. Non sappiamo se *NONISO* appartiene a NP perché non sappiamo come fornire certificati brevi che provano che i grafi non sono isomorfi. Ciò nonostante, come mostreremo, quando due grafi non sono isomorfi, un Provatore può convincere di ciò un Verificatore.

Si supponga di avere due grafi: G_1 e G_2. Se sono isomorfi, il Provatore può convincere il Verificatore di ciò presentando l'isomorfismo o riordinamento. Ma se non sono isomorfi, come può il Provatore convincere il Verificatore di ciò? Non si dimentichi: il Verificare non ha necessariamente fiducia nel Provatore, e quindi non è sufficiente per il Provatore *asserire* che non sono isomorfi. Il Provatore deve *convincere* il Verificatore. Si consideri il semplice protocollo che segue.

Il Verificatore sceglie in modo casuale G_1 o G_2 e poi riordina in modo casuale i suoi nodi per ottenere un grafo H. Il Verificatore invia H al Provatore. Il Provatore deve rispondere dichiarando se l'origine di H è stata G_1 o G_2. Ciò conclude il protocollo.

Se G_1 e G_2 sono realmente non isomorfi, il Provatore può sempre completare il protocollo perché il Provatore può sempre stabilire se H deriva da G_1 o da G_2. Tuttavia, se i grafi fossero isomorfi, H potrebbe essere stato generato sia da G_1 che da G_2. Quindi, anche con un potere computazionale illimitato, il Provatore non avrebbe una chance migliore di 50–50 di produrre la risposta corretta. Pertanto, se il Provatore è capace di rispondere correttamente (diciamo in 100 ripetizioni del protocollo), il Verificatore matura un'evidenza convincente che i grafi sono realmente non isomorfi.

Definizione del modello

Per definire formalmente il modello di sistema di prova interattiva, descriviamo il Verificatore, il Provatore, e la loro interazione. Sarà di aiuto tenere a mente l'esempio del non isomorfismo di grafi. Definiamo il ***Verificatore*** come una funzione V che computa la prossima trasmissione al Provatore dalla storia dei messaggi inviati fino ad ora. La funzione V ha tre input:

1. **La stringa di input.** L'obiettivo è stabilire se questa stringa è un elemento di qualche linguaggio. Nell'esempio *NONISO* la stringa di input codifica due grafi.
2. **L'input casuale.** Per comodità nel costruire la definizione, forniamo al Verificatore una stringa di input scelta in modo casuale invece della capacità equivalente di effettuare mosse probabilistiche durante le propria computazione.
3. **Storia parziale dei messaggi.** Una funzione non ha memoria del dialogo che è avvenuto fino ad ora, quindi forniamo la memoria esternamente attraverso una stringa che rappresenta lo scambio di messaggi fino all'istante attuale. Utilizziamo la notazione $m_1\#m_2\#\cdots\#m_i$ per rappresentare lo scambio di messaggi da m_1 fino ad m_i.

L'output del Verificatore è il messaggio successivo m_{i+1} nella sequenza oppure *accetta* o *rifiuta*, che denotano la conclusione dell'interazione. Perciò, V ha la forma funzionale $V\colon \Sigma^* \times \Sigma^* \times \Sigma^* \longrightarrow \Sigma^* \cup \{accetta, rifiuta\}$.

$V(w, r, m_1\#\cdots\#m_i) = m_{i+1}$ significa che la stringa di input è w, l'input casuale è r, la storia corrente dei messaggi è m_1 fino ad m_i, ed il prossimo messaggio del Verificatore al Provatore è m_{i+1}.

Il ***Provatore*** è una parte con capacità computazionale illimitata. Lo definiamo come una funzione P con due input:

1. **Stringa di input**
2. **Storia parziale dei messaggi**

L'output del Provatore è il prossimo messaggio per il Verificatore. Formalmente, P ha la forma $P\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$.

$P(w, m_1\#\cdots\#m_i) = m_{i+1}$ significa che il Provatore invia m_{i+1} al Verificatore dopo aver scambiato i messaggi da m_1 ad m_i fin ora.

Di seguito definiamo l'interazione tra il Provatore ed il Verificatore. Per specifiche stringhe w ed r, scriviamo $(V{\leftrightarrow}P)(w, r) = \textit{accetta}$ se esiste una sequenza di messaggi da m_1 ad m_k per qualche k per cui

1. per $0 \leq i < k$, dove i è un numero pari, $V(w, r, m_1\#\cdots\#m_i) = m_{i+1}$;
2. per $0 < i < k$, dove i è un numero dispari, $P(w, m_1\#\cdots\#m_i) = m_{i+1}$; ed
3. il messaggio finale m_k nella storia dei messaggi è *accetta*.

Per semplificare la definizione della classe IP, assumiamo che le lunghezze dell'input casuale del Verificatore e di ciascuno dei messaggi scambiati tra il Verificatore ed il Provatore siano $p(n)$ per qualche polinomio p che dipende soltanto dal Verificatore. Inoltre, assumiamo che il numero totale di messaggi scambiati sia al più $p(n)$. La definizione seguente dà la probabilità che un sistema di prova interattivo accetti una stringa di input w. Per ogni stringa w di lunghezza n, definiamo

$$\Pr\big[\, V{\leftrightarrow}P \text{ accetta } w \,\big] = \Pr\big[\, (V{\leftrightarrow}P)(w, r) = \textit{accetta} \,\big],$$

dove r è una stringa scelta in modo casuale di lunghezza $p(n)$.

DEFINIZIONE **10.28**

Diremo che il linguaggio A appartiene ad **IP** se esiste una funzione V calcolabile in tempo polinomiale tale che per qualche funzione P (arbitraria), per ogni funzione $\widetilde{P}$ e per ogni stringa w,

1. $w \in A$ implica $\Pr\big[\, V{\leftrightarrow}P \text{ accetta } w \,\big] \geq \frac{2}{3}$, e
2. $w \notin A$ implica $\Pr\big[\, V{\leftrightarrow}\widetilde{P} \text{ accetta } w \,\big] \leq \frac{1}{3}$.

In altre parole, se $w \in A$ allora qualche Provatore P (un Provatore "onesto") induce il Verificatore ad accettare con alta probabilità; ma se $w \notin A$, allora nessun Provatore (neanche un Provatore "truffaldino" $\widetilde{P}$) induce il Verificatore ad accettare con alta probabilità.

Possiamo amplificare la probabilità di successo di un sistema di prova interattivo attraverso la ripetizione, come abbiamo fatto nel Lemma 10.5, per rendere la probabilità di errore esponenzialmente piccola. Naturalmente IP contiene entrambe le classi NP e BPP. Abbiamo anche mostrato che essa contiene il linguaggio *NONISO*, di cui non è nota l'appartenenza nè a NP nè a BPP. Come mostreremo tra breve, IP è una classe sorprendentemente grande, che coincide con la classe PSPACE.

IP = PSPACE

In questa sezione dimostriamo uno dei teoremi più notevoli in teoria della complessità: l'uguaglianza tra le classi IP e PSPACE. Quindi, per ogni linguaggio in PSPACE, un Provatore può convincere un Verificatore probabilistico di tempo polinomiale dell'appartenenza di una stringa al linguaggio, anche se una dimostrazione di appartenenza convenzionale potrebbe avere lunghezza esponenziale.

TEOREMA 10.29

IP = PSPACE.

Dividiamo la dimostrazione del teorema in lemmi che provano l'inclusione in ciascuna delle direzioni. Il primo lemma mostra che IP $\subseteq$ PSPACE. Sebbene leggermente tecnica, la dimostrazione di questo lemma è una simulazione standard di un sistema di prova interattivo tramite una macchina di spazio polinomiale.

LEMMA 10.30

IP $\subseteq$ PSPACE.

DIMOSTRAZIONE. Sia A un linguaggio appartenente ad IP. Si assuma che il Verificatore V per A, scambi esattamente $p = p(n)$ messaggi quando l'input w ha lunghezza n. Costruiamo una macchina PSPACE che simula V. Prima di tutto, per ogni stringa w, definiamo

$$\Pr\big[\, V \text{ accetta } w \,\big] = \max_P \Pr\big[\, V{\leftrightarrow}P \text{ accetta } w \,\big].$$

Questo valore è almeno $\frac{2}{3}$ se w appartiene ad A, ed è al più $\frac{1}{3}$ altrimenti. Mostriamo come calcolare questo valore in spazio polinomiale. Si denoti con M_j una storia di messaggi $m_1\#\cdots\#m_j$. Generalizziamo la definizione dell'interazione di V e P in modo tale che inizi con un flusso arbitrario di messaggi M_j. Scriviamo $(V{\leftrightarrow}P)(w, r, M_j) = \textit{accetta}$ se possiamo estendere M_j con messaggi m_{j+1} fino a m_p in modo tale che

1. per $0 \leq i < p$, dove i è un numero pari, $V(w, r, m_1\#\cdots\#m_i) = m_{i+1}$;
2. per $j \leq i < p$, dove i è un numero dispari, $P(w, m_1\#\cdots\#m_i) = m_{i+1}$; ed
3. il messaggio finale m_p nella storia dei messaggi è *accetta*.

Si osservi che queste condizioni richiedono che i messaggi di V siano consistenti con i messaggi già presenti in M_j. Generalizzando ulteriormente le nostre definizioni precedenti, definiamo

$$\Pr\big[\, V{\leftrightarrow}P \text{ accetta } w \text{ partendo da } M_j \,\big] = \Pr_r\big[\, (V{\leftrightarrow}P)(w, r, M_j) = \textit{accetta} \,\big].$$

Qui e nel prosieguo di questa dimostrazione, la notazione $\Pr_r$ significa che la probabilità viene calcolata su tutte le stringhe r che sono consistenti con M_j. Se non esiste nessuna r, allora la probabilità vale 0. Definiamo poi

$$\Pr[\,V \text{ accetta } w \text{ partendo da } M_j\,] = \\ = \max_P \Pr[\,V \leftrightarrow P \text{ accetta } w \text{ partendo da } M_j\,].$$

Per ogni $0 \le j \le p$ e per ogni flusso di messaggi M_j, sia N_{M_j} definito induttivamente per j decrescente, a partire dai casi base $j = p$. Per un flusso di messaggi M_p che contiene p messaggi, sia $N_{M_p} = 1$ se M_p è consistente con i messaggi di V per qualche stringa r ed $m_p = \textit{accetta}$. Altrimenti, sia $N_{M_p} = 0$.

Per $j < p$ ed un flusso di messaggi M_j, definiamo N_{M_j} come segue.

$$N_{M_j} = \begin{cases} \max_{m_{j+1}} N_{M_{j+1}} & j < p, \quad j \text{ dispari} \\ \text{wt-avg}_{m_{j+1}} N_{M_{j+1}} & j < p, \quad j \text{ pari} \end{cases}$$

Qui $\text{wt-avg}_{m_{j+1}} N_{M_{j+1}}$ indica $\sum_{m_{j+1}} \left(\Pr_r[\,V(w, r, M_j) = m_{j+1}\,] \cdot N_{M_{j+1}}\right)$. L'espressione è la media di $N_{M_{j+1}}$, pesata dalla probabilità che il Verificatore abbia inviato il messaggio m_{j+1}.

Sia M_0 il flusso di messaggi vuoto. Facciamo due affermazioni sul valore N_{M_0}. La prima è che possiamo calcolare il valore N_{M_0} in spazio polinomiale. Ci riusciamo calcolando ricorsivamente N_{M_j} per ogni j ed M_j. Calcolare $\max_{m_{j+1}}$ è immediato. Per calcolare $\text{wt-avg}_{m_{j+1}}$, passiamo attraverso tutte le stringhe r di lunghezza p, ed eliminiamo quelle che portano il Verificatore a produrre un output che risulti inconsistente con M_j. Se non resta nessuna stringa r, allora $\text{wt-avg}_{m_{j+1}}$ è 0. Se resta qualche stringa, determiniamo la frazione delle stringhe rimanenti r che portano il Verificatore a dare in output m_{j+1}. Dopodichè pesiamo $N_{M_{j+1}}$ con questa frazione per calcolare il valore medio. La profondità della ricorsione è p, e pertanto è necessario soltanto spazio polinomiale.

La seconda affermazione è che N_{M_0} eguaglia la $\Pr[\,V \text{ accetta } w\,]$, il valore richiesto al fine di stabilire se w appartiene ad A. Dimostriamo questa seconda affermazione per induzione come segue.

FATTO 10.31

Per ogni $0 \le j \le p$ e per ogni M_j,

$$N_{M_j} = \Pr[\,V \text{ accetta } w \text{ partendo da } M_j\,].$$

Proviamo questa affermazione per induzione su j, dove la base è $j = p$ e l'induzione procede da p giù fino a 0.

Base: Si provi l'affermazione per $j = p$. Sappiamo che m_p è *accetta* o *rifiuta*. Se m_p è *accetta*, N_{M_p} è posto a 1, e

$\Pr[\,V \text{ accetta } w \text{ partendo da } M_j\,] = 1$ perché il flusso di messaggi già indica l'accettazione, quindi l'affermazione è vera. Il caso in cui m_p è *rifiuta* è simile.

Passo induttivo: Si assuma che l'asserto sia vero per qualche $j + 1 \le p$ e un flusso di messaggi M_{j+1}. Si provi che è vero per j e un flusso di messaggi M_j. Se j è pari, m_{j+1} è un messaggio da V a P. Valgono allora le eguaglianze:

$$\begin{aligned} N_{M_j} &\overset{1}{=} \sum_{m_{j+1}} \left(\Pr_r[\,V(w, r, M_j) = m_{j+1}\,] \cdot N_{M_{j+1}}\right) \\ &\overset{2}{=} \sum_{m_{j+1}} \left(\Pr_r[\,V(w, r, M_j) = m_{j+1}\,] \cdot \Pr[\,V \text{ accetta } w \text{ partendo da } M_{j+1}\,]\right) \\ &\overset{3}{=} \Pr[\,V \text{ accetta } w \text{ partendo da } M_j\,]. \end{aligned}$$

L'uguaglianza 1 è la definizione di N_{M_j}. L'uguaglianza 2 è basata sull'ipotesi induttiva. L'uguaglianza 3 segue dalla definizione di $\Pr[\,V \text{ accetta } w \text{ partendo da } M_j\,]$. Pertanto, l'affermazione vale se j è pari. Se j è dispari, m_{j+1} è un messaggio da P a V. Valgono allora le eguaglianze:

$$\begin{aligned} N_{M_j} &\overset{1}{=} \max_{m_{j+1}} N_{M_{j+1}} \\ &\overset{2}{=} \max_{m_{j+1}} \Pr[\,V \text{ accetta } w \text{ partendo da } M_{j+1}\,] \\ &\overset{3}{=} \Pr[\,V \text{ accetta } w \text{ partendo da } M_j\,]. \end{aligned}$$

L'uguaglianza 1 è la definizione di N_{M_j}. L'uguaglianza 2 usa l'ipotesi induttiva. Spezziamo l'uguaglianza 3 in due diseguaglianze. Risulta $\le$ perché il Provatore che massimizza la linea inferiore potrebbe mandare il messaggio m_{j+1} che massimizza la linea superiore. Risulta $\ge$ perché lo stesso Provatore non può far nulla di meglio che mandare quello stesso messaggio. L'invio di qualsiasi altra cosa al posto di un messaggio che massimizza la linea superiore abbasserebbe il valore risultante. Ciò prova l'affermazione per j dispari e completa una direzione della dimostrazione del Teorema 10.29.

Proviamo ora l'altra direzione del teorema. La prova di questo lemma introduce un metodo algebrico nuovo per analizzare la computazione.

LEMMA 10.32

PSPACE $\subseteq$ IP.

Prima di arrivare alla dimostrazione del lemma, proviamo un risultato più debole che illustra la tecnica. Definiamo il ***problema del conteggio***

per la soddisfacibilità come il linguaggio

$\#SAT = \{\langle \phi, k \rangle \mid \phi$ è una formula cnf con esattamente k assegnamenti che la soddisfano$\}$.

TEOREMA 10.33

$\#SAT \in$ IP.

IDEA. Questa prova presenta un protocollo in cui il Provatore persuade il Verificatore che k è il numero reale di assegnamenti che soddisfano una formula cnf ϕ data. Prima di descrivere il protocollo stesso, consideriamo un altro protocollo che somiglia a quello corretto ma non è adeguato perché richiede un Verificatore di tempo esponenziale. Assumiamo che ϕ abbia variabili da x_1 fino a x_m.

Sia f_i la funzione in cui, per $0 \leq i \leq m$ ed $a_1, \ldots, a_i \in \{0,1\}$, poniamo $f_i(a_1, \ldots, a_i)$ uguale al numero di assegnamenti che soddisfano ϕ tali che ogni $x_j = a_j$ per $j \leq i$. La funzione costante $f_0()$ rappresenta il numero di assegnamenti che soddisfano ϕ. La funzione $f_m(a_1, \ldots, a_m)$ è 1 se quegli a_i soddisfano ϕ; altrimenti, è 0. Un'identità facile vale per tutti gli $i < m$ ed $a_1, \ldots, a_i$:

$$f_i(a_1, \ldots, a_i) = f_{i+1}(a_1, \ldots, a_i, 0) + f_{i+1}(a_1, \ldots, a_i, 1).$$

Il protocollo per $\#SAT$ inizia con la fase 0 e finisce con la fase $m+1$. L'input è la coppia $\langle \phi, k \rangle$.

Fase 0. P invia $f_0()$ a V.
V verifica se $k = f_0()$ e *rifiuta* se non lo è.

Fase 1. P invia $f_1(0)$ ed $f_1(1)$ a V.
V verifica se $f_0() = f_1(0) + f_1(1)$ e *rifiuta* se non lo è.

Fase 2. P invia $f_2(0,0)$, $f_2(0,1)$, $f_2(1,0)$ ed $f_2(1,1)$ a V.
V verifica se $f_1(0) = f_2(0,0) + f_2(0,1)$ ed $f_1(1) = f_2(1,0) + f_2(1,1)$ e *rifiuta* se non lo è.

⋮

Fase m. P invia $f_m(a_1, \ldots, a_m)$ per ciascun assegnamento degli a_i.
V verifica le 2^{m-1} equazioni che collegano f_{m-1} con f_m e *rifiuta* se una di esse fallisce.

Fase m+1. V verifica che i valori $f_m(a_1, \ldots, a_m)$ siano corretti per ciascun assegnamento degli a_i valutando ϕ su ciascun assegnamento. Se tutti gli assegnamenti sono corretti, V *accetta*; altrimenti, V *rifiuta*. Ciò completa la descrizione del protocollo.

Questo protocollo non fornisce una prova che $\#SAT$ appartiene ad IP perché il Verificatore ha bisogno di tempo esponenziale solo per leggere i messaggi di lunghezza esponenziale che il Provatore invia. Tuttavia esaminiamone la correttezza perché ci può aiutare a capire il prossimo protocollo più efficiente.

Intuitivamente, un protocollo decide un linguaggio A se il Provatore può convincere il Verificatore dell'appartenenza di stringhe ad A. In altre parole, se una stringa è un elemento di A, un Provatore può portare il Verificatore ad accettare con alta probabilità. Se la stringa non è un elemento di A, nessun Provatore - neanche uno disonesto o malintensionato - può portare il Verificatore ad accettare con più che una piccola probabilità. Usiamo il simbolo P per designare il Provatore che segue correttamente il protocollo, e che con ciò porta V ad accettare con alta probabilità quando l'input appartiene ad A. Utilizziamo il simbolo $\widetilde{P}$ per indicare il Provatore che interagisce con il Verificatore quando l'input non appartiene ad A. Si pensi a $\widetilde{P}$ come ad un avversario - come se $\widetilde{P}$ tentasse di convincere V ad accettare quando V dovrebbe rifiutare. La notazione $\widetilde{P}$ evoca un Provatore truffaldino.

Nel protocollo $\#SAT$ appena descritto, il Verificatore ignora il proprio input casuale ed opera deterministicamente una volta che il Provatore è stato selezionato. Per dimostrare che il protocollo è corretto, proviamo due affermazioni. La prima, se k è il numero corretto di assegnamenti soddisfacenti per ϕ nell'input $\langle \phi, k \rangle$, un qualche Provatore P porta V ad accettare. Il Provatore che dà risposte accurate in ogni fase consegue il risultato. La seconda, se k non è corretto, qualsiasi Provatore $\widetilde{P}$ porta V a rifiutare. Argomentiamo questo caso come segue.

Se k non è corretto e $\widetilde{P}$ dà risposte accurate, V rifiuta immediatamente in fase 0 perché $f_0()$ è il numero di assegnamenti soddisfacenti per ϕ e pertanto $f_0() \neq k$. Per evitare che V rifiuti in fase 0, $\widetilde{P}$ deve inviare un valore scorretto per $f_0()$, denotato $\widetilde{f}_0()$. Intuitivamente, $\widetilde{f}_0()$ è una *bugia* sul valore di $f_0()$. Come nella vita reale, bugia genera bugia, e $\widetilde{P}$ è costretto a continuare a dire bugie sugli altri valori di f_i per evitare di essere scoperto durante le fasi successive. Alla fine queste bugie porteranno $\widetilde{P}$ ad essere scoperto in fase $m+1$, dove V verifica direttamente i valori di f_m.

Precisamente, poiché $\widetilde{f}_0() \neq f_0()$, almeno uno dei valori $f_1(0)$ ed $f_1(1)$ che $\widetilde{P}$ invia in fase 1 devono essere scorretti; altrimenti, V rifiuta quando verifica se $f_0() = f_1(0) + f_1(1)$. Diciamo che $f_1(0)$ è scorretto e chiamiamo $\widetilde{f}_1(0)$ il valore che viene inviato al suo posto. Procedendo in questo modo, vediamo che in ogni fase $\widetilde{P}$ deve finire con l'inviare qualche valore scorretto $\widetilde{f}_i(a_1, \ldots, a_i)$, o V rigetterebbe a quel punto. Ma quando V verifica il valore scorretto $\widetilde{f}_m(a_1, \ldots, a_m)$ in fase $m+1$, rifiuta comunque. Quindi abbiamo mostrato che, se k è scorretto, V rifiuta indipendentemente da cosa fa $\widetilde{P}$. Pertanto, il protocollo è corretto.

Il problema con questo protocollo è che il numero di messaggi raddoppia ad ogni fase. Il raddoppio si verifica perché il Verificatore richiede i due valori $f_{i+1}(\ldots, 0)$ ed $f_{i+1}(\ldots, 1)$ per confermare il singolo valore $f_i(\ldots)$.

Se potessimo trovare un modo per il Verificatore di confermare un valore di f_i con soltanto un singolo valore di f_{i+1}, il numero di messaggi non crescerebbe affatto. Possiamo farlo estendendo le funzioni f_i ad input non booleani e confermando il singolo valore $f_{i+1}(\ldots, z)$ per qualche z scelto in modo casuale da un campo finito.

DIMOSTRAZIONE. Sia ϕ una formula cnf con variabili da x_1 fino a x_m. Attraverso una tecnica chiamata ***aritmetizzazione***, associamo a ϕ un polinomio $p(x_1, \ldots, x_m)$ dove p emula ϕ simulando le operazioni booleane $\wedge$, $\vee$, e $\neg$ con le operazioni aritmetiche $+$ e $\times$ come segue. Se α e β sono sottoformule, sostituiamo le espressioni

$$\begin{array}{lll} \alpha \wedge \beta & \text{con} & \alpha\beta, \\ \neg\alpha & \text{con} & 1 - \alpha, \text{ e} \\ \alpha \vee \beta & \text{con} & \alpha * \beta = 1 - (1-\alpha)(1-\beta). \end{array}$$

Un'osservazione per quanto riguarda p che sarà importante nel seguito è che il grado di ciascuna delle sue variabili non è grande. Le operazioni $\alpha\beta$ e $\alpha * \beta$ producono ciascuna un polinomio di grado al più la somma dei gradi dei polinomi per α e β. Quindi, il grado di ciascuna delle variabili è al più n, la lunghezza di ϕ.

Se alle variabili di p vengono assegnati valori booleani, esso ha lo stesso valore di ϕ su quell'assegnamento. Valutare p quando alle variabili sono assegnati valori non booleani non ha una interpretazione ovvia in ϕ. Tuttavia, la dimostrazione usa comunque questi assegnamenti per analizzare ϕ, all'incirca come la dimostrazione del Teorema 10.13 usa assegnamenti non booleani per analizzare i programmi ramificati a lettura singola. Le variabili assumono valori su un campo finito $\mathcal{F}$ con q elementi, dove q è almeno 2^n.

Usiamo p per ridefinire le funzioni f_i che abbiamo definito nella sezione sull'idea della dimostrazione. Per $0 \leq i \leq m$ e per $a_1, \ldots, a_i \in \mathcal{F}$, sia

$$f_i(a_1, \ldots, a_i) = \sum_{a_{i+1}, \ldots, a_m \in \{0,1\}} p(a_1, \ldots, a_m).$$

Si noti che questa ridefinizione estende la definizione originale perché le due coincidono quando a_1 fino ad a_i assumono valori booleani. Quindi, $f_0()$ è ancora il numero di assegnamenti che soddisfano ϕ. Ciascuna delle funzioni $f_i(x_1, \ldots, x_i)$ può essere espressa come un polinomio in x_1 fino a x_i. Il grado di ciascuno di questi polinomi è al più quello di p.

Di seguito presentiamo il protocollo per $\#SAT$. All'inizio V riceve input $\langle \phi, k \rangle$ ed aritmetizza ϕ per ottenere il polinomio p. Tutta l'aritmetica è fatta sul campo $\mathcal{F}$ con q elementi, dove q è un primo più grande di 2^n. (Trovare questo primo q richiede un passo extra, ma qui ignoriamo questo punto perché la dimostrazione che daremo a breve del risultato più forte IP = PSPACE non lo richiede.) Un commento tra parentesi doppie è presente all'inizio della descrizione di ciascuna fase.

Fase 0. $[\![P \text{ invia } f_0().]\!]$
$P \to V$: P invia $f_0()$ a V.
V verifica che $k = f_0()$. V *rifiuta* se il controllo ha esito negativo.

Fase 1. $[\![P \text{ persuade } V \text{ che } f_0() \text{ è corretto se } f_1(r_1) \text{ è corretto.}]\!]$
$P \to V$: P invia i coefficienti di $f_1(z)$ visto come un polinomio in z.
V usa questi coefficienti per valutare $f_1(0)$ ed $f_1(1)$.
V controlla se $f_0() = f_1(0) + f_1(1)$ e *rifiuta* se non lo è.
(Ricorda che tutti i calcoli sono effettuati su $\mathcal{F}$.)
$V \to P$: V sceglie r_1 in modo casuale da $\mathcal{F}$ e lo invia a P.

Fase 2. $[\![P \text{ persuade } V \text{ che } f_1(r_1) \text{ è corretto se } f_2(r_1, r_2) \text{ è corretto.}]\!]$
$P \to V$: P invia i coefficienti di $f_2(r_1, z)$ visto come un polinomio in z.
V usa questi coefficienti per valutare $f_2(r_1, 0)$ ed $f_2(r_1, 1)$.
V verifica se $f_1(r_1) = f_2(r_1, 0) + f_2(r_1, 1)$ e *rifiuta* se non lo è.
$V \to P$: V sceglie r_2 in modo casuale da $\mathcal{F}$ e lo invia a P.
$\vdots$

Fase *i*. $[\![P \text{ persuade } V \text{ che } f_{i-1}(r_1, \ldots, r_{i-1}) \text{ è corretto se } f_i(r_1, \ldots, r_i) \text{ è corretto.}]\!]$
$P \to V$: P invia i coefficienti di $f_i(r_1, \ldots, r_{i-1}, z)$ visto come un polinomio in z.
V usa questi coefficienti per valutare $f_i(r_1, \ldots, r_{i-1}, 0)$ ed $f_i(r_1, \ldots, r_{i-1}, 1)$.
V verifica se $f_{i-1}(r_1, \ldots, r_{i-1}) = f_i(r_1, \ldots, r_{i-1}, 0) + f_i(r_1, \ldots, r_{i-1}, 1)$ e *rifiuta* se non lo è.
$V \to P$: V sceglie r_i in modo casuale da $\mathcal{F}$ e lo invia a P.
$\vdots$

Fase *m*+1. $[\![V \text{ verifica direttamente che } f_m(r_1, \ldots, r_m) \text{ è corretto.}]\!]$
V valuta $p(r_1, \ldots, r_m)$ per compararlo con il valore che V ha per $f_m(r_1, \ldots, r_m)$. Se sono uguali, V *accetta*; altrimenti, V *rifiuta*.

Ciò completa la descrizione del protocollo.

Facciamo vedere ora che questo protocollo decide $\#SAT$. Prima di tutto, se ϕ ha k assegnamenti soddisfacenti, V ovviamente accetta con certezza se il Provatore P segue il protocollo. In secondo luogo, mostriamo che se ϕ non ha k assegnamenti, nessun Provatore può farlo accettare con più di una bassa probabilità. Sia $\tilde{P}$ un qualsiasi Provatore.
Per evitare che V rifiuti immediatamente, $\tilde{P}$ deve inviare un valore scorretto $\tilde{f}_0()$ per $f_0()$ in fase 0. Pertanto, in fase 1, uno dei valori che V calcola per $f_1(0)$ ed $f_1(1)$ deve essere scorretto. Quindi, i coefficienti che $\tilde{P}$ invia per $f_1(z)$, visto come un polinomio in z, devono essere scorretti. Sia $\tilde{f}_1(z)$ la funzione che questi coefficienti rappresentano. Di seguito viene un passo chiave della dimostrazione.

Quando V sceglie un r_1 casuale in $\mathcal{F}$, asseriamo che è improbabile che $\widetilde{f}_1(r_1)$ sia uguale a $f_1(r_1)$. Per $n \geq 10$, mostriamo che

$$\Pr[\,\widetilde{f}_1(r_1) = f_1(r_1)\,] < n^{-2}.$$

Tale limitazione alla probabilità deriva dal Lemma 10.14: un polinomio in una singola variabile di grado al più d non può avere più di d radici, a meno che non assuma sempre valore 0. Pertanto, due polinomi qualsiasi in una sola variabile di grado al più d possono coincidere in al più d punti, a meno che non siano concordi ovunque.

Ricordiamo che il grado del polinomio per f_1 è al più n, e che V rifiuta se il grado del polinomio che $\widetilde{P}$ invia per $\widetilde{f}_1$ è più grande di n. Abbiamo già stabilito che queste funzioni non sono coincidenti ovunque, quindi il Lemma 10.14 implica che essi possono coincidere in al più n punti. La taglia di $\mathcal{F}$ è più grande di 2^n. La possibilità che r_1 risulti essere uno dei punti in cui le funzioni coincidono è al più $n/2^n$, che è meno di n^{-2} per $n \geq 10$.

Per ricapitolare ciò che abbiamo mostrato fino ad ora, se $\widetilde{f}_0()$ è scorretto, il polinomio per $\widetilde{f}_1$ deve essere scorretto, e quindi $\widetilde{f}_1(r_1)$ sarà verosimilmente scorretto in virtù dell'asserto precedente. Nell'evento improbabile che $\widetilde{f}_1(r_1)$ coincida con $f_1(r_1)$, $\widetilde{P}$ è stato "fortunato" in questa fase e sarà capace di far accettare V (anche se V dovrebbe rifiutare) seguendo le istruzioni per P nel resto del protocollo.

Continuando ulteriormente con l'argomento, se $\widetilde{f}_1(r_1)$ fosse sbagliato, almeno uno dei valori che V calcola per $f_2(r_1, 0)$ ed $f_2(r_1, 1)$ in fase 2 deve essere sbagliato, quindi i coefficienti che $\widetilde{P}$ invia per $f_2(r_1, z)$, visto come un polinomio in z, devono essere sbagliati. Sia $\widetilde{f}_2(r_1, z)$ invece la funzione che questi coefficienti rappresentano. I polinomi per $f_2(r_1, z)$ ed $\widetilde{f}_2(r_1, z)$ hanno grado al più n. Pertanto, come in precedenza, la probabilità che essi coincidano in un r_2 scelto in modo casuale in $\mathcal{F}$ è al più n^{-2}. Quindi, quando V sceglie r_2 in modo casuale, $\widetilde{f}_2(r_1, r_2)$ è verosimilmente scorretto.

Il caso generale discende in modo analogo per mostrare che per ciascun $1 \leq i \leq m$, se

$$\widetilde{f}_{i-1}(r_1, \ldots, r_{i-1}) \neq f_{i-1}(r_1, \ldots, r_{i-1}),$$

allora per $n \geq 10$ e per r_i scelto in modo casuale in $\mathcal{F}$,

$$\Pr[\,\widetilde{f}_i(r_1, \ldots, r_i) = f_i(r_1, \ldots, r_i)\,] \leq n^{-2}.$$

Pertanto, fornendo un valore scorretto per $f_0()$, $\widetilde{P}$ è forzato plausibilmente a dare valori scorretti per $f_1(r_1)$, $f_2(r_1, r_2)$, e così via fino a $f_m(r_1, \ldots, r_m)$. La probabilità che $\widetilde{P}$ risulti fortunato perché V sceglie un r_i, dove $\widetilde{f}_i(r_1, \ldots, r_i) = f_i(r_1, \ldots, r_i)$ anche se $\widetilde{f}_i$ ed f_i sono differenti in qualche fase, è al più il numero di fasi m per n^{-2} ovvero al più $1/n$. Se $\widetilde{P}$ non ha mai fortuna, invierà alla fine un valore scorretto per $f_m(r_1, \ldots, r_m)$.

Ma V verifica il valore di f_m direttamente in fase $m+1$ e cattura qualsiasi errore a quel punto. Quindi se k non è il numero di assegnamenti che soddisfano ϕ, nessun Provatore può portare il Verificatore ad accettare con probabilità maggiore di $1/n$.

Per completare la prova del teorema, dobbiamo solo mostrare che il Verificatore è probabilistico ed opera in tempo polinomiale, cose che risultano ovvie dalla sua descrizione.

Nel seguito torniamo alla prova del Lemma 10.32, che PSPACE $\subseteq$ IP. La prova è simile a quella del Teorema 10.33 eccetto per un'idea aggiuntiva usata qui per abbassare i gradi dei polinomi che si presentano nel protocollo.

IDEA. Proviamo prima di tutto l'idea che abbiamo usato nella dimostrazione precedente e identifichiamo il punto in cui la difficoltà si presenta. Per mostrare che ogni linguaggio in PSPACE appartiene ad IP, dobbiamo soltanto mostrare che il linguaggio $TQBF$, che è PSPACE-completo, appartiene ad IP. Sia ψ una formula booleana quantificata della forma

$$\psi = \mathsf{Q}_1 x_1\, \mathsf{Q}_2 x_2 \cdots \mathsf{Q}_m x_m\, [\phi],$$

dove ϕ è una formula cnf e ciascun Q_i è $\exists$ o $\forall$. Definiamo le funzioni f_i come prima, eccetto che ora prendiamo in considerazione i quantificatori. Per $0 \leq i \leq m$ ed $a_1, \ldots, a_m \in \{0,1\}$, sia

$$f_i(a_1, \ldots, a_i) = \begin{cases} 1 & \text{se } \mathsf{Q}_{i+1} x_{i+1} \cdots \mathsf{Q}_m x_m\, [\phi(a_1, \ldots, a_i)] \text{ è vera;} \\ 0 & \text{altrimenti,} \end{cases}$$

dove $\phi(a_1, \ldots, a_i)$ è ϕ con a_1 fino ad a_i che sostituiscono x_1 fino a x_i. Perciò, $f_0()$ è il valore di verità di ψ. Valgono allora le identità aritmetiche

$$\mathsf{Q}_{i+1} = \forall\colon \quad f_i(a_1, \ldots, a_i) = f_{i+1}(a_1, \ldots, a_i, 0) \cdot f_{i+1}(a_1, \ldots, a_i, 1) \quad \text{e}$$
$$\mathsf{Q}_{i+1} = \exists\colon \quad f_i(a_1, \ldots, a_i) = f_{i+1}(a_1, \ldots, a_i, 0) * f_{i+1}(a_1, \ldots, a_i, 1).$$

Si ricordi che abbiamo definito $x * y$ come $1 - (1-x)(1-y)$.

Una variante naturale del protocollo per $\#SAT$ porta a pensare allo stesso protocollo in cui estendiamo f_i ad un campo finito ed usiamo le identità per i quantificatori invece delle identità per la somma. Il problema con quest'idea è che quando aritmetizzata, ogni quantificatore può raddoppiare il grado del polinomio risultante. I gradi dei polinomi possono allora diventare esponenzialmente grandi, cosa che richiederebbe al Verificatore di restar in esecuzione per un tempo esponenziale al fine di elaborare i coefficienti in numero esponenziale che il Provatore dovrebbe spedire per descrivere i polinomi.

Per mantenere piccoli i gradi dei polinomi, introduciamo un'operazione di riduzione R, che riduce i gradi dei polinomi senza cambiare il loro comportamento su input booleani.

DIMOSTRAZIONE. Sia $\psi = \mathsf{Q}x_1 \cdots \mathsf{Q}x_m\ [\phi]$ una formula boolena quantificata, dove ϕ è una formula cnf. Per aritmetizzare ψ, introduciamo l'espressione

$$\psi' = \mathsf{Q}x_1\ Rx_1\ \mathsf{Q}x_2\ Rx_1Rx_2\ \mathsf{Q}x_3\ Rx_1Rx_2Rx_3\ \cdots\ \mathsf{Q}x_m\ Rx_1\cdots Rx_m\ [\phi].$$

Non ci si preoccupi al momento del significato di Rx_i. È utile solo per definire le funzioni f_i. Riscriviamo ψ' come

$$\psi' = \mathsf{S}_1 y_1\, \mathsf{S}_2 y_2 \cdots \mathsf{S}_k y_k\ [\phi],$$

dove ciascun $\mathsf{S}_i \in \{\forall, \exists, R\}$ e $y_i \in \{x_1, \ldots, x_m\}$.

Per ogni $i \le k$, definiamo la funzione f_i. Definiamo $f_k(x_1, \ldots, x_m)$ come il polinomio $p(x_1, \ldots, x_m)$ ottenuto aritmetizzando ϕ. Per $i < k$, definiamo f_i in termini di f_{i+1}:

$$\begin{aligned}
&\mathsf{S}_{i+1} = \forall\text{:} && f_i(\ldots) = f_{i+1}(\ldots, 0) \cdot f_{i+1}(\ldots, 1);\\
&\mathsf{S}_{i+1} = \exists\text{:} && f_i(\ldots) = f_{i+1}(\ldots, 0) * f_{i+1}(\ldots, 1);\\
&\mathsf{S}_{i+1} = R\text{:} && f_i(\ldots, a) = (1-a) f_{i+1}(\ldots, 0) + a f_{i+1}(\ldots, 1).
\end{aligned}$$

Se S_{i+1} è $\forall$ o $\exists$, f_i ha una variabile di input in meno di quante ne ha f_{i+1}. Se S_{i+1} è R, le due funzioni hanno lo stesso numero di variabili di input. Quindi, in generale, la funzione f_i non dipenderà da i variabili. Per evitare pedici ingombranti, utilizziamo "..." al posto di a_1 fino ad a_j per gli opportuni valori di j. Inoltre, riordiniamo gli input delle funzioni in modo tale che la variabile di input y_{i+1} risulti l'ultimo argomento.

Si noti che l'operazione Rx su polinomi non cambia i loro valori su input booleani. Pertanto, $f_0()$ è ancora il valore di verità di ψ. Tuttavia, si noti che l'operazione Rx produce un risultato che è lineare in x. Abbiamo aggiunto $Rx_1 \cdots Rx_i$ dopo $\mathsf{Q}_i x_i$ in ψ' al fine di ridurre ad 1 il grado di ciascuna variabile prima della quadratura per aritmetizzare Q_i.

Siamo ora pronti per descrivere il protocollo. Tutte le operazioni aritmetiche in questo protocollo sono su un campo finito $\mathcal{F}$ di dimensione almeno n^4, dove n è la lunghezza di ψ. V può trovare un primo di questa dimensione da solo, quindi P non ne deve fornire uno.

Fase 0. $[\![P \text{ invia } f_0().]\!]$
$P \to V$: P invia $f_0()$ a V.
V verifica se $f_0() = 1$ e *rifiuta* se non lo è.
⋮

Fase i. $[\![P \text{ persuade } V \text{ che } f_{i-1}(r_1 \cdots) \text{ è corretto se } f_i(r_1 \cdots, r) \text{ è corretto.}]\!]$
$P \to V$: P invia i coefficienti di $f_i(r_1 \cdots, z)$ visto come un polinomio in z.

(Qui $r_1 \cdots$ denota un assegnamento delle variabili con i valori $r_1, r_2, \ldots$ scelti in maniera casuale precedentemente.)
V usa questi coefficienti per valutare $f_i(r_1 \cdots, 0)$ ed $f_i(r_1 \cdots, 1)$.
V verifica che queste identità valgono:

$$f_{i-1}(r_1 \cdots) = \begin{cases} f_i(r_1 \cdots, 0) \cdot f_i(r_1 \cdots, 1) & \mathsf{S}_i = \forall, \\ f_i(r_1 \cdots, 0) * f_i(r_1 \cdots, 1) & \mathsf{S}_i = \exists, \end{cases}$$

e

$$f_{i-1}(r_1 \cdots, r) = (1-r) f_i(r_1 \cdots, 0) + r f_i(r_1 \cdots, 1) \quad \mathsf{S}_i = R.$$

Altrimenti, V *rifiuta*.
$V \to P$: V sceglie un r in modo casuale in $\mathcal{F}$ e lo invia a P.
(Quando $\mathsf{S}_i = R$, questo r sostituisce il precedente r.)
Va alla Fase $i+1$, dove P deve persuadere V che $f_i(r_1 \cdots, r)$ è corretto.
⋮

Fase k+1. $[\![V \text{ verifica direttamente che } f_k(r_1, \ldots, r_m) \text{ è corretto.}]\!]$
V valuta $p(r_1, \ldots, r_m)$ per compararlo con il valore che V ha per $f_k(r_1, \ldots, r_m)$. Se sono uguali, V *accetta*; altrimenti, V *rifiuta*.
Ciò completa la descrizione del protocollo.

Provare la correttezza di questo protocollo è simile a provare la correttezza del protocollo $\#SAT$. Chiaramente, se ψ è vera, P può seguire il protocollo e V accetterà. Se ψ è falsa, $\widetilde{P}$ deve mentire in fase 0 inviando un valore scorretto per $f_0()$. In fase i, se V ha un valore scorretto per $f_{i-1}(r_1 \cdots)$, uno dei valori $f_i(r_1 \cdots, 0)$ ed $f_i(r_1 \cdots, 1)$ deve essere scorretto ed il polinomio per f_i deve essere scorretto. Di conseguenza, per un r scelto in modo casuale , la probabilità che $\widetilde{P}$ sia fortunato in questa fase perché $f_i(r_1 \cdots, r)$ è corretto è al più il grado del polinomio diviso per la dimensione del campo, ovvero n/n^4. Il protocollo procede per $O(n^2)$ fasi, quindi la probabilità che $\widetilde{P}$ sia fortunato in qualche fase è al più $1/n$. Se $\widetilde{P}$ non è mai fortunato, V rifiuta in fase $k+1$.

10.5 COMPUTAZIONE PARALLELA

Un ***calcolatore parallelo*** può eseguire operazioni multiple simultaneamente. I calcolatori paralleli possono risolvere certi problemi

più velocemente dei ***calcolatori sequenziali***, che possono eseguire una sola operazione alla volta. In pratica, la distinzione tra i due è leggermente sfocata perché la maggior parte dei calcolatori reali (inclusi quelli "sequenziali") sono progettati per usare un qualche grado di parallelismo mentre eseguono istruzioni individuali. Ci soffermiamo qui sul parallelismo *massivo*, con cui un grosso numero (si pensi a milioni o più) di elementi di calcolo partecipano attivamente ad una singola computazione.

In questa sezione introduciamo brevemente la teoria della computazione parallela. Descriviamo un modello di calcolatore parallelo e lo usiamo per dare esempi di determinati problemi che si prestano bene alla parallelizzazione. Analizziamo anche la possibilità che il parallelismo possa non essere appropriato per certi altri problemi.

Circuiti booleani uniformi

Uno dei modelli più popolari nella ricerca teorica sugli algoritmi paralleli è la ***Macchina Parallela ad Accesso Casuale (Parallel Random Access Machine)*** o ***PRAM***. Nel modello PRAM, processori idealizzati con un insieme di istruzioni semplici, progettato a partire dai calcolatori reali, interagiscono attraverso una memoria condivisa. In questa breve sezione non possiamo descrivere le macchine PRAM in dettaglio. Invece, utilizziamo un modello alternativo di calcolatore parallelo che abbiamo introdotto per un altro scopo nel Capitolo 9: i circuiti booleani.

I circuiti booleani hanno diversi vantaggi e svantaggi come modello di calcolo parallelo. Sul versante positivo, il modello è semplice da descrivere, e ciò rende più facile fornire dimostrazioni. I circuiti portano con sè anche una somiglianza ovvia con i progetti hardware reali, ed in questo senso il modello è realistico. Sul versante negativo, i circuiti sono scomodi da "programmare" perché i processori individuali sono estremamente ridotti. Inoltre, non permettiamo cicli nella nostra definizione di circuiti booleani, in contrasto con i circuiti che possiamo realmente costruire.

Nel modello a circuito booleano di un calcolatore parallelo, consideriamo ciascuna porta come un processore individuale, quindi definiamo la ***complessità di processori*** di un circuito booleano come la sua *taglia*. Assumiamo che ciascun processore calcoli la sua funzione in un singolo passo di tempo, quindi definiamo la ***complessità di tempo parallelo*** di un circuito booleano come la sua *profondità*, o la distanza più lunga da una variabile di input alla porta di output.

Qualsiasi circuito specifico ha un numero fissato di variabili di input, quindi usiamo famiglie come stabilito nella Definizione 9.27 per decidere i linguaggi. Abbiamo necessità di imporre un requisito tecnico sulle famiglie di circuiti affinché esse corrispondano a modelli di calcolo parallelo come le macchine PRAM, dove una singola macchina è capace di gestire tutte le lunghezze dell'input. Tale requisito stabilisce che possiamo facilmente ottenere

tutti i membri in una famiglia di circuiti. Questo requisito di ***uniformità*** è ragionevole perché sapere che esiste un circuito piccolo per decidere certi elementi di un linguaggio non è molto utile se il circuito stesso è difficile da trovare. Ciò ci porta alla definizione che segue.

DEFINIZIONE **10.34**

Una famiglia di circuiti $(C_0, C_1, C_2, \ldots)$ è ***uniforme*** se qualche trasduttore di spazio logaritmico T dà in output $\langle C_n \rangle$ quando l'input di T è $\mathtt{1}^n$.

Si ricordi che la Definizione 9.28 ha formalizzato la complessità di taglia e di profondità dei linguaggi in termini di famiglie di circuiti di minima taglia e profondità. Qui consideriamo taglia e profondità *simultanee* di una singola famiglia di circuiti, al fine di stabilire quanti processori sono necessari per ottenere una particolare complessità di tempo parallelo o viceversa. Diremo che un linguaggio ha complessità circuitale di ***taglia e profondità simultanee*** al più $(f(n), g(n))$ se esiste una famiglia di circuiti uniforme per il linguaggio con complessità di taglia $f(n)$ e complessità di profondità $g(n)$.

ESEMPIO 10.35

Sia A un linguaggio su $\{\mathtt{0},\mathtt{1}\}$ che consiste di tutte le stringhe con un numero dispari di 1. Possiamo verificare l'appartenenza ad A calcolando la funzione di parità. Possiamo implementare la porta a due input per il calcolo della parità $x \oplus y$ con le operazioni standard AND, OR, e NOT come $(x \wedge \neg y) \vee (\neg x \wedge y)$. Siano $x_1, \ldots, x_n$ gli input del circuito. Un modo per ottenere un circuito per la funzione di parità è costruire le porte g_i in modo che $g_1 = x_1$ e $g_i = x_i \oplus g_{i-1}$ per $i \leq n$. Questa costruzione ha profondità e taglia $O(n)$.

L'Esempio 9.29 descrive un altro circuito per la funzione di parità con taglia $O(n)$ e profondità $O(\log n)$, costruendo un albero binario di porte $\oplus$. Questa costruzione costituisce un miglioramento significativo perché usa tempo parallelo esponenzialmente ridotto rispetto alla costruzione precedente. Pertanto, la complessità di taglia e profondità di A è $(O(n), O(\log n))$.

ESEMPIO 10.36

Si ricordi che possiamo usare i circuiti per calcolare funzioni che danno in output stringhe. Si consideri la funzione ***moltiplicazione di matrici booleane***. L'input ha $2m^2 = n$ variabili che rappresentano due matrici

$A = \{a_{ik}\}$ e $B = \{b_{ik}\}$ di dimensione $m \times m$. L'output consiste di m^2 valori che rappresentano la matrice $C = \{c_{ik}\}$ di dimensioni $m \times m$, dove

$$c_{ik} = \bigvee_j (a_{ij} \wedge b_{jk}).$$

Il circuito per questa funzione ha porte g_{ijk} che calcolano $a_{ij} \wedge b_{jk}$ per ogni i, j, e k. In aggiunta, per ogni i e k, il circuito contiene un albero binario di porte $\vee$ per calcolare $\bigvee_j g_{ijk}$. Ciascuno di questi alberi contiene $m - 1$ porte OR ed ha profondità $\log m$. Di conseguenza, questi circuiti per la moltiplicazione di matrici booleane hanno taglia $O(m^3) = O(n^{3/2})$ e profondità $O(\log n)$.

ESEMPIO 10.37

Se $A = \{a_{ij}\}$ è una matrice $m \times m$, definiamo la ***chiusura transitiva*** di A come la matrice

$$A \vee A^2 \vee \cdots \vee A^m,$$

dove A^i è la matrice prodotto di A con se stessa i volte e $\vee$ è l'OR bit a bit degli elementi della matrice. L'operazione di chiusura transitiva è strettamente correlata al problema *PATH* e quindi alla classe NL. Se A è la matrice di adiacenza di un grafo diretto G, A^i è la matrice di adiacenza del grafo con gli stessi nodi in cui un arco indica la presenza di un cammino di lunghezza i in G. La chiusura transitiva di A è la matrice di adiacenza del grafo in cui un arco indica la presenza di un cammino di qualsiasi lunghezza in G.

Possiamo rappresentare il calcolo di A^i con un albero binario di taglia i e profondità $\log i$ in cui un nodo calcola il prodotto delle due matrici al di sotto di esso. Ciascun nodo è calcolato da un circuito di taglia $O(n^{3/2})$ e profondità logaritmica. Quindi il circuito che calcola A^m ha taglia $O(n^2)$ e profondità $O(\log^2 n)$. Costruiamo circuiti per ciascun A^i, che aggiungono un altro fattore m alla taglia ed un livello aggiuntivo di profondità $O(\log n)$. Quindi la complessità di taglia e profondità della chiusura transitiva è $(O(n^{5/2}), O(\log^2 n))$.

La classe NC

Molti problemi interessanti hanno complessità di taglia e profondità $(O(n^k), O(\log^k n))$, per qualche costante k. Tali problemi possono essere considerati altamente parallelizzabili con un numero moderato di processori. Ciò suggerisce la definizione seguente.

DEFINIZIONE 10.38

Per $i \geq 1$, sia $\mathbf{NC}^i$ la classe dei linguaggi che possono essere decisi da una famiglia di circuiti uniforme[3] con taglia polinomiale e profondità $O(\log^i n)$. Sia $\mathbf{NC}$ la classe dei linguaggi che sono in NC^i per qualche i. Le funzioni che sono calcolate da queste famiglie di circuiti sono chiamate $\mathbf{NC}^i$ ***computabili*** o $\mathbf{NC}$ ***computabili***.[4]

Analizziamo la relazione di queste classi di complessità con altre classi di linguaggi che abbiamo incontrato. Prima di tutto, instauriamo una connessione tra spazio di una macchina di Turing e profondità di circuito. I problemi che sono risolvibili con profondità logaritmica sono risolvibili anche in spazio logaritmico. Viceversa, i problemi che sono risolvibili in spazio logaritmico, anche non deterministicamente, sono risolvibili con profondità logaritmica quadratica.

TEOREMA 10.39

$\mathrm{NC}^1 \subseteq \mathrm{L}$.

DIMOSTRAZIONE. Descriviamo a grandi linee un algoritmo di spazio logaritmico per decidere un linguaggio A in NC^1. Su input w di lunghezza n, l'algoritmo può costruire la descrizione dell'n-esimo circuito nella famiglia di circuiti uniforme per A. Successivamente l'algoritmo può valutare il circuito utilizzando una visita in profondità a partire dalla porta di output. È richiesta memoria per questa ricerca soltanto per tener traccia del cammino verso la porta attualmente esplorata, e per tener traccia dei risultati parziali che sono stati ottenuti lungo il cammino. Il circuito ha profondità logaritmica; quindi è richiesto soltanto spazio logaritmico dalla simulazione.

TEOREMA 10.40

$\mathrm{NL} \subseteq \mathrm{NC}^2$.

[3] Definire l'uniformità in termini di trasduttori di spazio logaritmico è standard per NC^i quando $i \geq 2$, ma dà un risultato non standard per NC^1 (che contiene la classe standard NC^1 come sottoinsieme). Nonostante ciò diamo questa definizione perché risulta più semplice ed adeguata per i nostri scopi.

[4] Steven Cook coniò il nome NC per "Nick's class" (la classe di Nick) poiché Nick Pippenger fu la prima persona a riconoscerne l'importanza.

IDEA. Si calcoli la chiusura transitiva del grafo delle configurazioni di una macchina NL. Si dia in output la posizione che corrisponde alla presenza di un cammino dalla configurazione iniziale alla configurazione di accettazione.

DIMOSTRAZIONE. Sia A un linguaggio che viene deciso da una macchina NL M, dove A è stato codificato nell'alfabeto $\{0,1\}$. Costruiamo una famiglia di circuiti uniforme $(C_0, C_1, \ldots)$ per A. Per ottenere C_n, costruiamo un grafo G che è simile al grafo della computazione per M su un input w di lunghezza n. Non conosciamo l'input w quando costruiamo il circuito - soltanto la sua lunghezza n. Gli input del circuito sono le variabili w_1 fino a w_n - ciascuna corrispondente ad una posizione nell'input.

Si ricordi che una configurazione di M su w descrive lo stato, il contenuto del nastro di lavoro, e le posizioni della testina sia del nastro di input che di quella del nastro di lavoro, ma non include w stesso. Quindi la collezione di configurazioni di M su w in realtà non dipende da w - solo dalla lunghezza n di w. Queste configurazioni, in numero polinomiale, formano i nodi di G.

Gli archi di G sono etichettati con le variabili di input w_i. Se c_1 e c_2 sono due nodi di G, e c_1 indica la posizione i della testina di input, collochiamo l'arco (c_1, c_2) in G con etichetta w_i (o $\overline{w_i}$) se c_1 può dare c_2 in un solo passo quando la testina di input sta leggendo un 1 (o 0), in base alla funzione di transizione di M. Se c_1 può dare c_2 in un solo passo, qualunque cosa la testina di input stia leggendo, collochiamo l'arco in G senza etichetta.

Se fissiamo gli archi di G in base a una stringa w di lunghezza n, esiste un cammino dalla configurazione iniziale a una configurazione di accettazione se e solo se M accetta w. Quindi, un circuito che calcola la chiusura transitiva di G e dà in output la posizione indicante la presenza di un tale cammino accetta esattamente quelle stringhe in A di lunghezza n. Tale circuito ha taglia polinomiale e profondità $O(\log^2 n)$.

Un trasduttore di spazio logaritmico è capace di costruire G e pertanto C_n su input 1^n. Si veda il Teorema 8.25 per una descrizione più dettagliata di un trasduttore di spazio logaritmico simile.

La classe dei problemi risolvibili in tempo polinomiale include tutti i problemi risolvibili in NC, come mostra il teorema seguente.

TEOREMA 10.41

NC $\subseteq$ P.

DIMOSTRAZIONE. Un algoritmo di tempo polinomiale può eseguire il trasduttore di spazio logaritmico per generare il circuito C_n e simularlo su un input di lunghezza n.

P-completezza

A questo punto consideriamo la possibilità che tutti i problemi in P si trovino anche in NC. L'uguaglianza tra queste classi sarebbe sorprendente perché implicherebbe che tutti i problemi risolvibili in tempo polinomiale sono altamente parallelizzabili. Introduciamo il fenomeno della P-completezza per fornire evidenza teorica che alcuni problemi in P sono inerentemente sequenziali.

DEFINIZIONE **10.42**

Un linguaggio B è **P-*completo*** se

1. $B \in$ P, e
2. ogni A in P è riducibile in spazio logaritmico a B.

Il prossimo teorema segue nello spirito il Teorema 8.23 ed ha una dimostrazione analoga perché le famiglie di circuiti NC possono calcolare riduzioni di spazio logaritmico. Lasciamo la sua prova come Esercizio 10.3.

TEOREMA 10.43

Se $A \leq_L B$ e B appartiene a NC, allora A appartiene a NC.

Mostriamo che il problema della valutazione di un circuito è P-completo. Per un circuito C ed un input x, indichiamo con $C(x)$ il valore di C su x. Sia

$$CIRCUIT\text{-}VALUE = \{\langle C, x\rangle \mid C \text{ è un circuito booleano e } C(x) = 1\}.$$

TEOREMA 10.44

CIRCUIT-VALUE è P-completo.

DIMOSTRAZIONE. La costruzione data nel Teorema 9.30 mostra come ridurre qualsiasi linguaggio A in P a *CIRCUIT-VALUE*. Su input w, la riduzione produce un circuito che simula la macchina di Turing di tempo polinomiale per A. L'input del circuito è w stesso. La riduzione può essere effettuata in spazio logaritmico perché il circuito che produce ha una struttura semplice e ripetitiva.

10.6 CRITTOGRAFIA

La pratica della cifratura, l'utilizzo di codici segreti per comunicazioni private, risale a migliaia di anni fa. Ai tempi dei Romani, Giulio Cesare cifrava i messaggi per i suoi generali per protezione contro la possibile intercettazione. Più recentemente, Alan Turing, l'inventore della macchina di Turing, ha guidato un gruppo di matematici britannici che hanno rotto il codice tedesco usato durante la Seconda Guerra Mondiale per inviare istruzioni agli U-boat che pattugliavano l'Oceano Atlantico. I governi ancora dipendono dai codici segreti ed investono notevoli risorse per progettare codici difficili da rompere, e nel trovare debolezze nei codici usati dagli altri. Ai giorni nostri società ed individui usano la cifratura per incrementare la sicurezza delle loro informazioni. Presto quasi tutte le comunicazioni elettroniche saranno protette crittograficamente.

Negli anni recenti la teoria della complessità computazionale ha portato ad una rivoluzione nella progettazione dei codici segreti. Il campo della crittografia, termine con cui quest'area è nota, ora si estende ben al di là dei codici segreti per la comunicazione privata e si rivolge ad una gamma molto ampia di questioni correlate alla sicurezza dell'informazione. Per esempio, oggi disponiamo della tecnologia per "firmare" digitalmente messaggi per autenticare l'identità del mittente; per permettere elezioni elettroniche in cui i partecipanti possono votare attraverso la rete ed i risultati possono essere conteggiati publicamente senza rivelare alcun voto individuale, prevenendo allo stesso tempo il voto multiplo ed altre violazioni; e per costruire nuovi tipi di codici segreti che non richiedono ai comunicanti di accordarsi in precedenza sugli algoritmi di cifratura e di decifratura.

La crittografia è un'applicazione pratica importante della teoria della complessità.

I telefoni cellulari, la trasmissione televisiva diretta via satellite, ed il commercio elettronico tramite Internet, dipendono tutti da misure crittografiche per proteggere l'informazione. Questi sistemi giocheranno presto un ruolo nelle vite della maggior parte delle persone. Infatti, la crittografia ha stimolato gran parte della ricerca nella teoria della complessità ed in altri campi matematici.

Chiavi segrete

Tradizionalmente, quando un mittente vuole cifrare un messaggio in modo tale che soltanto un certo ricevente possa decifrarlo, il mittente ed il ricevente condividono una ***chiave segreta***. La chiave segreta è un pezzo di informazione che viene usato dagli algoritmi di cifratura e di decifratura. Il mantenimento della segretezza della chiave è cruciale per la sicurezza del codice, poiché qualsiasi persona con accesso alla chiave può cifrare e decifrare messaggi.

Una chiave troppo corta può essere scoperta attraverso una ricerca di forza bruta nell'intero spazio di chiavi possibili. Anche una chiave piuttosto lunga può essere vulnerabile a certi tipi di attacco - diremo a breve qualcosa in più. L'unico modo per ottenere sicurezza crittografica perfetta è con chiavi che siano tanto lunghe quanto la lunghezza combinata di tutti i messaggi inviati.

Una chiave che è tanto lunga quanto la lunghezza del messaggio combinato viene detta ***one-time pad***. Essenzialmente, ogni bit di una chiave one-time pad viene usato soltanto una volta per cifrare un bit del messaggio, e poi quel bit della chiave viene scartato. Il problema principale con le one-time pad è che esse possono essere piuttosto grandi se si prevede una comunicazione in quantità significativa. Per la maggior parte degli scopi, le chiavi one-time pad sono troppo ingombranti per essere considerate pratiche.

Un codice crittografico che permette una quantità illimitata di comunicazioni sicure soltanto con chiavi di lunghezza moderata è preferibile. È interessante notare che tali codici non possono esistere in principio ma paradossalmente vengono usati in pratica. Questo tipo di codice non può esistere in principio perché una chiave che è significativamente più corta della lunghezza del messaggio combinato può essere trovata attraverso una ricerca di forza bruta nello spazio delle chiavi possibili. Pertanto, un codice che è basato su tali chiavi è vulnerabile in principio. Ma lì sta la soluzione al paradosso. Un codice potrebbe fornire comunque sicurezza adeguata in pratica perché una ricerca di forza bruta è estremamente lenta quando la chiave è moderatamente lunga – per dire, nell'ordine dei 100 bit. Naturalmente, se il codice può essere rotto in qualche altro modo più veloce, è insicuro e non dovrebbe essere utilizzato. La difficoltà sta nell'esser sicuri che il codice non possa essere rotto velocemente.

Al momento non abbiamo alcun modo per assicurare che un codice con chiavi di lunghezza moderata sia realmente sicuro. Per garantire che un codice non possa essere violato velocemente, avremmo bisogno di una *dimostrazione matematica* che, alla fin fine, la chiave non può essere trovata velocemente. Tuttavia, tali prove sembrano al di là delle capacità della matematica contemporanea! La ragione è che una volta che una chiave è stata scoperta, la verifica della sua correttezza può essere fatta facilmente ispezionando i messaggi che sono stati decifrati con essa. Pertanto, il problema della verifica della chiave può essere formulato come appartenente a P. Se potessimo dimostrare che le chiavi non possono essere trovate in tempo polinomiale, otterremmo un progresso matematico maggiore provando che P è diverso da NP.

Poiché non siamo capaci di dimostrare matematicamente che i codici sono inviolabili, confidiamo invece su evidenze circostanziate. In passato

l'evidenza sulla qualità di un codice era ottenuta assumendo esperti che tentavano di violarlo. Se fossero stati incapaci di farlo, la fiducia nella sua sicurezza sarebbe cresciuta. Un tale approccio ha ovvie mancanze. Se qualcuno ha esperti migliori dei nostri, o se non possiamo fidarci dei nostri stessi esperti, l'integrità del nostro codice potrebbe esser compromessa. Nonostante ciò, questo approccio era l'unico disponibile fino a poco tempo fa, ed è stato usato per supportare l'affidabilità di codici ampiamente usati quali il Data Encryption Standard (DES) che fu approvato dall'istituto nazionale per gli standard e la tecnologia degli Stati Uniti.

La teoria della complessità fornisce un altro modo per acquisire evidenza circa la sicurezza di un codice. Possiamo mostrare che la complessità di violare il codice è correlata alla complessità di un qualche altro problema per il quale un'evidenza fortissima di intrattabilità è già disponibile. Si ricordi che abbiamo usato la teoria della NP-completezza per fornire evidenza che certi problemi sono intrattabili. La riduzione di un problema NP-completo al problema della violazione di un codice mostrerebbe che il problema della violazione del codice è esso stesso NP-completo. Tuttavia, ciò non fornisce evidenza sufficiente per la sicurezza, perché la NP-completezza riguarda la complessità del caso pessimo. Un problema potrebbe essere NP-completo ma ancora facile da risolvere la maggior parte delle volte. I codici devono essere difficili da violare quasi sempre, quindi abbiamo bisogno di misurare la complessità del caso medio piuttosto che la complessità del caso pessimo.

Un problema che è generalmente ritenuto difficile nel caso medio è il problema della fattorizzazione degli interi. Matematici di prim'ordine sono stati interessati alla fattorizzazione per secoli, ma nessuno di loro ha ancora scoperto una procedura veloce per eseguirla. Alcuni codici moderni sono stati costruiti attorno al problema della fattorizzazione cosicché violare il codice corrisponde a fattorizzare un numero. Ciò costituisce un'evidenza convincente per la sicurezza di questi codici, perché un modo efficiente per violare un tale codice implicherebbe un algoritmo di fattorizzazione veloce, il che costituirebbe uno sviluppo notevole nella teoria computazionale dei numeri.

Crittosistemi a chiave pubblica

Anche quando le chiavi crittografiche sono moderatamente corte, la loro gestione rappresenta ancora un ostacolo ad un uso diffuso nella crittografia convenzionale. Un problema è che ogni coppia di parti che desidera una comunicazione privata deve stabilire una chiave segreta congiunta per questo scopo. Un altro problema è che ciascun individuo ha necessità di mantenere un database segreto di tutte le chiavi che sono state stabilite.

Lo sviluppo recente della crittografia a chiave pubblica fornisce una soluzione elegante ad entrambi i problemi. In un crittosistema convenzionale o ***crittosistema a chiave privata***, la stessa chiave viene usata sia per la

cifratura che per la decifratura. Si confronti ciò con il nuovo ***crittosistema a chiave pubblica*** per cui la chiave di decifratura è diversa, e non facilmente calcolabile, dalla chiave di cifratura.

Anche se si tratta di un'idea apparentemente semplice, la separazione delle due chiavi ha conseguenze profonde. Ora ciascun individuo ha soltanto bisogno di stabilire una singola coppia di chiavi: una chiave di cifratura E ed una chiave di decifratura D. L'individuo mantiene D segreta ma pubblicizza E. Se un altro individuo volesse inviargli un messaggio, cercherebbe E nella cartella pubblica, cifrerebbe con essa il messaggio, e glielo invierebbe. Il primo individuo è l'unico che conosce D, quindi soltanto lui potrebbe decifrare il messaggio.

Alcuni crittosistemi a chiave pubblica possono essere anche usati per produrre ***firme digitali***. Se un individuo applica il proprio algoritmo segreto di decifratura al messaggio prima di inviarlo, chiunque può verificare che esso realmente proviene da lui applicando l'algoritmo pubblico di cifratura. Egli ha così effettivamente "firmato" il messaggio. Questa applicazione assume che le funzioni di cifratura e di decifratura possano essere applicate in qualsiasi ordine, come accade nel caso del crittosistema RSA.

Funzioni one way

Analizziamo ora brevemente alcuni dei pilastri teorici della teoria moderna della crittografia, chiamati *funzioni one way* e *funzioni trapdoor*. Uno dei vantaggi nell'uso della teoria della complessità come fondamento della crittografia è che aiuta a chiarire le assunzioni che vengono fatte quando discutiamo di sicurezza. Assumendo l'esistenza di una funzione one way, possiamo costruire crittosistemi a chiave privata sicuri. Assumendo l'esistenza di funzioni trapdoor possiamo costruire crittosistemi a chiave pubblica. Entrambe le assunzioni hanno conseguenze teoriche e pratiche aggiuntive. Definiremo questi tipi di funzioni dopo aver fornito alcuni preliminari.

Una funzione $f\colon \Sigma^* \longrightarrow \Sigma^*$ ***preserva la lunghezza*** se le lunghezze di w ed $f(w)$ sono uguali per ogni w. Una funzione che preserva la lunghezza è una ***permutazione*** se non associa mai a due stringhe lo stesso punto; cioè, se $f(x) \neq f(y)$ ogni volta che $x \neq y$.

Si ricordi la definizione di macchina di Turing probabilistica data in Sezione 10.2. Diremo che una macchina di Turing probabilistica M calcola una ***funzione probabilistica*** $M\colon \Sigma^* \longrightarrow \Sigma^*$ dove, se w è un input ed x è un output, denoteremo con

$$\Pr[M(w) = x]$$

la probabilità che M si arresti in uno stato di accettazione con x sul suo nastro quando ha iniziato con input w. Si noti che M qualche volta può fallire

nell'accettare su un input w, quindi

$$\sum_{x \in \Sigma^*} \Pr\big[\, M(w) = x \,\big] \leq 1.$$

Giungiamo quindi alla definizione di funzione one way. In parole povere, una funzione è one way (a senso unico) se è facile da calcolare ma quasi sempre difficile da invertire. Nella definizione seguente f denota la funzione one way facilmente calcolata, ed M denota l'algoritmo probabilistico di tempo polinomiale che possiamo immaginare che tenti di invertire f. Definiamo prima le permutazioni one way perché il caso è in qualche modo più semplice.

DEFINIZIONE **10.45**

Una ***permutazione one way*** è una permutazione f con le due proprietà che seguono.

1. È calcolabile in tempo polinomiale.
2. Per ogni TM M probabilistica di tempo polinomiale, ogni k, ed ogni n sufficientemente grande, se prendiamo un w in modo casuale di lunghezza n e mandiamo in esecuzione M sull'input $f(w)$,

$$\Pr_{M,w}\big[\, M(f(w)) = w \,\big] \leq n^{-k}.$$

Qui $\Pr_{M,w}$ significa che la probabilità viene calcolata sulle scelte casuali effettuate da M e la scelta casuale di w.

Una ***funzione one way*** è una funzione f che preserva la lunghezza con le due proprietà che seguono.

1. È calcolabile in tempo polinomiale.
2. Per ogni TM M probabilistica di tempo polinomiale, ogni k, ed n sufficientemente grande, se prendiamo un w in modo casuale di lunghezza n e mandiamo in esecuzione M sull'input $f(w)$,

$$\Pr_{M,w}\big[\, M(f(w)) = y, \text{ dove } f(y) = f(w) \,\big] \leq n^{-k}.$$

Per permutazioni one way, qualsiasi algoritmo probabilistico di tempo polinomiale ha soltanto una piccola probabilità di invertire f; cioè, è improbabile calcolare w da $f(w)$. Per funzioni one way, è improbabile che un algoritmo probabilistico di tempo polinomiale risulti in grado di trovare un y a cui viene associato $f(w)$.

ESEMPIO 10.46

La funzione moltiplicazione *mult* è un candidato per una funzione one-way. Sia $\Sigma = \{0,1\}$; e per ogni $w \in \Sigma^*$ sia $mult(w)$ la stringa che rappresenta il prodotto della prima metà con la seconda di w. Formalmente,

$$mult(w) = w_1 \cdot w_2,$$

dove $w = w_1 w_2$ ed è tale che $|w_1| = |w_2|$, oppure $|w_1| = |w_2| + 1$ se $|w|$ è dispari. Le stringhe w_1 e w_2 sono trattate come numeri binari. Completiamo la rappresentazione di $mult(w)$ con alcuni 0 a sinistra in modo tale che abbia la stessa lunghezza di w. Nonostante i notevoli sforzi della ricerca nel problema della fattorizzazione di interi, non si conosce nessun algoritmo probabilistico di tempo polinomiale che possa invertire *mult*, anche su una frazione polinomiale degli input.

Se assumiamo l'esistenza di una funzione one way, possiamo costruire un crittosistema a chiave privata che si può dimostrare sicuro. La costruzione è troppo complicata per presentarla qui. Invece, illustriamo come implementare con una funzione one way una diversa applicazione crittografica.

Un'applicazione semplice di una funzione one way è un sistema di password che si può dimostrare sicuro. In un tipico sistema di password, un utente deve inserire una password per ottenere l'accesso a qualche risorsa. Il sistema mantiene un database delle password degli utenti in forma cifrata. Le password sóno cifrate per esser protette nel caso in cui il database venisse lasciato sprotetto per cause accidentali o per errori di progettazione. I database delle password sono spesso lasciati sprotetti, quindi vari programmi applicativi possono leggerli e verificare la presenza di password. Quando un utente inserisce una password, il sistema verifica che sia valida cifrandola per stabilire se combacia con la versione memorizzata nel database. Ovviamente, uno schema di cifratura che sia difficile da invertire è desiderabile perché rende difficile ottenere la password decifrata dalla forma cifrata. Una funzione one way è una scelta naturale per una funzione di cifratura di password.

Funzioni trapdoor

Non sappiamo se l'esistenza di una funzione one way da sola è sufficiente per costruire un crittosistema a chiave pubblica. Per ottenere una tale costruzione, usiamo un oggetto correlato chiamato *funzione trapdoor* (funzione con trabocchetto), che può essere invertita efficientemente in presenza di un'informazione speciale.

Prima di tutto dobbiamo discutere la nozione di funzione che indicizza una famiglia di funzioni. Se abbiamo una famiglia di funzioni $\{f_i\}$ per i in Σ^*, possiamo rappresentarla con la singola funzione $f\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$, dove

$f(i,w) = f_i(w)$, per ogni i e w. Chiamiamo f funzione di indicizzazione. Diciamo che f preserva la lunghezza se ciascuna delle funzioni indicizzate f_i preserva la lunghezza.

DEFINIZIONE 10.47

Una ***funzione trapdoor*** $f\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$ è una funzione di indicizzazione che preserva la lunghezza a cui sono associate una TM G ausiliaria, probabilistica di tempo polinomiale, ed una funzione ausiliaria $h\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. La terna f, G, ed h soddisfa le seguenti tre condizioni.

1. Le funzioni f ed h sono calcolabili in tempo polinomiale.
2. Per ogni TM E probabilistica di tempo polinomiale, ogni k ed ogni n sufficientemente grande, se prendiamo un output casuale $\langle i,t\rangle$ di G su 1^n ed un $w \in \Sigma^n$ scelto in modo casuale , allora

$$\Pr_{E,w}\big[\, E(i, f_i(w)) = y, \text{ dove } f_i(y) = f_i(w)\,\big] \leq n^{-k}.$$

3. Per ogni n, ogni w di lunghezza n, ed ogni output $\langle i,t\rangle$ di G che viene prodotto con probabilità non nulla per qualche input di G,

$$h(t, f_i(w)) = y, \text{ dove } f_i(y) = f_i(w).$$

La TM probabilistica G genera un indice i di una funzione nella famiglia degli indici e genera allo stesso tempo simultaneamente un valore t che permette di invertire f_i velocemente. La condizione 2 stabilisce che f_i è difficile da invertire in mancanza di t. La conditione 3 stabilisce che f_i è facile da invertire quando t è noto. La funzione h è la funzione che inverte.

ESEMPIO 10.48

Descriviamo qui la funzione trapdoor che sottende al ben noto crittosistema RSA. Diamo il trio f, G, ed h ad esso associato. La macchina generatrice opera come segue. Su input 1^n, sceglie in modo casuale due numeri di taglia n e verifica che siano primi. Se non sono primi, ripete la selezione fino a quando non riesce o raggiunge un limite temporale pre-specificato e riporta fallimento. Dopo aver trovato p e q, calcola $N = pq$ ed il valore $\phi(N) = (p-1)(q-1)$. Sceglie un numero casuale e tra 1 e $\phi(N)$, e controlla che il numero sia relativamente primo a $\phi(N)$. Se non lo è, l'algoritmo sceglie un altro numero e ripete il controllo. Successivamente, l'algoritmo calcola l'inverso moltiplicativo d di e modulo $\phi(N)$. Il calcolo è possibile perchè l'insieme dei numeri in $\{1,\ldots,\phi(N)\}$ che sono relativamente primi a $\phi(N)$ forma un gruppo rispetto all'operazione di moltiplicazione modulo $\phi(N)$. Infine, G dà in output $((N,e),d)$. L'indice della funzione f consiste dei due numeri N ed e. Sia

$$f_{N,e}(w) = w^e \bmod N.$$

La funzione inversa h è

$$h(d,x) = x^d \bmod N.$$

La funzione h inverte correttamente perché $h(d, f_{N,e}(w)) = w^{ed} \bmod N = w$.

Possiamo utilizzare una funzione trapdoor come la funzione trapdoor RSA per costruire un crittosistema a chiave pubblica come segue. La chiave pubblica è l'indice i generato dalla macchina probabilistica G. La chiave segreta è il valore corrispondente t. L'algoritmo di cifratura spezza il messaggio m in blocchi di taglia al più $\log N$. Per ogni blocco w, il mittente calcola f_i. La sequenza di stringhe risultante è il messaggio cifrato. Il ricevente usa la funzione h per ottenere il messaggio originale dal cifrato.

ESERCIZI

10.1 Si mostri che una famiglia di circuiti con profondità $O(\log n)$ è anche una famiglia di circuiti di taglia polinomiale.

10.2 Si mostri che 12 non è uno pseudoprimo perché non supera qualche test di Fermat.

10.3 Si provi che se $A \leq_{\mathrm{L}} B$ e B appartiene a NC, allora A appartiene a NC.

10.4 Si mostri che la funzione di parità con n input può essere calcolata da un programma ramificato che ha $O(n)$ nodi.

10.5 Si mostri che la funzione di maggioranza con n input può essere calcolata da un programma ramificato che ha $O(n^2)$ nodi.

10.6 Si mostri che qualsiasi funzione con n input può essere calcolata da un programma ramificato che ha $O(2^n)$ nodi.

A10.7 Si mostri che BPP $\subseteq$ PSPACE.

PROBLEMI

10.8 Sia BPL la collezione di linguaggi che vengono decisi da macchine di Turing probabilistiche di spazio logaritmico con probabilità di errore $\frac{1}{3}$. Si provi che BPL $\subseteq$ P.

10.9 Sia $EQ_{\text{BP}} = \{\langle B_1, B_2\rangle |\ B_1$ e B_2 sono programmi ramificati equivalenti$\}$. Si mostri che EQ_{BP} è coNP-completo.

10.10 Si definisca una ***macchina* ZPP** come una macchina di Turing probabilistica a cui sono permessi tre tipi di output su ciascuna delle sue diramazioni: *accetta*, *rifiuta*, e *?*. Una macchina ZPPM decide un linguaggio A se M dà in output la risposta corretta su ogni stringa di input w (*accetta* se $w \in A$ e *rifiuta* se $w \notin A$) con probabilità almeno $\frac{2}{3}$, ed M non dà mai in output la risposta sbagliata. Su ogni input, M può dare in output *?* con probabilità al più $\frac{1}{3}$. Inoltre, il tempo di esecuzione medio su tutte le diramazioni di M su w deve essere limitato da un polinomio nella lunghezza di w. Si mostri che RP $\cap$ coRP = ZPP, dove ZPP è la collezione dei linguaggi che sono riconosciuti da macchine ZPP.

10.11 Si mostri che se NP $\subseteq$ BPP, allora NP = RP.

10.12 Si provi che se A è un linguaggio regolare, esiste una famiglia di programmi ramificati $(B_1, B_2, \ldots)$ dove ciascun B_n accetta esattamente le stringhe in A di lunghezza n ed è limitato in dimensione da una costante per n.

10.13 Si provi che se A è un linguaggio appartenente ad L, esiste una famiglia di programmi ramificati $(B_1, B_2, \ldots)$ dove ciascun B_n accetta esattamente le stringhe in A di lunghezza n ed è limitato in dimensione da un polinomio in n.

A* **10.14** Si provi che per ogni intero $p > 1$, se p non è pseudoprimo, allora p fallisce il test di Fermat per almeno la metà di tutti i numeri in $\mathcal{Z}_p^+$.

10.15 Si ricordi che NP^{SAT} è la classe di linguaggi che sono decisi da macchine di Turing non deterministiche di tempo polinomiale con un oracolo per il problema della soddisfacibilità. Si mostri che $\text{NP}^{SAT} = \Sigma_2\text{P}$.

* **10.16** Si provi il piccolo teorema di Fermat, che è enunciato nel Teorema 10.6. (Suggerimento: Si consideri la sequenza $a^1, a^2, \ldots$. Cosa deve accadere e come?)

10.17 Si mostri che se PH = PSPACE, allora la gerarchia di tempo polinomiale ha soltanto un numero di livelli distinti finito.

10.18 Si mostri che se P = NP, allora P = PH.

10.19 Sia M una macchina di Turing probabilistica di tempo polinomiale, e sia C un linguaggio dove per $0 < \epsilon_1 < \epsilon_2 < 1$ fissati,

a. $w \notin C$ implica $\Pr[M \text{ accetta } w] \leq \epsilon_1$, e

b. $w \in C$ implica $\Pr[M \text{ accetta } w] \geq \epsilon_2$.

Si mostri che $C \in$ BPP. (Suggerimento: Si usi il risultato del Lemma 10.5.)

10.20 Sia $CNF_{\text{H}} = \{\langle\phi\rangle |\ \phi$ è una formula cnf soddisfacibile dove ciascuna clausola contiene un numero qualsiasi di letterali, ma al più un letterale negato$\}$. Il Problema 7.52 ha chiesto di mostrare che $CNF_{\text{H}} \in$ P. Ora si dia una riduzione di spazio logaritmico da *CIRCUIT-VALUE* a CNF_{H} per concludere che CNF_{H} è P-completo.

* **10.21** Un ***automa a pila a k testine*** (k-PDA) è un automa a pila deterministico con k testine di input a sola lettura a due vie ed una pila di lettura/scrittura. Si definisca la classe $\text{PDA}_k = \{A |\ A$ è riconosciuto da un k-PDA$\}$. Si mostri che $\text{P} = \bigcup_k \text{PDA}_k$. (Suggerimento: Si ricordi che P è uguale a spazio logaritmico alternante.)

* **10.22** Una ***formula booleana*** è un circuito booleano dove ogni porta ha soltanto un filo di output. La stessa variabile di input può apparire più volte in una formula booleana. Si provi che un linguaggio ha una famiglia di formule di dimensione polinomiale se e solo se appartiene ad NC^1. Si ignorino considerazioni sull'uniformità.

10.23 Sia A un linguaggio regolare su {0,1}. Si mostri che A ha complessità di taglia e profondità $(O(n), O(\log n))$.

SOLUZIONI SELEZIONATE

10.7 Se M è una TM probabilistica che computa in tempo polinomiale, possiamo modificare M in modo tale che effettui esattamente n^r lanci di moneta su ciascuna diramazione della propria computazione, per qualche costante r. Pertanto, il problema di determinare la probabilità che M accetti la sua stringa di input si riduce a contare quante diramazioni sono accettanti ed a confrontare questo numero con $\frac{2}{3}2^{(n^r)}$. Questo conteggio può essere effettuato utilizzando spazio polinomiale.

10.14 Diciamo che a è un *testimone* se su di esso fallisce il test di Fermat per p; cioè, se $a^{p-1} \not\equiv 1 \pmod{p}$. Sia $\mathcal{Z}_p^*$ l'insieme di tutti i numeri in $\{1, \ldots, p-1\}$ che sono relativamente primi a p. Se p non è pseudoprimo, ha un testimone a in $\mathcal{Z}_p^*$.

Si usi a per ottenere molti altri testimoni. Si trovi un unico testimone in $\mathcal{Z}_p^*$ per ciascun non testimone. Se $d \in \mathcal{Z}_p^*$ è un non testimone, risulta $d^{p-1} \equiv 1 \pmod{p}$. Quindi $(da \bmod p)^{p-1} \not\equiv 1 \pmod{p}$ e pertanto $da \bmod p$ è un testimone. Se d_1 e d_2 sono non testimoni distinti in $\mathcal{Z}_p^*$, allora $d_1 a \bmod p \neq d_2 a \bmod p$. Altrimenti, $(d_1 - d_2)a \equiv 0 \pmod{p}$, e quindi $(d_1 - d_2)a = cp$ per qualche intero c. Ma d_1 e d_2 appartengono a $\mathcal{Z}_p^*$, e pertanto $(d_1 - d_2) < p$, quindi $a = cp/(d_1 - d_2)$ e p hanno un fattore maggiore di 1 in comune, cosa impossibile perché a e p sono relativamente primi. Pertanto, il numero di testimoni in $\mathcal{Z}_p^*$ deve essere tanto grande quanto il numero di non testimoni in $\mathcal{Z}_p^*$, e di conseguenza almeno metà degli elementi di $\mathcal{Z}_p^*$ sono testimoni.

Successivamente, si mostri che ogni elemento b di $\mathcal{Z}_p^+$ che non è relativamente primo a p è un testimone. Se b e p hanno un fattore comune, allora b^e e p condividono quel fattore per qualsiasi $e > 0$. Pertanto, $b^{p-1} \not\equiv 1 \pmod{p}$. Dunque, è possibile concludere che almeno la metà degli elementi di $\mathcal{Z}_p^+$ sono testimoni.

Bibliografia selezionata

1. ADLEMAN, L. Two theorems on random polynomial time. In *Proceedings of the Nineteenth IEEE Symposium on Foundations of Computer Science* (1978), 75–83.
2. ADLEMAN, L. M., AND HUANG, M. A. Recognizing primes in random polynomial time. In *Proceedings of the Nineteenth Annual ACM Symposium on the Theory of Computing* (1987), 462–469.
3. ADLEMAN, L. M., POMERANCE, C., AND RUMELY, R. S. On distinguishing prime numbers from composite numbers. *Annals of Mathematics 117* (1983), 173–206.
4. AGRAWAL, M., KAYAL, N., AND SAXENA, N. PRIMES is in P. *The Annals of Mathematics*, Second Series, vol. 160, no. 2 (2004), 781–793.
5. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *Data Structures and Algorithms.* Addison-Wesley, 1982.
6. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, Tools.* Addison-Wesley, 1986.
7. AKL, S. G. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall International, 1989.
8. ALON, N., ERDÖS, P., AND SPENCER, J. H. *The Probabilistic Method.* John Wiley & Sons, 1992.
9. ANGLUIN, D., AND VALIANT, L. G. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences 18* (1979), 155–193.
10. ARORA, S., LUND, C., MOTWANI, R., SUDAN, M., AND SZEGEDY, M. Proof verification and hardness of approximation problems. In *Proceedings of the Thirty-third IEEE Symposium on Foundations of Computer Science* (1992), 14–23.
11. BAASE, S. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1978.
12. BABAI, L. E-mail and the unexpected power of interaction. In *Proceedings of the Fifth Annual Conference on Structure in Complexity Theory* (1990), 30–44.

13. BACH, E., AND SHALLIT, J. *Algorithmic Number Theory, Vol. 1.* MIT Press, 1996.

14. BALCÁZAR, J. L., DÍAZ, J., AND GABARRÓ, J. *Structural Complexity I, II.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1988 (I) and 1990 (II).

15. BEAME, P. W., COOK, S. A., AND HOOVER, H. J. Log depth circuits for division and related problems. *SIAM Journal on Computing 15*, 4 (1986), 994–1003.

16. BLUM, M., CHANDRA, A., AND WEGMAN, M. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters 10* (1980), 80–82.

17. BRASSARD, G., AND BRATLEY, P. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.

18. CARMICHAEL, R. D. On composite numbers p which satisfy the Fermat congruence $a^{P-1} \equiv P \mod P$. *American Mathematical Monthly 19* (1912), 22–27.

19. CHOMSKY, N. Three models for the description of language. *IRE Trans. on Information Theory 2* (1956), 113–124.

20. COBHAM, A. The intrinsic computational difficulty of functions. In *Proceedings of the International Congress for Logic, Methodology, and Philosophy of Science*, Y. Bar-Hillel, Ed., North-Holland, 1964, 24–30.

21. COOK, S. A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing* (1971), 151–158.

22. CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to Algorithms.* MIT Press, 1989.

23. EDMONDS, J. Paths, trees, and flowers. *Canadian Journal of Mathematics 17* (1965), 449–467.

24. ENDERTON, H. B. *A Mathematical Introduction to Logic.* Academic Press, 1972.

25. EVEN, S. *Graph Algorithms.* Pitman, 1979.

26. FELLER, W. *An Introduction to Probability Theory and Its Applications, Vol. 1.* John Wiley & Sons, 1970.

27. FEYNMAN, R. P., HEY, A. J. G., AND ALLEN, R. W. *Feynman lectures on computation.* Addison-Wesley, 1996.

28. GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability—A Guide to the Theory of NP-completeness.* W. H. Freeman, 1979.

29. GILL, J. T. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing 6*, 4 (1977), 675–695.

30. GÖDEL, K. On formally undecidable propositions in *Principia Mathematica* and related systems I. In *The Undecidable*, M. Davis, Ed., Raven Press, 1965, 4–38.

31. GOEMANS, M. X., AND WILLIAMSON, D. P. .878-approximation algorithms for MAX CUT and MAX 2SAT. In *Proceedings of the Twenty-sixth Annual ACM Symposium on the Theory of Computing* (1994), 422–431.

32. GOLDWASSER, S., AND MICALI, S. Probabilistic encryption. *Journal of Computer and System Sciences* (1984), 270–299.

33. GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof-systems. *SIAM Journal on Computing* (1989), 186–208.

34. GREENLAW, R., HOOVER, H. J., AND RUZZO, W. L. *Limits to Parallel Computation: P-completeness Theory.* Oxford University Press, 1995.

35. HARARY, F. *Graph Theory*, 2d ed. Addison-Wesley, 1971.

36. HARTMANIS, J., AND STEARNS, R. E. On the computational complexity of algorithms. *Transactions of the American Mathematical Society 117* (1965), 285–306.

37. HILBERT, D. Mathematical problems. Lecture delivered before the International Congress of Mathematicians at Paris in 1900. In *Mathematical Developments Arising from Hilbert Problems*, vol. 28. American Mathematical Society, 1976, 1–34.

38. HOFSTADTER, D. R. *Goedel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1979.

39. HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

40. IMMERMAN, N. Nondeterminstic space is closed under complement. *SIAM Journal on Computing 17* (1988), 935–938.

41. JOHNSON, D. S. The NP-completeness column: Interactive proof systems for fun and profit. *Journal of Algorithms 9*, 3 (1988), 426–444.

42. KARP, R. M. Reducibility among combinatorial problems. In *Complexity of Computer Computations* (1972), R. E. Miller and J. W. Thatcher, Eds., Plenum Press, 85–103.

43. KARP, R. M., AND LIPTON, R. J. Turing machines that take advice. *ENSEIGN: L'Enseignement Mathematique Revue Internationale 28* (1982).

44. KNUTH, D. E. On the translation of languages from left to right. *Information and Control* (1965), 607–639.

45. LAWLER, E. L. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1991.

46. LAWLER, E. L., LENSTRA, J. K., RINOOY KAN, A. H. G., AND SHMOYS, D. B. *The Traveling Salesman Problem.* John Wiley & Sons, 1985.

47. LEIGHTON, F. T. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes.* Morgan Kaufmann, 1991.

48. LEVIN, L. Universal search problems (in Russian). *Problemy Peredachi Informatsii 9*, 3 (1973), 115–116.

49. LEWIS, H., AND PAPADIMITRIOU, C. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

50. LI, M., AND VITANYI, P. *Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, 1993.

51. LICHTENSTEIN, D., AND SIPSER, M. GO is PSPACE hard. *Journal of the ACM* (1980), 393–401.

52. LUBY, M. *Pseudorandomness and Cryptographic Applications.* Princeton University Press, 1996.

53. LUND, C., FORTNOW, L., KARLOFF, H., AND NISAN, N. Algebraic methods for interactive proof systems. *Journal of the ACM 39*, 4 (1992), 859–868.

54. MILLER, G. L. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences 13* (1976), 300–317.

55. NIVEN, I., AND ZUCKERMAN, H. S. *An Introduction to the Theory of Numbers*, 4th ed. John Wiley & Sons, 1980.

56. PAPADIMITRIOU, C. H. *Computational Complexity.* Addison-Wesley, 1994.

57. PAPADIMITRIOU, C. H., AND STEIGLITZ, K. *Combinatorial Optimization (Algorithms and Complexity).* Prentice-Hall, 1982.

58. PAPADIMITRIOU, C. H., AND YANNAKAKIS, M. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences 43*, 3 (1991), 425–440.

59. POMERANCE, C. On the distribution of pseudoprimes. *Mathematics of Computation 37*, 156 (1981), 587–593.

60. PRATT, V. R. Every prime has a succinct certificate. *SIAM Journal on Computing 4*, 3 (1975), 214–220.

61. RABIN, M. O. Probabilistic algorithms. In *Algorithms and Complexity: New Directions and Recent Results*, J. F. Traub, Ed., Academic Press (1976) 21–39.

62. REINGOLD, O. Undirected st-connectivity in log-space. *Journal of the ACM 55*, 4 (2008), 1–24.

63. RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

64. ROCHE, E., AND SCHABES, Y. *Finite-State Language Processing.* MIT Press, 1997.

65. SCHAEFER, T. J. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences 16*, 2 (1978), 185–225.

66. SEDGEWICK, R. *Algorithms*, 2d ed. Addison-Wesley, 1989.

67. SHAMIR, A. IP = PSPACE. *Journal of the ACM 39*, 4 (1992), 869–877.

68. SHEN, A. IP = PSPACE: Simplified proof. *Journal of the ACM 39*, 4 (1992), 878–880.

69. SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing 26*, (1997), 1484–1509.

70. SIPSER, M. Lower bounds on the size of sweeping automata. *Journal of Computer and System Sciences 21*, 2 (1980), 195–202.

71. SIPSER, M. The history and status of the P versus NP question. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Theory of Computing* (1992), 603–618.

72. STINSON, D. R. *Cryptography: Theory and Practice.* CRC Press, 1995.

73. SZELEPCZÉNYI, R. The method of forced enumeration for nondeterministic automata, *Acta Informatica 26*, (1988), 279–284.

74. TARJAN, R. E. *Data structures and network algorithms*, vol. 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*, SIAM, 1983.

75. TURING, A. M. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings, London Mathematical Society*, (1936), 230–265.

76. ULLMAN, J. D., AHO, A. V., AND HOPCROFT, J. E. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

77. VAN LEEUWEN, J., Ed. *Handbook of Theoretical Computer Science A: Algorithms and Complexity.* Elsevier, 1990.

Indice analitico