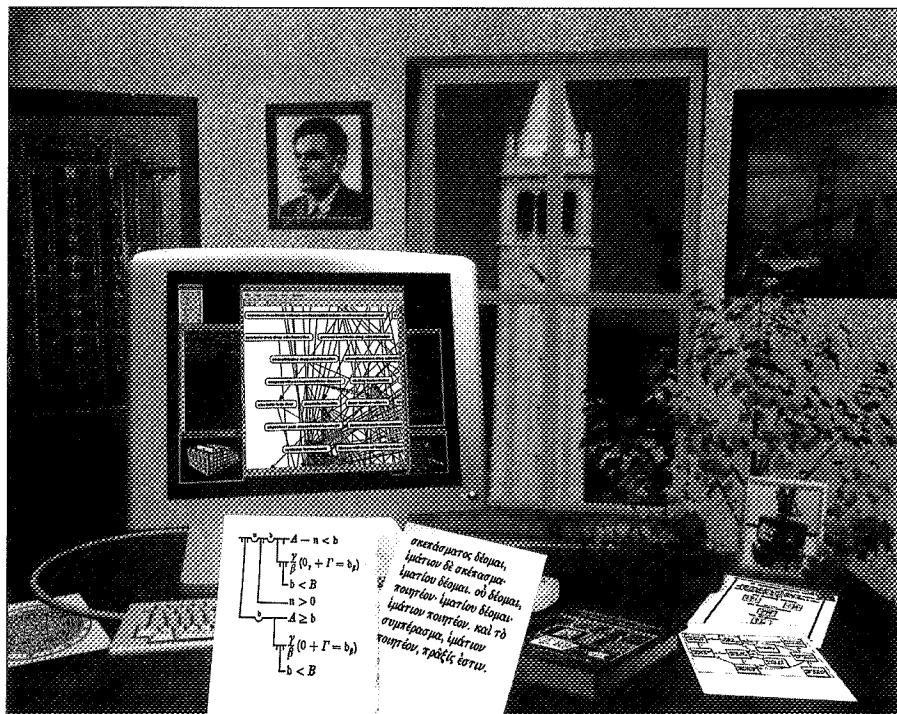


Stuart Russell Peter Norvig

Intelligenza artificiale

Un approccio moderno

seconda edizione



PEARSON
Prentice
Hall

Copyright © 2005 Pearson Education Italia S.r.l.
Via Fara, 28 - 20124 Milano
Tel. 02/6739761 Fax 02/673976503
E-mail: hpeitalia@pearson.com
Web: <http://hpe.pearsoned.it>

Authorized translation from the English language edition, entitled: Artificial Intelligence: A Modern Approach, 2nd Edition by Russell, Stuart; Norvig, Peter, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2003

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Italian language edition published by Pearson Education Italia Srl, Copyright © 2005

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Education Italia o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.

LA FOTOCOPIATURA DEI LIBRI È UN REATO L'editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume. Le richieste di riproduzione vanno inoltrate ad AIDRO (Associazione Italiana per i Diritti di Riproduzione delle Opere dell'Ingegno), Via delle Erbe, 2 - 20121 Milano - Tel. e Fax 02/80.95.06.

Traduzione e revisione tecnica: Stefano Gaburri

Copy-editing: Federica Sonzogno

Composizione: TOTEM di Andrea Astolfi

Stampa: Legoprint – Lavis (TN)

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

ISBN 88-7192-228-X

Printed in Italy

2^a edizione italiana: aprile 2005

La struttura dell'opera

		Volume	
		1	2
Parte I	Intelligenza artificiale		1 Introduzione
			2 Agenti intelligenti
Parte II	Risoluzione di problemi		3 Risolvere i problemi con la ricerca
			4 Ricerca informata ed esplorazione
			5 Problemi di soddisfacimento di vincoli
			6 Ricerca con avversari
Parte III	Conoscenza e ragionamento		7 Agenti logici
			8 Logica del primo ordine
			9 Inferenza nella logica del primo ordine
			10 Rappresentazione della conoscenza
Parte IV	Pianificazione		11 Pianificazione
			12 Pianificazione e azione nel mondo reale
Parte V	Conoscenza incerta e ragionamento		13 Incertezza
			14 Ragionamento probabilistico
			15 Ragionamento probabilistico nel tempo
			16 Decisioni semplici
			17 Decisioni complesse
Parte VI	Apprendimento		18 Apprendimento dalle osservazioni
			19 Conoscenza e apprendimento
			20 Metodi di apprendimento statistico
			21 Apprendimento per rinforzo
Parte VII	Comunicazione, percezione e azione		22 Comunicazione
			23 Elaborazione probabilistica del linguaggio
			24 Percezione
			25 Robotica
Parte VIII	Conclusioni		26 Fondamenti filosofici
			27 Presente e futuro dell'intelligenza artificiale
	Appendici		A Basi matematiche
			B Cenni sui linguaggi e sugli algoritmi

[Bibliografia](#)

[Indice analitico](#)

Nota dell'Editore

L'edizione italiana presenta, rispetto a quella inglese, alcune importanti modifiche quali la suddivisione dell'opera originale – veramente encyclopedica – in due volumi, e la parziale riorganizzazione strutturale degli argomenti presentati. Lo schema qui sopra riportato illustra sinteticamente le caratteristiche della nostra pubblicazione: i box di spunta indicano in quale dei due volumi sarà presente ciascun capitolo; come è possibile vedere, i Capitoli 1-2, 26-27 e le Appendici saranno presenti in entrambi i volumi.

Sono nati così due testi, autonomi e autoconsistenti, che rendono non solo più agevole la consultazione ma consentono anche una migliore fruibilità dei contenuti sia da parte degli studenti (che trovano gli argomenti strutturati secondo la nuova organizzazione dei corsi di laurea triennale e specialistica) sia da parte dei professionisti che vogliono estendere le conoscenze al di fuori dal proprio campo specialistico. Crediamo, in questo modo, di fornire un prezioso contributo per promuovere la conoscenza, la ricerca e la passione nei confronti di una disciplina così vasta e affascinante come l'intelligenza artificiale.

Sommario

I contenuti del volume

V

Parte Prima Intelligenza artificiale

Capitolo 1 Introduzione

1.1	Cos'è l'intelligenza artificiale?	4
	Agire umanamente: l'approccio del test di Turing	5
	Pensare come esseri umani: l'approccio della modellazione cognitiva	6
	Pensare razionalmente: l'approccio delle "leggi del pensiero"	7
	Agire razionalmente: l'approccio degli agenti razionali	8
1.2	I fondamenti dell'Intelligenza Artificiale	9
	Filosofia (428 a. C. – presente)	10
	Matematica (c. 800–presente)	12
	Economia (1776 – presente)	15
	Neuroscienze (1861–presente)	16
	Psicologia (1879 – presente)	19
	Ingegneria informatica (1940–presente)	21
	Teoria del controllo e cibernetica (1948–presente)	22
	Linguistica (1957–presente)	23
1.3	La storia dell'intelligenza artificiale	24
	La gestazione dell'intelligenza artificiale (1943–1955)	24
	La nascita dell'intelligenza artificiale (1956)	25
	Primi entusiasmi, grandi aspettative (1952–1969)	26
	Una dose di realtà (1966–1973)	30
	Sistemi basati sulla conoscenza: la chiave per il potere? (1969–1979)	32
	L'IA diventa un'industria (1980–presente)	34
	Il ritorno delle reti neurali (1986–presente)	35
	L'IA diventa una scienza (1987–presente)	36
	La comparsa degli agenti intelligenti (1995–presente)	38
	Lo stato dell'arte	38
1.4	Riepilogo	40
	Note storiche e bibliografiche	41
	Esercizi	42

Capitolo 2 Agenti intelligenti

2.1	Agenti e ambienti	46
2.2	Comportarsi correttamente: il concetto di razionalità	48
	Misure di prestazione	49
	Razionalità	50
	Onniscienza, apprendimento e autonomia	51
2.3	La natura degli ambienti	53
	Specificare un ambiente	53
	Proprietà degli ambienti	56
2.4	La struttura degli agenti	61
	Programmi agente	61
	Agenti reattivi semplici	63
	Agenti reattivi basati su modello	66
	Agenti basati su obiettivi	68
	Agenti basati sull'utilità	69
	Agenti capaci di apprendere	70
2.5	Riepilogo	73
	Note storiche e bibliografiche	74
	Esercizi	76

Parte Seconda Risoluzione di problemi

Capitolo 3 Risolvere i problemi con la ricerca

3.1	Agenti risolutori di problemi	82
	Problemi ben definiti e soluzioni	85
	La formulazione dei problemi	86
3.2	Problemi esemplificativi	87
	Problemi giocattolo	88
	Problemi reali	91
3.3	Cercare soluzioni	93
	Misurare le prestazioni nella risoluzione di problemi	96
3.4	Strategie di ricerca non informata	99
	Ricerca in ampiezza	99
	Ricerca a costo uniforme	100
	Ricerca in profondità	101
	Ricerca a profondità limitata	104
	Ricerca ad approfondimento iterativo	104
	Ricerca bidirezionale	107
	Confronto tra le strategie di ricerca non informata	108
3.5	Evitare ripetizioni negli stati	108
3.6	Ricerca con informazione parziale	111
	Problemi senza sensori	112
	Problemi di contingenza	114
3.7	Riepilogo	115
	Note storiche e bibliografiche	117
	Esercizi	119

Capitolo 4 Ricerca informata ed esplorazione

4.1	Strategie di ricerca informata o euristica	126
	Ricerca best-first greedy o "golosa"	127
	Ricerca A*: minimizzare il costo totale stimato della soluzione	129
	Ricerca euristica con memoria limitata	134
	Imparare a cercare meglio	138
4.2	Funzioni euristiche	139
	Effetto dell'accuratezza dell'euristica sulle prestazioni	140
	Inventare funzioni euristiche ammissibili	141
	Apprendere euristiche dall'esperienza	144
4.3	Algoritmi di ricerca locale e problemi di ottimizzazione	145
	Ricerca hill-climbing	146
	Simulated annealing	150
	Ricerca local beam	152
	Algoritmi genetici	153
4.4	Ricerca locale in spazi continui	156
4.5	Agenti per ricerca online e ambienti sconosciuti	160
	Problemi di ricerca online	161
	Agenti di ricerca online	162
	Ricerca locale online	165
	Apprendimento nella ricerca online	168
4.6	Riepilogo	168
	Note storiche e bibliografiche	169
	Esercizi	175

Capitolo 5 Problemi di soddisfacimento di vincoli

5.1	Problemi di soddisfacimento di vincoli	180
5.2	Ricerca con backtracking per CSP	184
	Ordinamento di variabili e valori	187
	Propagazione di informazioni attraverso i vincoli	188
	Backtracking intelligente: guardarsi indietro	192
5.3	Ricerca locale per problemi di soddisfacimento di vincoli	195
5.4	La struttura dei problemi	197
5.5	Riepilogo	201
	Note storiche e bibliografiche	202
	Esercizi	205

Capitolo 6 Ricerca con avversari

6.1	Giochi	209
6.2	Decisioni ottime nei giochi	211
	Strategie ottime	212
	L'algoritmo minimax	214
	Decisioni ottime nei giochi multiplayer	214
6.3	Potatura alfa-beta	217
6.4	Decisioni imperfette in tempo reale	221
	Funzioni di valutazione	221
	Tagliare la ricerca	224

classe 3 febbraio 07

6.5	Giochi che includono elementi casuali	227
	Valutazione della posizione nei giochi con nodi di possibilità	229
	La complessità di expectiminimax	230
	Giochi di carte	231
6.6	Lo stato dell'arte dei programmi di gioco	233
6.7	Discussione	237
6.8	Riepilogo	239
	Note storiche e bibliografiche	239
	Esercizi	244

Parte Terza Conoscenza e ragionamento

Capitolo 7 Agenti logici

7.1	Agenti basati sulla conoscenza	253
7.2	Il mondo del wumpus	255
7.3	Logica	259
7.4	Calcolo proposizionale: una logica molto semplice	263
	Sintassi	264
	Semantica	265
	Una semplice base di conoscenza	268
	Inferenza	268
	Equivalenza, validità e soddisfabilità	270
7.5	Schemi di ragionamento nel calcolo proposizionale	272
	Risoluzione	274
	Forma normale congiuntiva	277
	Un algoritmo di risoluzione	278
	Completezza della risoluzione	278
	Concatenazione in avanti e all'indietro	280
7.6	Inferenza proposizionale efficiente	284
	Un algoritmo con backtracking completo	284
	Algoritmi di ricerca locale	286
	Problemi di soddisfabilità difficili	288
7.7	Agenti basati sulla logica proposizionale	289
	Localizzare pozzi e mostri usando l'inferenza logica	290
	Tener traccia della posizione e dell'orientamento	291
	Agenti basati su circuiti	293
	Un confronto	297
7.8	Riepilogo	299
	Note storiche e bibliografiche	300
	Esercizi	303

Capitolo 8 Logica del primo ordine

8.1	Ancora sulla rappresentazione	308
8.2	Sintassi e semantica della logica del primo ordine	313
	Modelli per la logica del primo ordine	313
	Simboli e interpretazioni	315
	Termini	317
	Formule atomiche	317
	Formule complesse	318
	Quantificatori	318
	Quantificazione universale (\forall)	318
	Quantificazione esistenziale (\exists)	320
	Quantificatori nidificati	321
	Connessioni tra \forall e \exists	322
	Uguaglianza	323
8.3	Usare la logica del primo ordine	324
	Asserzioni e query nella logica del primo ordine	324
	Il dominio della parentela	325
	Numeri, insiemi e liste	327
	Il mondo del wumpus	329
8.4	Ingegneria della conoscenza nella logica del primo ordine	333
	Il processo di ingegneria della conoscenza	333
	Il dominio dei circuiti elettronici	335
8.5	Riepilogo	340
	Note storiche e bibliografiche	341
	Esercizi	342

Capitolo 9 L'inferenza nella logica del primo ordine

9.1	Inferenza proposizionale e inferenza del primo ordine	348
	Regole di inferenza per i quantificatori	348
	Riduzione all'inferenza proposizionale	349
9.2	Unificazione e lifting	351
	Una regola di inferenza del primo ordine	351
	Unificazione	353
	Memorizzazione e recupero di informazioni	354
9.3	Concatenazione in avanti	357
	Clausole definite del primo ordine	357
	Un semplice algoritmo di concatenazione in avanti	359
	Concatenazione in avanti efficiente	361
9.4	Concatenazione all'indietro	366
	Un algoritmo di concatenazione all'indietro	367
	Programmazione logica	368
	Implementazione efficiente di programmi logici	370
	Inferenza ridondante e cicli infiniti	373
	Programmazione logica con vincoli	375
9.5	Risoluzione	376
	Forma normale congiuntiva per la logica del primo ordine	377
	La risoluzione come regola di inferenza	379
	Alcuni esempi di dimostrazione	380
	Completezza della risoluzione	383

Gestire l'uguaglianza	386
Strategie di risoluzione	388
Dimostratori di teoremi	390
9.6 Riepilogo	395
Note storiche e bibliografiche	396
Esercizi	402
Capitolo 10 Rappresentazione della conoscenza	
10.1 Ingegneria ontologica	407
10.2 Categorie e oggetti	410
Composizione fisica	412
Misure	414
Oggetti e sostanze	416
10.3 Azioni, situazioni ed eventi	417
L'ontologia del calcolo delle situazioni	418
Descrivere le azioni nel calcolo delle situazioni	420
Risolvere il problema di rappresentazione del frame	422
Risolvere il problema inferenziale del frame	424
Tempo e calcolo degli eventi	425
Eventi generalizzati	426
Processi	428
Intervalli	430
Fluenti e oggetti	431
10.4 Eventi e oggetti mentali	433
Una teoria formale delle credenze	433
Conoscenza e credenza	436
Conoscenza, tempo e azione	436
10.5 Il mondo dello shopping su Internet	437
Confrontare le offerte	442
10.6 Sistemi di ragionamento per categorie	444
Reti semantiche	444
Logiche descrittive	448
10.7 Ragionare con informazione di default	450
Mondi aperti e mondi chiusi	450
La negazione come fallimento e la semantica del modello stabile	453
Circoscrizione e logica di default	454
10.8 Sistemi di mantenimento della verità	457
10.9 Riepilogo	460
Note storiche e bibliografiche	461
Esercizi	468

13

14 - Per 300 ore

15 - 16 SOM, lettura 09

Slide - 17,

Parte Quarta Pianificazione

Capitolo 11 Pianificazione

11.1 Il problema della pianificazione	478
Il linguaggio dei problemi di pianificazione	479
Espressività ed estensioni	481
Esempio: trasporto aereo di merci	483
Esempio: il problema della ruota di scorta	484
Esempio: il mondo dei blocchi	484
11.2 Pianificazione con ricerca nello spazio degli stati	486
Ricerca in avanti nello spazio degli stati	486
Ricerca all'indietro nello spazio degli stati	488
Euristiche per la ricerca nello spazio degli stati	490
11.3 Pianificazione con ordinamento parziale	492
Un esempio di pianificazione con ordinamento parziale	496
Pianificazione con ordinamento parziale con variabili libere	499
Euristiche per la pianificazione con ordinamento parziale	500
11.4 Grafi di pianificazione	501
Grafi di pianificazione per la stima euristica	504
L'algoritmo Graphplan	506
Terminazione di Graphplan	509
11.5 Pianificazione con la logica proposizionale	510
Descrivere problemi di pianificazione in logica proposizionale	510
Complessità delle codifiche proposizionali	514
11.6 Analisi degli approcci alla pianificazione	516
11.7 Riepilogo	518
Note storiche e bibliografiche	519
Esercizi	523

Capitolo 12 Pianificazione e azione nel mondo reale

12.1 Tempo, scheduling e risorse	529
Scheduling con vincoli sulle risorse	532
12.2 Pianificazione con reti gerarchiche	535
Rappresentare le scomposizioni di azioni	537
Modificare il pianificatore per gestire le scomposizioni	539
Discussione	543
12.3 Pianificazione e azione in ambienti non deterministici	545
12.4 Pianificazione condizionale	548
Pianificazione condizionale in ambienti completamente osservabili	548
Pianificazione condizionale in ambienti parzialmente osservabili	553
12.5 Monitoraggio dell'esecuzione e ripianificazione	558
12.6 Pianificazione continua	564
12.7 Pianificazione multiagente	569
Cooperazione: obiettivi e piani congiunti	570
Pianificazione multibody	571

Meccanismi di coordinamento	573
Competizione	574
12.8 Riepilogo	575
Note storiche e bibliografiche	576
Esercizi	581

Parte Ottava Conclusioni

Capitolo 26 Fondamenti filosofici

26.1 IA debole: le macchine possono agire in modo intelligente?	588
L'argomentazione derivante dall'incapacità	589
L'obiezione matematica	590
L'argomentazione derivante dall'informalità	592
26.2 IA forte: le macchine possono veramente pensare?	594
Il problema mente-corpo	597
L'esperimento del cervello nella vasca	598
L'esperimento della protesi cerebrale	599
La stanza cinese	601
26.3 L'etica e i rischi dello sviluppo di intelligenze artificiali	604
26.4 Riepilogo	609
Note storiche e bibliografiche	609
Esercizi	612

Capitolo 27 IA: presente e futuro

27.1 Componenti per agenti	614
27.2 Architetture di agenti	616
27.3 Stiamo andando nella giusta direzione?	618
27.4 E se l'IA avesse successo?	621

Appendice A Fondamenti matematici

A.1 Analisi di complessità e notazione O()	623
Analisi asintotica	623
NP e problemi intrinsecamente difficili	625
A.2 Vettori, matrici e algebra lineare	626
A.3 Distribuzioni di probabilità	628
Note storiche e bibliografiche	630

Appendice B Note sui linguaggi e gli algoritmi

B.1 Definire i linguaggi con la forma Backus-Naur (BNF)	631
B.2 Descrivere gli algoritmi con lo pseudocodice	632
B.3 Supporto online	633

Bibliografia	635
Indice analitico	665

Prefazione¹

L'intelligenza artificiale (IA) è un argomento molto vasto, e questo è un libro ponderoso. Abbiamo cercato di presentare l'intero panorama della disciplina, che racchiude la logica, la probabilità e la matematica del continuo, la percezione, il ragionamento, l'apprendimento e l'azione nonché tutto ciò che va dai dispositivi microelettronici ai robot per l'esplorazione planetaria. Le dimensioni del libro sono dovute anche al fatto che abbiamo cercato di presentare le nozioni con una certa profondità, sebbene nella parte principale di ogni capitolo trattiamo solo le idee più importanti, rimandando per ulteriori ricerche alle note bibliografiche.

Il sottotitolo di questo volume è “un approccio moderno”. Questa frase può sembrare alquanto vuota, ma ciò che intendiamo dire è che abbiamo cercato di presentare tutti gli argomenti in un contesto comune, invece di limitarci a esporre ogni aspetto dell'intelligenza artificiale nella sua specifica cornice storica. Chiediamo scusa agli specialisti dei singoli campi dell'IA se, in questo modo, le loro specifiche aree di ricerca dovessero risultare meno riconoscibili.

Il principale tema unificante è l'idea di **agente intelligente**. Nella nostra definizione, l'IA è lo studio degli agenti che ricevono percezioni dall'ambiente ed eseguono azioni. Ogni agente implementa una funzione che mette in corrispondenza sequenze percettive e azioni, e il nostro scopo è presentare diverse tecniche per rappresentare tali funzioni: alcune di queste sono i sistemi di produzione, gli agenti reattivi, i pianificatori condizionali in tempo reale, le reti neurali e i sistemi basati sulla teoria delle decisioni. Verrà inoltre spiegato il ruolo dell'apprendimento nell'estendere il campo d'azione del progettista in territori sconosciuti e illustrato come tale ruolo rappresenti un vincolo sulla progettazione degli agenti, favorendo la rappresentazione esplicita della conoscenza e del ragionamento. La robotica e la visione non sono trattati come problemi indipendenti, ma nella loro funzione al servizio del raggiungimento degli obiettivi. Viene inoltre posto l'accento sull'importanza dell'ambiente nel determinare l'architettura di agente più appropriata.

Il nostro scopo principale è trasmettere le *idee* emerse negli ultimi cinquant'anni di ricerca nel campo dell'IA, e nei due precedenti millenni di pensiero. Abbiamo cercato di evitare eccessivi formalismi nella presentazione dei concetti, mantenendo tuttavia la precisione. Quando l'abbiamo ritenuto appropriato, abbiamo reso l'esposizione più concreta includendo gli algoritmi sotto forma di pseudocodice; quest'ultimo è descritto brevemente nell'Appendice B. Implementazioni in diversi linguaggi di programmazione sono disponibili sul sito web del libro, aima.cs.berkeley.edu.

¹ Abbiamo scelto di riportare integralmente in queste pagine il testo della Prefazione dell'edizione originale – che fornisce una panoramica sulla struttura dell'opera e sul contenuto di tutti i capitoli – per offrire al lettore una visione complessiva del testo di Stuart Russell e Peter Norvig. L'edizione italiana è stata suddivisa in due volumi, come dettagliato a pagina V (N.d.E.).

Il libro è principalmente rivolto a un corso o a una serie di corsi universitari. Può essere usato anche per uno studio successivo alla laurea, magari integrandolo con alcune delle fonti suggerite nella bibliografia. Data l'estensiva copertura degli argomenti e il gran numero di algoritmi descritti in dettaglio, il volume può anche essere utile come primo riferimento per studenti laureati nel campo dell'IA o professionisti che vogliono estendere le conoscenze al di fuori dal proprio campo specialistico. L'unico requisito è quello di possedere una certa familiarità con i concetti base dell'informatica (algoritmi, strutture dati, complessità) al livello di uno studente universitario di secondo anno. Per comprendere i dettagli teorici delle reti neurali e dell'apprendimento statistico sarà anche necessario possedere una conoscenza introduttiva dell'analisi matematica: alcune nozioni base sono fornite nell'Appendice A.

Una visione d'insieme

Il libro è suddiviso in otto parti. La prima fornisce una panoramica della disciplina basata sul concetto di agente intelligente, definito come un sistema in grado di decidere cosa deve fare e quindi intraprendere le azioni necessarie. La Parte II, **Risoluzione di problemi**, si concentra sui metodi utilizzati quando è necessario "pensare nel futuro", per esempio quando si deve esplorare una zona sconosciuta o giocare a scacchi. La Parte III, **Conoscenza e ragionamento**, tratta i vari modi di rappresentare la conoscenza riguardante il mondo – come funziona, come si presenta, e i possibili effetti delle azioni – e come ragionare logicamente su tale conoscenza. La Parte IV, **Pianificazione**, discute quindi come applicare tali metodi di ragionamento per decidere che cosa fare, in particolar modo costruendo *piani*. La Parte V, **Conoscenza incerta e ragionamento**, è analoga alle due precedenti, ma si concentra sul ragionamento e il processo decisionale in condizioni di *incertezza* sul mondo, come potrebbe accadere in un sistema di diagnosi medica.

Insieme, le Parti II–V descrivono la parte dell'agente intelligente responsabile del processo decisionale. La Parte VI, **Apprendimento**, descrive i metodi che possono arrivare a generare la conoscenza necessaria per il loro funzionamento. La Parte VII, **Comunicazione, percezione e azione**, descrive le modalità con cui l'agente intelligente può percepire l'ambiente attraverso la visione, il tatto, l'udito o la comprensione del linguaggio, e i modi in cui può trasformare i piani in azioni, attraverso il movimento robotizzato o la formazione di frasi in linguaggio naturale. L'ultima parte, infine, analizza il passato e il futuro dell'intelligenza artificiale e le sue implicazioni etiche e filosofiche.

Cambiamenti dalla prima edizione²

Molto è cambiato nell'IA dalla pubblicazione della prima edizione nel 1995, e molto è cambiato in questo libro. Ogni capitolo è stato riscritto per riflettere gli ultimi sviluppi della ricerca, per reinterpretare i risultati già acquisiti alla luce delle nuove scoperte o per migliorare didatticamente la presentazione delle idee. Gli studenti di IA dovrebbero essere incoraggiati dal fatto che le tecniche correnti sono molto più pratiche di quelle del 1995; gli algoritmi di pianificazione

² La precedente edizione di questo testo era stata pubblicata da UTET (N.d.E.).

presentati nella prima edizione, ad esempio, potevano generare piani di qualche dozzina di passi, mentre quelli che presentiamo ora possono scalare fino a decine di migliaia di passi. Miglioramenti simili, misurabili in termini di ordini di grandezza, si possono riscontrare nell'inferenza probabilistica, nell'interpretazione del linguaggio e in altre discipline. I cambiamenti più notevoli sono riportati qui sotto.

- ◆ Nella Parte I riconosciamo il contributo storico della teoria del controllo, della teoria dei giochi, dell'economia e delle neuroscienze. Questo ci aiuta a gettare le basi per un trattamento più integrato di queste idee nei capitoli successivi.
- ◆ Nella Parte II trattiamo gli algoritmi di ricerca online e abbiamo aggiunto un capitolo dedicato al soddisfacimento di vincoli, che rappresenta un collegamento naturale al materiale dedicato alla logica.
- ◆ Nella Parte III, il calcolo proposizionale – precedentemente presentato come un argomento propedeutico alla logica del primo ordine – viene considerato ora a pieno titolo un utile linguaggio di rappresentazione, grazie a veloci algoritmi di inferenza e alla progettazione di agenti basati su circuiti. I capitoli sulla logica del primo ordine sono stati riorganizzati per presentare più chiaramente gli argomenti e abbiamo aggiunto un nuovo esempio: il dominio dello shopping su Internet.
- ◆ Nella Parte IV abbiamo incluso metodi di pianificazione più recenti come GRAPHPLAN e la pianificazione basata sulla soddisficiabilità; abbiamo inoltre approfondito la trattazione dello scheduling e della pianificazione condizionale, gerarchica e multiagente.
- ◆ Nella Parte V abbiamo arricchito il materiale sulle reti Bayesiane con nuovi algoritmi come l'eliminazione delle variabili e l'algoritmo Monte Carlo applicato alle catene di Markov; inoltre abbiamo aggiunto un capitolo dedicato al ragionamento temporale in condizioni di incertezza che include modelli di Markov nascosti, filtri di Kalman e reti Bayesiane dinamiche. La trattazione dei processi decisionali di Markov è più approfondita, e abbiamo aggiunto sezioni sulla teoria dei giochi e la progettazione di meccanismi.
- ◆ Nella Parte VI viene meglio collegato il materiale sull'apprendimento statistico, simbolico e neurale e sono state aggiunte sezioni sugli algoritmi boosting, l'algoritmo EM, l'apprendimento basato sulle istanze e i metodi kernel (macchine a vettore di supporto).
- ◆ Nella Parte VII la trattazione sull'elaborazione del linguaggio è arricchita con sezioni sulla gestione del discorso e l'induzione grammaticale, oltre che da un capitolo dedicato ai modelli probabilistici del linguaggio, con applicazioni nei campi dell'*information retrieval* e della traduzione automatica. La trattazione della robotica pone l'accento sull'integrazione di dati sensoriali incerti, e il capitolo sulla visione presenta materiale aggiornato sul riconoscimento di oggetti.
- ◆ Nella Parte VIII abbiamo aggiunto una sezione dedicata alle implicazioni etiche dell'IA.

Come usare questo libro

Grazie all'ampiezza della materia trattata, il libro può essere adottato sia in corsi brevi di introduzione per studenti di primo livello che per corsi di specializzazione dedicati agli argomenti più avanzati. I programmi dei corsi che hanno adottato la prima edizione del libro sono disponibili sul web all'indirizzo aima.cs.berkeley.edu, insieme a molti suggerimenti per aiutarvi a trovare la soluzione più appropriata alle vostre necessità.

Il libro include 385 esercizi: quelli che richiedono una certa quantità di programmazione sono marcati con l'icona di un mouse. Per risolverli si può ricorrere al nostro deposito di codice al solito indirizzo aima.cs.berkeley.edu; alcuni sono abbastanza complessi da poter essere considerati veri e propri progetti. Alcuni esercizi richiedono di svolgere una ricerca nella letteratura esistente e sono marcati con l'icona di un pila di libri. I docenti che adottano questo testo potranno richiedere il CD-ROM con il Solution Manual contattando hpeitalia@pearson.com.

In tutto il libro, i concetti fondamentali sono evidenziati da una lente d'ingrandimento. Abbiamo anche incluso un dettagliato indice di circa 10.000 voci. Ogni volta che un **termine nuovo** è definito per la prima volta, è riportato a margine, in modo da facilitarne il ritrovamento.

Come usare il sito web

Il libro è supportato da numerosi supplementi e risorse on-line. All'indirizzo aima.cs.berkeley.edu troverete:

- ◆ implementazioni dei numerosi algoritmi descritti nel libro in diversi linguaggi di programmazione
- ◆ l'elenco delle università che adottano questo testo, con molti link ai relativi supporti online
- ◆ materiale iconografico, slide in ppt, tex, ps e pdf, e altro materiale utile per i docenti (in lingua inglese)
- ◆ un elenco commentato di oltre 800 siti con materiale e informazioni utili nel campo dell'IA
- ◆ alcuni capitoli dell'edizione originale di esempio (tek, ps e pdf)
- ◆ istruzioni su come partecipare a un gruppo di discussione sul libro
- ◆ i link ai siti degli autori, ai quali inviare commenti o quesiti
- ◆ errata corrigé

Ringraziamenti

Jitendra Malik ha scritto la maggior parte del Capitolo 24, dedicato alla visione. Gran parte del Capitolo 25 (robotica) è stato scritto da Sebastian Thrun in quest'edizione e da John Canny nella prima edizione. Doug Edwards ha svolto la ricerca per le note storiche nella prima edizione. Tim Huang, Mark Paskin e Cynthia Bruyns hanno aiutato con la formattazione dei diagrammi e degli algoritmi. Alan Apt, Sondra Chavez, Toni Holm, Jake Warde, Irwin Zucker e Camille Trentacoste alla Prentice Hall hanno fatto del loro meglio per farci rispettare i tempi e hanno proposto molti utili suggerimenti sul progetto e il contenuto del libro.

Stuart ringrazia i suoi genitori per il continuo supporto e la moglie, Loy Sheflott, per l'infinita pazienza e la saggezza senza limiti. Spera che Gordon e Lucy leggeranno presto queste righe. Anche RUGS (Russell's Unusual Group of Students) si è dimostrato insolitamente utile.

Peter ringrazia i suoi genitori (Torsten e Gerda) per averlo avviato agli studi e sua moglie (Kris), i figli e gli amici per averlo incoraggiato e sopportato nelle lunghe ore di scrittura e quelle, ancora più lunghe, di riscrittura.

Siamo in debito nei confronti dei bibliotecari di Berkeley, Stanford, del MIT e della NASA, e degli sviluppatori di CiteSeer e Google, che hanno rivoluzionato il modo di fare ricerca.

Non possiamo ringraziare tutte le persone che hanno usato il libro e fornito suggerimenti, ma vorremmo riconoscere qui i commenti particolarmente utili di Eyal Amir, Krzysztof Apt, Ellery Aziel, Jeff Van Baalen, Brian Baker, Don Barker, Tony Barrett, James Newton Bass, Don Beal, Howard Beck, Wolfgang Bibel, John Binder, Larry Bookman, David R. Boxall, Gerhard Brewka, Selmer Bringsjord, Carla Brodley, Chris Brown, Wilhelm Burger, Lauren Burka, Joao Cachopo, Murray Campbell, Norman Carver, Emmanuel Castro, Anil Chakravarthy, Dan Chisarick, Roberto Cipolla, David Cohen, James Coleman, Julie Ann Comparini, Gary Cottrell, Ernest Davis, Rina Dechter, Tom Dietterich, Chuck Dyer, Barbara Engelhardt, Doug Edwards, Kutluhan Erol, Oren Etzioni, Hana Filip, Douglas Fisher, Jeffrey Forbes, Ken Ford, John Fosler, Alex Franz, Bob Futrelle, Marek Galecki, Stefan Gerberding, Stuart Gill, Sabine Glesner, Seth Golub, Gosta Graahne, Russ Greiner, Eric Grimson, Barbara Grosz, Larry Hall, Steve Hanks, Othar Hansson, Ernst Heinz, Jim Hendler, Christoph Herrmann, Vasant Honavar, Tim Huang, Seth Hutchinson, Joost Jacob, Magnus Johansson, Dan Jurafsky, Leslie Kaelbling, Keiji Kanazawa, Surekha Kasibhatla, Simon Kasif, Henry Kautz, Gernot Kerschbaumer, Richard Kirby, Kevin Knight, Sven Koenig, Daphne Koller, Rich Korf, James Kurien, John Lafferty, Gus Larsson, John Lazzaro, Jon LeBlanc, Jason Leatherman, Frank Lee, Edward Lim, Pierre Louveaux, Don Loveland, Sridhar Mahadevan, Jim Martin, Andy Mayer, David McGrane, Jay Mendelsohn, Brian Milch, Steve Minton, Vibhu Mittal, Leora Morgenstern, Stephen Muggleton, Kevin Murphy, Ron Musick, Sung Myaeng, Lee Naish, Pandu Nayak, Bernhard Nebel, Stuart Nelson, XuanLong Nguyen, Illah Nourbakhsh, Steve Omohundro, David Page, David Palmer, David Parkes, Ron Parr, Mark Paskin, Tony Passera, Michael Pazzani, Wim Pijs, Ira Pohl, Martha Pollack, David Poole, Bruce Porter, Malcolm Pradhan, Bill Pringle, Lorraine Prior, Greg Provan, William Rapaport, Philip Resnik, Francesca Rossi, Jonathan Schaeffer, Richard Scherl, Lars Schuster, Soheil Shams, Stuart Shapiro, Jude Shavlik, Satinder Singh, Daniel Sleator, David Smith, Bryan So, Robert Sproull, Lynn Stein, Larry Stephens, Andreas Stolcke, Paul Stradling, Devika Subramanian, Rich Sutton, Jonathan Tash, Austin Tate, Michael Thielscher, William Thompson, Sebastian Thrun, Eric Tiedemann, Mark Torrance, Randall Upham, Paul Utgoff, Peter van Beek, Hal Varian, Sunil Vemuri, Jim Waldo, Bonnie Webber, Dan Weld, Michael Wellman, Michael Dean White, Kamin Whitehouse, Brian Williams, David Wolfe, Bill Woods, Alden Wright, Richard Yen, Weixiong Zhang, Shlomo Zilberman e dei revisori anonimi alla Prentice Hall.

Circa il frontespizio

L'immagine nel frontespizio è stata progettata dagli autori e realizzata da Lisa Marie Sardegna e Maryann Simmons utilizzando SGI InventorTM e Adobe PhotoshopTM. L'immagine raffigura i seguenti elementi della storia dell'IA.

- ◆ L'algoritmo di pianificazione di Aristotele dal *De Motu Animalium* (c. 400 a. C.).
- ◆ Il generatore di concetti dalla *Ars Magna* di Raimondo Lullo (c. 1300 d. C.).
- ◆ Il motore differenziale di Charles Babbage, il primo prototipo di computer universale (1848).
- ◆ La notazione di Gottlob Frege per la logica del primo ordine (1789).
- ◆ I diagrammi per il ragionamento logico di Lewis Carroll (1886).
- ◆ La notazione delle reti probabilistiche di Sewall Wright (1921).
- ◆ Alan Turing (1912–1954).
- ◆ Il robot Shakey (1969–1973).
- ◆ Un moderno sistema esperto per la diagnostica (1993).

Gli autori

Stuart Russell è nato nel 1962 a Portsmouth, in Inghilterra. Si è laureato in fisica *cum laude* alla Oxford University nel 1982, e ha ottenuto il dottorato in informatica a Stanford nel 1986. In seguito è passato all'Università della California a Berkeley, ove è professore di informatica, direttore del Centro per i Sistemi Intelligenti e titolare della cattedra Smith-Zadeh in ingegneria. Nel 1990 ha ricevuto il Presidential Young Investigator Award della National Science Foundation, e nel 1995 ha vinto *ex aequo* il premio Computers and Thought. Nel 1996 è stato Miller Professor dell'Università della California e ha ricevuto una Chancellor's Professorship nel 2000. Nel 1998 ha tenuto le Forsythe Memorial Lectures presso la Stanford University. È membro ed ex componente del consiglio esecutivo della American Association for Artificial Intelligence. Ha pubblicato più di 100 articoli su una vasta gamma di argomenti di intelligenza artificiale. Tra i suoi altri libri vi sono *The Use of Knowledge in Analogy and Induction* e (con Eric Wefald) *Do the Right Thing: Studies in Limited Rationality*.

Peter Norvig è direttore della Search Quality alla Google, Inc. È membro e componente del consiglio esecutivo della American Association for Artificial Intelligence. In precedenza è stato a capo della Computational Sciences Division all'Ames Research Center della NASA, ove ha supervisionato la ricerca e lo sviluppo in intelligenza artificiale e robotica. Prima era stato "chief scientist" alla Junglee, dove ha aiutato a sviluppare uno dei primi servizi di estrazione di informazione da Internet, e "senior scientist" presso i laboratori della Sun Microsystems dove si è occupato di *information retrieval* intelligente. Ha conseguito la laurea in matematica applicata alla Brown University e il dottorato in informatica all'Università della California a Berkeley. È stato professore alla University of Southern California e membro del gruppo di ricerca a Berkeley. Ha pubblicato oltre 50 articoli di informatica, e i libri *Paradigms of AI Programming: Case Studies in Common Lisp*, *Verbmobil: A Translation System for Face-to-Face Dialog* e *Intelligent Help Systems for UNIX*.

Per Loy, Gordon e Lucy — S.J.R.

Per Kris, Isabella e Juliet — P.N.

Intelligenza artificiale

Parte Prima

Capitolo 1

Introduzione

Nella quale cerchiamo di spiegare perché consideriamo l'intelligenza artificiale un argomento degno di studio accurato, e cerchiamo anche di decidere che cos'è precisamente, essendo questa una buona cosa da fare prima di cominciare.

Gli esseri umani fanno riferimento a se stessi con il termine homo sapiens perché tengono che le proprie capacità mentali siano molto importanti. Per migliaia d'anni abbiamo cercato di comprendere come pensiamo; ovvero come un semplice mucchio di materia può percepire, capire, predire e manipolare un mondo molto più grande e complicato. Il campo dell'intelligenza artificiale, o IA, va ancora più in là: il suo obiettivo non è solo comprendere, ma anche costruire entità intelligenti.

L'IA non è una scienza nuovissima: le sue origini risalgono al periodo immediatamente successivo alla Seconda Guerra Mondiale, e il termine stesso fu coniato nel 1956. Insieme alla biologia molecolare, è regolarmente citata come "il campo in cui vorrei maggiormente lavorare" da parte di scienziati di altre discipline. Uno studente di fisica potrebbe ragionevolmente pensare che tutte le idee migliori siano già state formulate da Galileo, Newton, Einstein e gli altri: L'intelligenza artificiale, invece, ha ancora molti posti liberi per dei nuovi Einstein a tempo pieno.

Al giorno d'oggi l'IA è suddivisa in un grande numero di sottodiscipline: alcune aree, come l'apprendimento e la percezione, hanno un campo d'azione generale; altre invece si occupano di problemi specifici, come il gioco degli scacchi, la dimostrazione di teoremi matematici, la scrittura di poesie e la diagnosi di alcune malattie. L'IA si occupa di sistematizzare e rendere automatiche alcune attività intellettive, e di conseguenza si può potenzialmente applicare a ogni sfera del pensiero umano. In questo senso, è un campo davvero universale.

1.1 Cos'è l'intelligenza artificiale?

Abbiamo affermato che l'IA è una disciplina interessante, ma non abbiamo ancora detto cos'è. La Figura 1.1 mostra otto definizioni di intelligenza artificiale prese da altrettanti libri di testo e poste lungo due assi: approssimativamente, quelle in alto rivolgono l'attenzione ai *processi di pensiero* e al *ragionamento*, quelle in basso al *comportamento*. Le definizioni poste a sinistra misurano il successo in base alla somiglianza a un'esecuzione *umana*, mentre quelle a destra usano come metro di paragone il concetto ideale di intelligenza, che noi chiamiamo **razionalità**. Un sistema è razionale se, date le sue conoscenze, "fa la cosa giusta".

Sistemi che pensano come esseri umani "L'eccitante, nuovo tentativo di far sì che i computer arrivino a pensare... <i>macchine dotate di mente</i> , nel pieno senso della parola." (Haugeland, 1985) "[L'automazione delle] attività che associamo al pensiero umano, come il processo decisionale, la risoluzione di problemi, l'apprendimento..." (Bellman, 1978)	Sistemi che pensano razionalmente "Lo studio delle facoltà mentali attraverso l'uso di modelli computazionali." (Charniak e McDermott, 1985) "Lo studio dei processi di calcolo che rendono possibile percepire, ragionare e agire." (Winston, 1992)
Sistemi che agiscono come esseri umani "L'arte di creare macchine che eseguono attività che richiedono intelligenza quando vengono svolte da persone." (Kurzweil, 1990) "Lo studio di come far eseguire ai computer le attività in cui, al momento, le persone sono più brave." (Rich e Knight, 1991)	Sistemi che agiscono razionalmente "L'Intelligenza Computazionale è lo studio della progettazione di agenti intelligenti." (Poole et al., 1998) "L'IA... riguarda il comportamento intelligente negli artefatti." (Nilsson, 1998)

Figura 1.1 Alcune definizioni di intelligenza artificiale, organizzate in quattro categorie.

Nella storia dell'IA, questi quattro approcci sono stati tutti sviluppati. Come potete immaginare, c'è una certa tensione tra chi adotta un approccio centrato sugli esseri umani e chi ne preferisce uno rivolto alla razionalità.¹ Nel primo caso l'intelligenza artificiale si avvicina alle scienze empiriche, richiedendo la verifica sperimentale delle ipotesi; il metodo razionalista, invece, sfrutta una combinazione di matematica e ingegneria. Ogni gruppo di studiosi ha duramente contestato l'altro, ma lo ha anche aiutato. Ora presenteremo i quattro approcci nei particolari.

Agire umanamente: l'approccio del test di Turing

Il test di Turing, proposto da Alan Turing nel 1950, è stato concepito per fornire una soddisfacente definizione operativa dell'intelligenza. Invece di proporre una lunga e probabilmente controversa lista di caratteristiche richieste a un computer per essere considerato intelligente, Turing ha suggerito un test basato sull'impossibilità di distinguere da entità che lo sono indubbiamente: gli esseri umani. Il computer passerà il test se un esaminatore umano, dopo aver posto alcune domande in forma scritta, non sarà in grado di capire se le risposte provengono da una persona o no. Il Capitolo 26 discute i dettagli del test e considera se un computer in grado di passarlo può essere davvero ritenuto intelligente. Per adesso, ci limitiamo a notare che programmare una macchina in grado di superare il test richiede un sacco di lavoro. Il computer dovrebbe possedere le seguenti capacità:

test di Turing

interpretazione del linguaggio naturale

rappresentazione della conoscenza
ragionamento automatico

apprendimento

- ◆ **interpretazione del linguaggio naturale** per comunicare con l'esaminatore nel suo linguaggio umano;
- ◆ **rappresentazione della conoscenza** per memorizzare quello che sa o sente;
- ◆ **ragionamento automatico** per utilizzare la conoscenza memorizzata in modo da rispondere alle domande e trarre nuove conclusioni;
- ◆ **apprendimento** per adattarsi a nuove circostanze, individuare ed estrapolare pattern.

¹ Dobbiamo precisare che, distinguendo tra comportamento *umano* e comportamento *razionale*, non stiamo implicando che gli esseri umani siano "irrazionali" nel senso che hanno problemi emotivi o sono pazzi. Stiamo semplicemente notando che non siamo perfetti: non siamo tutti grandi maestri di scacchi, anche se magari conosciamo le regole; inoltre, sfortunatamente, non tutti gli studenti prenderanno 30 all'esame. Alcuni errori sistematici nel ragionamento umano sono catalogati da Kahneman et al. (1982).

test di Turing totale

visione artificiale

robotica



scienze cognitive

Il test di Turing evita deliberatamente l'interazione diretta tra l'esaminatore e il computer, dato che la simulazione *fisica* di una persona non è richiesta per la determinazione dell'intelligenza. Tuttavia, esiste anche un cosiddetto test di Turing totale che include un segnaletico video in modo che l'esaminatore possa verificare le capacità percettive del soggetto, e anche la possibilità di passare oggetti fisici attraverso una finestrella. Per passare il test di Turing totale, il computer necessiterà anche di:

- ◆ visione artificiale per percepire gli oggetti;
- ◆ robotica per manipolare gli oggetti e spostarsi fisicamente.

Le sei discipline elencate qui sopra abbracciano gran parte dell'intelligenza artificiale, e bisogna dare credito a Turing di aver concepito un test che è rimasto significativo a distanza di cinquant'anni. Tuttavia, i ricercatori non hanno dedicato molti sforzi al tentativo di costruire un sistema capace di passare il test di Turing, ritenendo più importante studiare i principî alla base dell'intelligenza che dedicarsi alla creazione di un duplice: dopotutto, la ricerca del "volo artificiale" ha raggiunto il successo quando i fratelli Wright e altri hanno smesso di imitare gli uccelli per studiare l'aerodinamica. I manuali di ingegneria aerospaziale non definiscono l'obiettivo della loro disciplina come la creazione di "macchine che volano esattamente come un piccione, in modo così perfetto da ingannare anche gli altri piccioni".

Pensare come esseri umani: l'approccio della modellazione cognitiva

Se vogliamo dire che un determinato programma ragiona come un essere umano, dobbiamo prima determinare come pensiamo, entrare *dentro* i meccanismi interni del cervello umano. Ci sono due modi per farlo: l'introspezione, ovvero il tentativo di catturare "al volo" i nostri pensieri mentre scorrono, oppure la sperimentazione psicologica. Una volta che abbiamo formulato una teoria della mente sufficientemente precisa, diventa possibile esprimere sotto forma di un programma per computer. Se il comportamento del software, per quanto riguarda il suo input/output e la temporizzazione, corrisponde a quello di una persona, potrebbe essere una prova che alcuni dei meccanismi del programma operano anche negli esseri umani. Ad esempio, Allen Newell e Herbert Simon, che hanno sviluppato il GPS o General Problem Solver (Newell e Simon, 1961), non si accontentarono di scrivere un programma che risolvesse correttamente i problemi: il loro vero interesse consisteva nel confronto della sequenza dei passi del suo ragionamento con un'analogia sequenza prodotta da soggetti umani. Il campo interdisciplinare delle scienze cognitive unisce modelli per computer sviluppati dall'intelligenza artificiale e tecniche di sperimentazione psicologica nel tentativo di costruire teorie precise e verificabili sul funzionamento della mente umana.

1.1 Cos'è l'intelligenza artificiale?

Il campo delle scienze cognitive è davvero affascinante, e merita da solo un'intera encyclopedia (Wilson e Keil, 1999). In questo libro non cercheremo di descrivere lo stato delle conoscenze sulla cognizione umana, anche se occasionalmente ci potrà capitare di commentare i punti in comune e le differenze con l'IA. La vera scienza cognitiva, comunque, è necessariamente basata sull'investigazione sperimentale di vere persone e animali, mentre noi partiremo sempre dall'ipotesi che il lettore compia i suoi esperimenti su computer.

Nei primi tempi dell'IA si faceva spesso confusione tra approcci diversi: un autore avrebbe potuto argomentare che un algoritmo eseguiva efficacemente un'attività, e quindi era un buon modello dell'esecuzione umana, o viceversa. Gli autori moderni separano chiaramente le due cose, e questa distinzione ha aiutato lo sviluppo sia dell'IA che delle scienze cognitive. I due campi continuano a influenzarsi positivamente, specialmente nell'area della visione e del linguaggio naturale. La visione, in particolare, ha recentemente fatto molti passi avanti grazie a un approccio integrato che prende in considerazione da un lato la sperimentazione neurofisiologica e dall'altro i modelli matematici.

Pensare razionalmente: l'approccio delle "leggi del pensiero"

Il filosofo greco Aristotele è stato uno dei primi a cercare di codificare formalmente il "pensiero corretto", ovvero i processi di ragionamento irrefutabili. I suoi sillogismi forniscono pattern di deduzione che portano sempre a conclusioni corrette quando sono corrette le premesse: ad esempio, il famoso "Socrate è un uomo; tutti gli uomini sono mortali; quindi Socrate è mortale". Si riteneva che queste leggi del pensiero governassero il funzionamento della mente; il loro studio ha dato origine alla disciplina chiamata logica.

I logici del XIX secolo hanno sviluppato una notazione precisa per formulare proposizioni riguardanti tutti gli aspetti del mondo e le relazioni tra essi: tale notazione è molto più potente di quella aritmetica, che permette sostanzialmente di indicare se due numeri sono uguali o no. Già nel 1965 esistevano programmi che potevano, in linea di principio, risolvere qualsiasi problema descritto in linguaggio logico e dotato di soluzione.² La tradizione logicista, come viene chiamata all'interno dell'intelligenza artificiale, spera di partire da programmi siffatti per costruire sistemi intelligenti.



sillogismi

logica

tradizione logicista

² Se non esiste soluzione, il programma potrebbe continuare a cercarne una e non terminare mai l'esecuzione.

Quest'approccio ha due punti deboli. Prima di tutto, non è affatto facile esprimere una conoscenza non formalizzata nei termini strettamente formali richiesti dalla notazione logica, in particolare quando tale conoscenza non è sicura al 100%. In secondo luogo, c'è una grande differenza tra essere in grado di risolvere un problema "in linea di principio" e farlo nella pratica. Anche problemi con solo qualche dozzina di fatti possono esaurire le risorse computazionali disponibili, a meno che non venga fornita al calcolatore una guida sui passi di ragionamento da applicare per primi. Benché questi ostacoli si possano applicare a qualsiasi tentativo di costruire sistemi in grado di ragionare, sono apparsi inizialmente nella tradizione logicista.

agente



Agire razionalmente: l'approccio degli agenti razionali

Un agente è semplicemente qualcosa che agisce, che fa qualcosa. Tuttavia si suppone che gli agenti computerizzati abbiano caratteristiche particolari, che li distinguono dai semplici "programmi": ad esempio potrebbero operare con controllo autonomo, essere in grado di percepire l'ambiente, persistere in un'attività per un lungo arco di tempo, adattarsi al cambiamento ed essere capaci di scambiarsi a vicenda gli obiettivi. Un agente razionale agisce in modo da ottenere il miglior risultato o, in condizione di incertezza, il miglior risultato atteso.

Nell'approccio all'IA basato sulle "leggi del pensiero", l'enfasi è posta sulla correttezza delle inferenze. Essere in grado di formulare deduzioni corrette è talvolta parte di un agente razionale, perché un modo di agire razionalmente è ragionare in termini logici, arrivare alla conclusione che una data azione porterà al soddisfacimento dei propri obiettivi, e agire quindi in tal senso. D'altra parte, l'inferenza corretta non rappresenta tutta la razionalità, perché in molte situazioni non si può dimostrare che c'è una particolare azione "giusta" da fare, e tuttavia qualcosa va fatto. Ci sono anche tipologie di comportamento razionale che non coinvolgono l'inferenza logica: ad esempio, ritirare la mano da una stufa rovente è un'azione di riflesso che porta solitamente a vantaggi maggiori di un'azione più lenta, compiuta dopo un attento ragionamento.

Tutte le abilità richieste dal test di Turing hanno lo scopo di agire razionalmente. Dobbiamo essere in grado di rappresentare la conoscenza e applicarvi un ragionamento, perché questo ci dà la possibilità di prendere le giuste decisioni in una vasta gamma di situazioni. Dobbiamo essere capaci di generare frasi comprensibili nel linguaggio naturale, perché questo ci aiuta a comunicare in una società complessa. L'apprendimento ci serve non solo per ostentare erudizione, ma perché avere una buona idea del funzionamento del mondo ci permette di generare strategie più efficaci per interagire con esso. Abbiamo bisogno della percezione visiva non solo perché guardare è divertente, ma per avere una buona idea delle possibili conseguenze di un'azione: ad esempio, dopo aver adocchiato un bocconcino prelibato possiamo avvicinarlo per farlo nostro.

Per tutte queste ragioni, lo studio dell'IA come progettazione di agenti razionali presenta almeno due vantaggi. Prima di tutto, è più generale dell'approccio basato sulle "leggi del pensiero", poiché il corretto uso dell'inferenza è solo uno di molteplici meccanismi utilizzabili per arrivare alla razionalità. In secondo luogo, si presta meglio a recepire gli sviluppi scientifici rispetto agli approcci basati sul comportamento o sul pensiero umano, dato che la razionalità è definita chiaramente e si può applicare in modo generale. Il comportamento umano, d'altra parte, è adattato a un ambiente specifico ed è il prodotto di un processo di evoluzione complicato e in gran parte sconosciuto, che è ancora ben lontano dal raggiungere la perfezione. *Questo libro, di conseguenza, si concentrerà sui principî generali degli agenti razionali e sui componenti adatti alla loro costruzione.* Come vedremo, nonostante l'apparente semplicità con cui si può formulare il problema, cercando di risolverlo verrà alla luce una gran quantità di altre questioni. Il Capitolo 2 ne esporrà alcune in dettaglio.

C'è un punto importante da tenere a mente: ben presto vedremo che raggiungere la razionalità perfetta – fare sempre la cosa giusta – non è fattibile all'interno di sistemi complicati, perché i requisiti computazionali sono semplicemente troppo alti. Nella maggior parte del libro, comunque, partiremo dall'ipotesi che la razionalità perfetta sia un buon punto d'inizio per l'analisi. Questa soluzione semplifica il problema e fornisce uno scenario adatto all'esposizione dei concetti che sono alla base della nostra disciplina. Il Capitolo 6 discute esplicitamente l'argomento della razionalità limitata, che tratta il caso in cui si deve agire in modo appropriato quando manca il tempo di svolgere tutti i calcoli necessari.



razionalità limitata

1.2 I fondamenti dell'Intelligenza Artificiale

In questo paragrafo forniremo una breve panoramica delle discipline che hanno contribuito all'IA con idee, punti di vista e tecniche. Come tutte le storie anche questa si deve necessariamente concentrare su un piccolo numero di persone, eventi e idee, trascurandone altre ugualmente importanti. La nostra esposizione è organizzata intorno a un piccolo insieme di quesiti: non vogliamo certo intendere che queste siano le uniche domande poste da tali discipline, o che esse abbiano avuto come scopo ultimo quello di contribuire all'Intelligenza Artificiale.

Filosofia (428 a. C. – presente)

- ◆ È possibile applicare regole formali per trarre conclusioni valide?
- ◆ In che modo la mente scaturisce dal cervello fisico?
- ◆ Da dove proviene la conoscenza?
- ◆ Come fa la conoscenza a trasformarsi in azione?

Aristotele (384-322 a.C.) fu il primo filosofo a formulare un insieme preciso di leggi che governano la parte razionale della mente. Egli sviluppò un sistema informale di sillogismi per il ragionamento corretto, che in via di principio consentivano a chiunque, date le premesse iniziali, di generare meccanicamente le conclusioni. Molto tempo dopo, Raimondo Lullo (morto nel 1315) ebbe l'idea di utilizzare un artefatto meccanico per eseguire dei ragionamenti utili. Le sue "ruote della logica" sono raffigurate nel frontespizio di questo libro. Thomas Hobbes (1588–1679) ipotizzò che il ragionamento avesse una natura affine al calcolo numerico, ovvero che noi "eseguiamo addizioni e sottrazioni nei nostri pensieri silenziosi". L'automazione della computazione vera e propria era già in corso; intorno al 1500 Leonardo da Vinci (1452–1519) progettò, ma non costruì, un calcolatore meccanico; alcune ricostruzioni recenti hanno dimostrato che il progetto era corretto. La prima macchina calcolatrice fu costruita attorno al 1623 dallo scienziato tedesco Wilhelm Schickard (1592–1635), benché sia più nota la Pascalina, costruita nel 1642 da Blaise Pascal (1623–1662). Pascal scrisse che "la macchina aritmetica produce effetti che sembrano più vicini al pensiero di tutte le azioni degli animali". Gottfried Wilhelm Leibniz (1646–1716) costruì un dispositivo meccanico concepito per eseguire operazioni su concetti invece che numeri, ma le sue capacità erano piuttosto limitate.

Una volta acquisita l'idea che un insieme di regole possa descrivere il funzionamento della parte più formale e razionale della mente, il passo successivo è considerarla come un sistema puramente fisico. Cartesio (1596–1650) fornì la prima discussione chiara sulla distinzione tra mente e materia e dei problemi conseguenti. Un problema connesso alla concezione puramente fisica della mente è che lascia ben poco spazio al libero arbitrio: se il cervello è governato interamente da leggi fisiche, allora non ha più volontà di una roccia che "decide" di cadere verso il centro della terra. Sebbene fosse un grande difensore del potere del ragionamento, Cartesio propugnava anche il dualismo, sostenendo che ci fosse una parte della mente umana (anima o spirito) esente dalle leggi fisiche, al di fuori della natura. Gli animali, al contrario, non possedevano questa natura duale, e si sarebbero potuti trattare come macchine. Un'alternativa al dualismo è il materialismo, che sostiene che il funzionamento del cervello secondo le leggi della fisica costituisce la mente. Il libero arbitrio è semplicemente il modo in cui la percezione delle scelte disponibili si manifesta al processo decisionale.

Data una mente fisica che manipola la conoscenza, il problema successivo è stabilire la fonte di tale conoscenza. Il movimento dell'empirismo, che prende piede dal *Novum Organum* di Francesco Bacone (1561–1626),³ è caratterizzato dal detto di John Locke (1632–1704): “Non c’è nulla nell’intelletto, che non fosse già nei sensi”. Il *Trattato sulla natura umana* (1739) di David Hume (1711–1776) propose quello che oggi è noto come principio di induzione: le regole generali sono dedotte da ripetute associazioni tra i loro elementi. Sviluppando il lavoro di Ludwig Wittgenstein (1889–1951) e Bertrand Russell (1872–1970), il famoso Circolo di Vienna, guidato da Rudolf Carnap (1891–1970), sviluppò la dottrina del positivismo logico. Questa teoria sostiene che tutta la conoscenza può essere espressa da teorie collegate, alla fine, a enunciati osservativi che corrispondono alle percezioni sensoriali.⁴ La teoria della conferma di Carnap e Carl Hempel (1905–1997) cercò di spiegare come si può acquisire conoscenza dall’esperienza. Il libro di Carnap *La struttura logica del mondo* (1928) definì un’esplicita procedura computazionale per estrarre conoscenza a partire da esperienze elementari. Si trattò probabilmente della prima teoria della mente come processo computazionale.

L’ultimo elemento della concezione filosofica della mente è il collegamento tra conoscenza e azione. Quest’aspetto è fondamentale per l’IA, perché l’intelligenza richiede tanto il ragionamento quanto l’azione. Inoltre, solo comprendendo le ragioni dietro le azioni possiamo capire come costruire un agente le cui azioni siano giustificabili (o razionali). Aristotele sostenne che le azioni sono giustificate da un collegamento logico tra gli scopi dell’agente e la conoscenza del risultato di ogni azione (l’ultima parte di questo frammento compare anche nel frontespizio di questo libro):

Ma perché succede che certe volte il pensiero sia accompagnato dall’azione e altre volte no, certe volte dal moto e altre volte no? Sembra che accada quasi la stessa cosa che si verifica nel caso del ragionamento e della creazione di inferenze riguardo oggetti immutabili. Ma in quel caso il risultato è una proposizione speculativa . . . laddove qui la conclusione che scaturisce da due premesse è un’azione. . . . Ho bisogno di coprirmi; un mantello offre copertura. Ho bisogno di un mantello. Quello di cui ho bisogno, devo farlo; ho bisogno di un mantello. Devo farmi un mantello. E la conclusione, “devo farmi un mantello”, è un’azione. (Nussbaum, 1978, p. 40)

empirismo

induzione

positivismo logico
enunciati osservativi
teoria della conferma

Carnap Vienna

³ Un aggiornamento dell’*Organon* di Aristotele, inteso come strumento del pensiero.

⁴ Secondo questo sistema, tutte le proposizioni dotate di significato possono essere verificate o falsificate analizzando il significato delle parole o eseguendo esperimenti. Dato che questo esclude intenzionalmente la maggior parte della metafisica, il positivismo logico fu molto impopolare in alcuni circoli.

Nell'*Etica Nicomachea* (Libro III, 3, 1112b), Aristotele sviluppa ulteriormente quest'argomento, suggerendo un algoritmo:

Le nostre decisioni non riguardano i fini, ma i mezzi. Infatti un medico non decide se deve curare, né un oratore se deve convincere, . . . Essi danno per scontato il fine e considerano come e con quali mezzi ottenerlo; e se sembra possibile raggiungerlo in più modi considerano quale sia il più facile ed efficace, mentre se il mezzo è uno solo allora considerano *come raggiungere il fine mediante quello e quali mezzi usare per raggiungere quello*, e così via finché non raggiungono la causa prima, . . . e ciò che viene ultimo in ordine di analisi sembra essere la prima cosa a essere messa in pratica. E se incontriamo un passaggio irrisolvibile dobbiamo abbandonare la ricerca, ad esempio se abbiamo bisogno di denaro e non possiamo procurarcelo; ma se una cosa sembra possibile allora cerchiamo di farla.

L'algoritmo di Aristotele è stato implementato 2300 anni dopo da Newell e Simon nel loro programma GPS. Ora lo definiremmo un sistema di pianificazione di regressione (v. Capitolo 11).

L'analisi basata sugli obiettivi è utile, ma non dice cosa fare quando per raggiungere uno scopo sono disponibili più azioni, o quando nessuna lo soddisferebbe pienamente. Antoine Arnauld (1612–1694) ha descritto correttamente una formula quantitativa per decidere quale azione intraprendere *in casi come questi*. Il libro *Utilitarismo* (1863) di John Stuart Mill (1806–1873) propugnava l'adozione di criteri razionali di decisione in tutte le sfere dell'attività umana. La più formale teoria delle decisioni è discussa nel paragrafo seguente.

Matematica (c. 800–presente)

- ◆ Quali sono le regole formali utilizzabili per trarre conclusioni valide?
- ◆ Cosa può essere calcolato?
- ◆ Come si deve ragionare quando l'informazione è incerta?

I filosofi hanno studiato la maggior parte dei concetti riguardanti l'IA, ma il passaggio a una scienza formale richiedeva un livello superiore di formalizzazione matematica in tre aree fondamentali: logica, computazione e probabilità.

L'idea di una logica formale si può far risalire ai filosofi dell'antica Grecia (v. Capitolo 7), ma il suo sviluppo matematico cominciò effettivamente con il lavoro di *George Boole* (1815–1864), che formulò i principî della logica proposizionale o booleana (Boole, 1847). Nel 1879, *Gottlob Frege* (1848–1925) estese la logica di Boole in modo da includere oggetti e relazioni, creando così la logica del primo ordine usata oggi come il più semplice sistema di rappresentazione della conoscenza.

za.⁵ Alfred Tarski (1902–1983) introdusse una teoria del riferimento che mostra come si possano collegare gli oggetti di una logica a quelli del mondo reale. Il passo successivo era determinare i limiti di ciò che si poteva ottenere con la logica e la computazione.

Si ritiene che il primo algoritmo non banale sia quello di Euclide per calcolare il massimo comune denominatore. Lo studio degli algoritmi risale ad al-Khuwarizmi, un matematico persiano del nono secolo, i cui lavori introdussero in Europa anche l'uso dei numeri arabi e dell'algebra. Boole e altri considerarono lo sviluppo di algoritmi per la deduzione logica, e verso la fine del XIX secolo molti scienziati dedicavano la loro ricerca alla formalizzazione del ragionamento matematico generale sotto forma di deduzione logica. Nel 1900, David Hilbert (1862–1943) presentò una lista di 23 problemi che (correttamente) riteneva avrebbero occupato i matematici per gran parte del secolo che iniziava. L'ultimo di tali problemi concerne la possibilità di scrivere un algoritmo per decidere il grado di verità di qualsiasi proposizione logica riguardante i numeri naturali: il famoso *Entscheidungsproblem*, o problema della decisione. Essenzialmente, Hilbert stava chiedendo se vi fossero limiti fondamentali alla potenza delle procedure di dimostrazione. Nel 1930, Kurt Gödel (1906–1978) mostrò che esiste effettivamente una procedura per dimostrare ogni proposizione vera nella logica del primo ordine di Frege e Russell, ma che tale logica non poteva esprimere il principio di induzione matematica necessario per definire i numeri naturali. Nel 1931 dimostrò che esistono dei limiti reali: il suo teorema di incompletezza stabilisce che all'interno di qualsiasi linguaggio abbastanza potente da descrivere le proprietà dei numeri naturali esistono proposizioni vere che sono indecidibili, nel senso che il loro valore di verità non può essere stabilito da nessun algoritmo.

Questo risultato fondamentale si può interpretare anche come la dimostrazione che esistono funzioni sui numeri interi che non possono essere rappresentate per via algoritmica, e sono quindi incomputabili. Questo spinse Alan Turing (1912–1954) a cercare di definire esattamente quali funzioni sono computabili. Questa nozione in effetti presenta qualche problema, perché non è possibile scrivere una definizione formale di computazione. Tuttavia in generale si accetta come definizione sufficiente la tesi di Church–Turing, che afferma che la macchina di Turing (1936) è in grado di calcolare ogni funzione computabile. Turing ha anche di-

algoritmo

teorema di incompletezza

⁵ La notazione proposta da Frege per la logica del primo ordine non è mai diventata popolare, per ragioni evidenti a chiunque guardi il frontespizio di questo libro.

mostrato che esistono funzioni che nessuna macchina di Turing può calcolare. Ad esempio, nessuna macchina può dire se *in generale* un dato programma, ricevuto un determinato input, restituirà un risultato o continuerà l'esecuzione per sempre.

Benché concetti come indecidibilità o incomputabilità siano molto importanti per comprendere i principî del calcolo automatico, quello di intrattabilità ha avuto un impatto molto superiore. Semplificando, si può dire che un problema è intrattabile quando il tempo richiesto per risolvere una sua determinata istanza cresce esponenzialmente con la dimensione dell'istanza stessa. La distinzione tra crescita polinomiale ed esponenziale nella complessità degli algoritmi fu enfatizzata per la prima volta a metà degli anni '60 (Cobham, 1964; Edmonds, 1965). Si tratta di un concetto fondamentale, perché una crescita esponenziale significa che istanze del problema anche moderatamente complesse non possono essere risolte in tempi ragionevoli. Di conseguenza, diventa necessario cercare di suddividere il problema generale della creazione di comportamento intelligente in sottoproblemi trattabili.

Come si fa a riconoscere un problema intrattabile? Un metodo è fornito dalla teoria della NP-complettezza, formulata inizialmente da Steven Cook (1971) e Richard Karp (1972). Cook e Karp hanno dimostrato che grandi classi di problemi canonici nei campi della ricerca combinatoria e del ragionamento sono NP-completi. Ogni classe di problemi a cui si può ridurre quella dei problemi NP-completi è probabilmente intrattabile: in effetti non è stato dimostrato che i problemi NP-completi siano necessariamente intrattabili, ma la maggior parte degli esperti ritiene che sia così. Questi risultati contrastano con l'ottimismo con cui la stampa non specializzata aveva salutato i primi computer – “super cervelloni elettronici” che erano “più veloci di Einstein”! Nonostante la velocità sempre maggiore dei calcolatori, i sistemi intelligenti dovranno sempre fare un uso molto accurato delle risorse computazionali disponibili. Per dirla in parole povere, il mondo è un'istanza di problema davvero grande! In tempi recenti, l'IA ha aiutato a spiegare perché alcune istanze di problemi NP-completi sono difficili da trattare mentre altre sono abbastanza semplici (Cheeseman et al., 1991).

Oltre alla logica e alla teoria della computazione, il terzo grande contributo dei matematici all'IA è la teoria della probabilità. Gerolamo Cardano (1501–1576) fu il primo a formulare il concetto di probabilità, descrivendolo nei termini dei possibili risultati degli eventi in un gioco d'azzardo. La probabilità divenne rapidamente una parte importante di tutte le scienze quantitative, rendendo più facile gestire misurazioni incerte e teorie incomplete. Pierre Fermat (1601–1665), Blaise Pascal (1623–1662), James Bernoulli (1654–1705), Pierre Laplace (1749–1827) e altri svilupparono la teoria e introdussero nuovi metodi statistici. Thomas Bayes (1702–1761) propose una regola capace di aggiornare i valori di probabilità in base ai dati via via raccolti. La sua regola e il campo di ricerca risultante, l'analisi bayesiana, sono alla base della maggior parte degli approcci moderni al ragionamento in condizioni di incertezza.

Economia (1776 – presente)

- ♦ Come dobbiamo prendere decisioni in modo da massimizzare il proprio vantaggio?
- ♦ Come dobbiamo farlo quando gli altri non sono d'accordo con noi?
- ♦ Come dobbiamo farlo quando i vantaggi potrebbero essere molto lontani nel futuro?

La scienza dell'economia ebbe origine nel 1776, quando il filosofo scozzese Adam Smith (1723–1790) pubblicò le *Ricerche sopra la natura e le cause della ricchezza delle nazioni*. Sebbene gli antichi greci e altri avessero già fornito contributi al pensiero economico, Smith fu il primo a trattare la disciplina come una vera scienza, applicando l'idea che i sistemi economici potessero essere considerati come un insieme di agenti tesi ognuno a massimizzare il proprio benessere economico. La maggior parte delle persone ritiene che l'economia abbia a che fare col denaro, ma gli esperti sanno che il loro campo di studi riguarda la modalità con cui le persone prendono decisioni che dovrebbero condurre a risultati a loro graditi. Il trattamento matematico degli "effetti graditi", o utilità, fu formalizzato inizialmente da Léon Walras (1834–1910), quindi sviluppato da Frank Ramsey (1931) e ancor più tardi arricchito da John von Neumann e Oskar Morgenstern nel loro libro *Teoria dei giochi ed economia* (1944).

La teoria delle decisioni, che fonde la teoria della probabilità con quella dell'utilità, fornisce un'infrastruttura formale completa in supporto alle decisioni (economiche o no) prese in condizioni di incertezza, ovvero nei casi in cui l'ambiente decisionale può essere descritto in modo probabilistico. Ciò si adatta bene a economie "grandi" in cui ogni agente non ha bisogno di prestare attenzione alle azioni degli altri agenti o individui. Nelle economie "piccole", la situazione assomiglia di più a un gioco: le azioni di un giocatore possono influenzare in modo significativo l'utilità di un altro, sia positivamente che negativamente. Lo sviluppo della teoria dei giochi da parte di von Neumann e Morgenstern (v. anche Luce e Raiffa, 1957) portò tra l'altro al risultato sorprendente secondo cui, in alcuni giochi, un agente razionale deve comportarsi in modo casuale, o almeno in un modo che sembra casuale agli avversari.

Nella maggior parte dei casi gli economisti non affrontarono la terza delle questioni elencate qui sopra, ovvero come prendere decisioni razionali quando gli effetti benefici delle azioni non sono immediati ma risultano invece dall'esecuzione di molteplici azioni in sequenza. Quest'argomento fu affrontato nel campo della ricerca operativa, che scaturì durante la Seconda Guerra Mondiale dagli sforzi britannici di ottimizzare le installazioni radar, e più tardi trovò applicazioni civili nel supporto a complesse decisioni di management. Gli studi di Richard Bellman (1957) portarono alla formalizzazione di una classe di problemi basati su sequenze di decisioni, chiamati processi decisionali di Markov.

teoria delle decisioni

teoria dei giochi

ricerca operativa

complessità

soddisfazione

neuroscienze

neuroni

Il lavoro svolto nei campi dell'economia e della ricerca operativa ha dato un grande contributo alla nostra nozione di agente intelligente; tuttavia per molti anni l'IA ha svolto la propria ricerca lungo un cammino completamente separato. Una delle ragioni era l'apparente complessità intrinseca nell'attività di prendere decisioni razionali. Herbert Simon (1916–2001), uno dei pionieri dell'IA, vinse il premio Nobel per l'economia nel 1978 per il suo lavoro che mostrava che modelli basati sulla soddisfazione – in grado di prendere decisioni “abbastanza buone”, invece di calcolare faticosamente la decisione ottima – fornivano una descrizione più accurata dell'effettivo comportamento umano (Simon, 1947). Negli anni '90 c'è stato un ritorno di interesse nelle tecniche di teoria delle decisioni applicata ai sistemi ad agenti (Wellman, 1995).

Neuroscienze (1861–presente)

- ♦ Come avviene l'elaborazione dell'informazione da parte del cervello?

Le neuroscienze si occupano dello studio del sistema nervoso, e in particolare del cervello. Il modo preciso in cui ha origine il pensiero rimane uno dei grandi misteri della scienza. Per migliaia d'anni si è postulato che il cervello fosse coinvolto nella formazione del pensiero, in base al fatto che i colpi alla testa possono portare a defezioni mentali. Anche il fatto che il cervello umano sia in qualche modo differente era risaputo; intorno al 335 a.C. Aristotele scrisse, “Di tutti gli animali, l'uomo ha il cervello più grande in proporzione alla sua massa”.⁶ Tuttavia, il cervello non fu riconosciuto comunemente come sede della coscienza fino alla metà del XVIII secolo: prima di allora, le ipotesi includevano tra l'altro il cuore, la milza e la ghiandola pineale.

Nel 1861, gli studi di Paul Broca (1824–1880) sulle afasie (deficit linguistici) nei pazienti con danni cerebrali diedero nuovo vigore al campo e convinsero gli studiosi di medicina dell'esistenza di aree specifiche del cervello responsabili di precise funzioni cognitive. In particolare, Broca mostrò che la produzione del linguaggio ha luogo in una porzione di cervello localizzata nell'emisfero sinistro, ora chiamata appunto area di Broca.⁷ A quel punto si sapeva già che il cervello è formato da cellule nervose o neuroni, ma si dovette aspettare il 1873 affinché Camil-

⁶ Da allora è stato appurato che alcune specie di delfini e balene hanno cervelli proporzionalmente più grandi. Oggi si pensa che la grande dimensione del cervello umano sia anche una conseguenza di un recente sviluppo del suo sistema di raffreddamento.

⁷ Molti citano Alexander Hood (1824) come possibile fonte precedente.

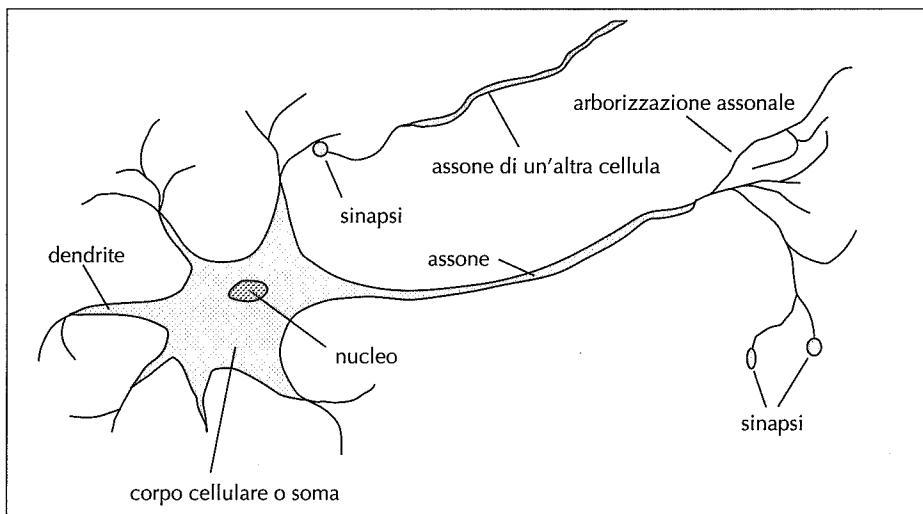


Figura 1.2 Le parti che compongono una cellula nervosa o neurone. Ogni neurone consiste in un corpo cellulare, o soma, che contiene il nucleo. Dal corpo si dirama una quantità di fibre chiamate dendriti e una singola, lunga fibra che prende il nome di assone. L'assone si prolunga per una grande distanza, maggiore di quella raffigurata nel diagramma: tipicamente gli assoni sono lunghi 1 cm. (100 volte il diametro del corpo cellulare), ma possono arrivare fino a un metro. Un neurone si collega con un numero che va da 10 a 100.000 altri neuroni, utilizzando punti di congiunzione chiamati sinapsi. I segnali si propagano da un neurone all'altro grazie a una complicata reazione elettrochimica e controllano l'attività cerebrale nel breve periodo, ma permettono anche dei cambiamenti a lungo termine nella posizione e nelle connessioni tra i neuroni. Si ritiene che questo meccanismo forni la base dell'apprendimento. La maggior parte dell'elaborazione delle informazioni ha luogo nella corteccia, il rivestimento esterno del cervello. L'unità base di organizzazione sembra essere una colonna di tessuto di circa mezzo millimetro di diametro, estesa per tutta la profondità della corteccia, che negli esseri umani è di circa 4 mm. Una colonna contiene circa 20.000 neuroni.

Io Golgi (1843–1926) sviluppasse una tecnica che permettesse la visualizzazione di neuroni singoli (v. Figura 1.2). Questa tecnica fu usata da Santiago Ramon y Cajal (1852–1934) nei suoi pionieristici studi sulle strutture neuronali del cervello.⁸

⁸ Golgi continuò a ritenere che il funzionamento del cervello avesse luogo principalmente all'interno di un mezzo continuo in cui erano inseriti i neuroni, mentre Cajal propendeva per la "dottrina neuronale". I due si divisero il premio Nobel nel 1906, ma i rispettivi discorsi di accettazione furono alquanto critici l'uno dell'altro.

Oggi possediamo alcuni dati relativi alla corrispondenza tra le aree del cervello e le parti del corpo da esse controllate, o da cui ricevono input sensoriale. Tali corrispondenze possono cambiare radicalmente nel corso di poche settimane, e alcuni animali sembrano avere corrispondenze multiple. Inoltre, non abbiamo ancora compreso appieno come opera il meccanismo in base al quale altre aree possono prendere il controllo di alcune funzioni quando si verificano dei danneggiamenti al tessuto nervoso. Praticamente non esiste alcuna teoria sulla memorizzazione dei singoli ricordi.

Le misurazioni dell'attività di un cervello integro cominciarono nel 1929 con l'invenzione dell'elettroencefalografo da parte di Hans Berger (EEG). I recenti sviluppi della risonanza magnetica funzionale (fMRI) (Ogawa et al., 1990) stanno fornendo ai neuroscienziati immagini dell'attività cerebrale con un dettaglio precedentemente impensabile, permettendo misurazioni che corrispondono in modo considerevole ai processi cognitivi in corso. Questi nuovi dati sono poi arricchiti dalla possibilità di registrare l'attività di singole cellule o neuroni. Nonostante questi grandi passi avanti, siamo ancora ben lontani dal comprendere come funziona veramente un qualsiasi processo cognitivo.

La conclusione veramente stupefacente è che *una collezione di semplici cellule può condurre al pensiero, all'azione e alla consapevolezza*, o, in altre parole, che *il cervello è la causa della mente* (Searle, 1992). L'unica vera teoria alternativa è il misticismo: la convinzione che ci sia un qualche reame mistico, al di là della scienza fisica, in cui operano le menti.

I cervelli e i computer digitali svolgono attività molto diverse e hanno caratteristiche differenti. La Figura 1.3 mostra che ci sono circa 1000 volte più neuroni nel tipico cervello umano che porte logiche nella CPU di un tipico computer di al-

	computer	cervello umano
unità computazionali	1 cpu, 10^8 porte logiche	10^{11} neuroni
unità di memorizzazione	10^{10} bit di ram	10^{11} neuroni
	10^{11} bit su disco rigido	10^{14} sinapsi
tempo di elaborazione per un ciclo	10^{-9} secondi	10^{-3} secondi
banda passante	10^{10} bit/secondo	10^{14} bit/secondo
aggiornamenti di memoria/sec	10^9	10^{14}

Figura 1.3 Un confronto approssimativo delle risorse computazionali di cui può disporre un computer (nel 2003) e il cervello umano. Dalla prima edizione del libro i numeri del computer sono aumentati almeno di un fattore 10, e presumibilmente faranno lo stesso nel prossimo decennio. I numeri che si riferiscono al cervello non sono mutati negli ultimi 10.000 anni.

te prestazioni. La Legge di Moore⁹ predice che il numero di porte logiche di un processore egualerà quelle dei neuroni in un cervello intorno al 2020. Naturalmente, da questa predizione si può dedurre ben poco; inoltre la differenza nella capacità di memorizzazione significa poco in confronto a quella nella velocità di commutazione o nel grado di parallelismo. I chip di computer possono eseguire un'istruzione in un nanosecondo, mentre i neuroni sono milioni di volte più lenti. I cervelli tuttavia sono capaci di rifarsi con gli interessi, perché tutti i neuroni e le sinapsi sono attive simultaneamente, mentre anche i computer più evoluti possono contare su un solo processore, o comunque un numero ridotto. Di conseguenza, *anche se un computer ha una velocità di commutazione un milione di volte superiore, il cervello risulta 100.000 volte più veloce nell'esecuzione delle attività.*



Psicologia (1879 – presente)

- ◆ Come pensano e agiscono gli esseri umani e gli animali?

Le origini della psicologia come scienza vengono solitamente fatte risalire al lavoro del medico tedesco Hermann von Helmholtz (1821–1894) e del suo studente Wilhelm Wundt (1832–1920). Helmholtz applicò il metodo scientifico allo studio della visione umana, e il suo *Manuale di ottica fisiologica* viene ancor oggi definito “il singolo trattato più importante sulla fisica e la fisiologia della visione umana” (Nalwa, 1993, p.15). Nel 1879, Wundt aprì il primo laboratorio di psicologia sperimentale presso l’Università di Lipsia ed effettuò esperimenti attentamente controllati, nei quali i suoi collaboratori dovevano eseguire qualche attività percettiva o associativa mentre si concentravano introspettivamente sui loro processi mentali. I controlli accurati contribuirono molto alla trasformazione della psicologia in una scienza, ma la natura soggettiva dei dati rendeva improbabile la falsificazione delle teorie da parte delle persone coinvolte negli esperimenti. I biologi che studiavano il comportamento animale, al contrario, non potevano basarsi su dati soggettivi e dovettero sviluppare una metodologia oggettiva, come descritto da H. S. Jennings (1906) nel suo influente lavoro *Il comportamento degli organismi inferiori*. Applicando agli umani questo punto di vista, il movimento **behaviorista**, guidato da John Watson (1878–1958), rifiutò *qualsiasi* teoria riguardante i processi mentali sulla base dell'affermazione che l'introspezione non poteva fornire alcun dato affidabile. I behavioristi insistevano sullo studio esclusivo delle misurazioni

behaviorismo

⁹ La Legge di Moore afferma che il numero di transistor per pollice quadrato raddoppia in un periodo che va da 1 a 1,5 anni. La capacità del cervello umano raddoppia in un periodo che va da 2 a 4 milioni di anni.

delle percezioni (o *stimoli*) forniti a un animale e delle risultanti azioni (o *risposte*). Costrutti mentali come conoscenze, credenze, scopi e meccanismi di ragionamento venivano liquidati come una non scientifica “psicologia popolare”. Il behaviorismo riuscì a scoprire un sacco di cose sui ratti e i piccioni, ma ebbe meno fortuna nella comprensione degli esseri umani. Ciononostante, esercitò una grande influenza sulla psicologia (in particolare negli Stati Uniti) tra gli anni '20 e il 1960.

La visione del cervello come un dispositivo per l'elaborazione di informazione, caratteristica principale della **psicologia cognitiva**, si può far risalire almeno ai lavori di William James¹⁰ (1842–1910). Anche Helmholtz sostenne che la percezione implicasse una qualche forma di inferenza logica inconscia. Negli Stati Uniti il punto di vista cognitivo fu in gran parte eclissato dal behaviorismo, ma alla Applied Psychology Unit di Cambridge, diretta da Frederic Bartlett (1886–1969), la modellazione cognitiva poté invece fiorire. *The Nature of Explanation* (1943), scritto da uno studente e successore di Bartlett di nome Kenneth Craik, ristabilì con forza la legittimità di termini “mentali” come credenza e scopo, sostenendo che fossero altrettanto scientifici dell’uso di pressione e temperatura quando si parla di gas, benché questi siano fatti di molecole per le quali non ha senso parlare di tali grandezze. Craik specificò tre requisiti fondamentali per un agente basato sulla conoscenza: (1) lo stimolo dev’essere tradotto in una rappresentazione interna; (2) la rappresentazione dev’essere manipolata da processi cognitivi per ottenere nuove rappresentazioni interne; (3) queste ultime devono essere a loro volta trasformate in azione. La sua spiegazione del perché questo sia un buon progetto per un agente è chiara:

Se l’organismo porta nella sua testa un “modello in scala” della realtà esterna e delle proprie possibili azioni sarà in grado di provare varie alternative, decidere quali di esse sia la migliore, reagire a situazioni future prima che si manifestino, utilizzare la conoscenza di eventi passati per gestire quelli presenti e futuri, e sotto ogni aspetto reagire in modo molto più ricco, affidabile e competente alle emergenze che si troverà a fronteggiare. (Craik, 1943)

Dopo la morte di Craik per un incidente di bicicletta nel 1945, il suo lavoro fu portato avanti da Donald Broadbent, il cui libro *Percezione e comunicazione* (1958) includeva alcuni dei primi modelli di fenomeni psicologici fondati sull’elaborazione di informazione. Intanto, negli Stati Uniti, lo sviluppo di modelli basati su computer portò alla creazione del nuovo campo della **scienza cognitiva**. Si può dire che la disciplina sia nata in durante un workshop nel settembre del 1956 al MIT

¹⁰ William James era il fratello del romanziere Henry James. Si dice scherzosamente che Henry scrivesse letteratura come fosse psicologia e William psicologia come fosse letteratura.

(come vedremo, due soli mesi dopo quello che a Dartmouth aveva segnato la “nascita” dell’IA). A quel workshop George Miller presentò *Il magico numero sette*, Noam Chomsky i *Tre modelli per la descrizione del linguaggio*, Allen Newell e Herbert Simon *La macchina Logic Theory*. Questi tre lavori fondamentali mostrarono come i modelli basati su computer potevano essere usati per discutere rispettivamente la psicologia della memoria, del linguaggio e del pensiero logico. Oggi è una considerazione comune tra gli psicologi che “una teoria cognitiva dovrebbe essere come un programma per computer” (Anderson, 1980), ovvero descrivere dettagliatamente un meccanismo di elaborazione dell’informazione che potrebbe implementare una qualche funzione cognitiva.

Ingegneria informatica (1940–presente)

- ♦ Com’è possibile costruire un computer efficiente?

Perché l’intelligenza artificiale abbia successo, sono necessarie due cose: intelligenza e un artefatto. Il computer è stato l’artefatto prescelto. Il moderno computer elettronico digitale fu inventato in modo indipendente e quasi simultaneamente dagli scienziati di tre delle nazioni coinvolte nella Seconda Guerra Mondiale. Il primo computer funzionante fu l’elettromeccanico Heath Robinson,¹¹ costruito nel 1940 dal gruppo di Alan Turing con un solo scopo: decifrare i messaggi tedeschi. Nel 1943, lo stesso gruppo sviluppò il Colossus, una potente macchina a uso generale basata su tubi a vuoto.¹² Il primo computer programmabile funzionante fu lo Z-3, invenzione di Konrad Zuse nella Germania del 1941. Zuse inventò anche i numeri in virgola mobile e il primo linguaggio di programmazione ad alto livello, Plankalkül. Il primo computer elettronico, l’ABC, fu assemblato da John Atanasoff e dal suo studente Clifford Berry tra il 1940 e il 1942 alla Iowa State University. La ricerca di Atanasoff ricevette poco supporto e riconoscimento; il più influente antenato dei computer moderni si rivelò essere l’ENIAC, sviluppato come parte di un progetto militare segreto all’Università della Pennsylvania da una squadra che includeva John Mauchly e John Eckert.

¹¹ Heath Robinson era un disegnatore di vignette famoso per le sue rappresentazioni di macchine bizzarre e assurdamente complicate il cui scopo era eseguire attività quotidiane, come spalmare il burro su un toast.

¹² Nel dopoguerra Turing voleva usare questi computer per fare ricerca nel campo dell’IA, ad esempio per sviluppare uno dei primi programmi di scacchi (Turing et al., 1953). I suoi sforzi furono impediti dal governo britannico.

Nel mezzo secolo successivo, ogni generazione di hardware per computer ha portato un incremento di velocità e potenza e un abbassamento del prezzo. Le prestazioni raddoppiano ogni 18 mesi circa, e a questo ritmo abbiamo ancora dieci o vent'anni davanti a noi: dopodiché, sarà necessario passare all'ingegneria molecolare o a qualche altra tecnologia completamente nuova.

Naturalmente, sono esistiti dispositivi di calcolo ben prima del computer elettronico. Abbiamo discusso le più antiche macchine automatiche a pagina 10. La prima macchina *programmabile* fu un telaio inventato nel 1805 da Joseph Marie Jacquard (1752–1834), che usava schede perforate per memorizzare complesse istruzioni sugli effetti da incorporare nel tessuto. A metà del XIX secolo, Charles Babbage (1792–1871) progettò due macchine, nessuna delle quali riuscì a completare. Il “Motore Differenziale”, che appare nel frontespizio di questo libro, aveva lo scopo di calcolare tabelle matematiche a uso dell’ingegneria e dei progetti scientifici. È stato finalmente costruito e se ne è dimostrato il corretto funzionamento nel 1991, al Museo delle Scienze di Londra (Swade, 1993). Il “Motore Analitico” di Babbage era molto più ambizioso: comprendeva una memoria indirizzabile, programmi memorizzati e salti condizionali, e fu il primo artefatto capace di calcoli universali. La collega di Babbage Ada Lovelace, figlia del poeta Lord Byron, fu forse il primo programmatore della storia (il linguaggio di programmazione Ada è così chiamato in suo onore). Ella scrisse programmi per il Motore Analitico ancora incompiuto e arrivò a supporre che la macchina potesse giocare a scacchi o comporre musica.

L’IA è in debito anche con la parte software degli studi informatici, che ha messo a disposizione sistemi operativi, linguaggi di programmazione e gli strumenti necessari per scrivere programmi moderni (nonché molti articoli sull’argomento). Ma questa è un’area in cui il debito è stato ripagato: gli studi di intelligenza artificiale hanno esplorato per la prima volta idee che si sono poi diffuse nell’informatica generale, tra cui la divisione di tempo, gli interpreti interattivi, i PC dotati di finestre e mouse, gli ambienti di sviluppo rapido, la struttura dati della lista concatenata, la gestione automatica della memoria e altri concetti chiave della programmazione simbolica, funzionale, dinamica e object-oriented.

Teoria del controllo e cibernetica (1948–presente)

- ◆ Come possono degli artefatti funzionare autonomamente?

Ctesibio di Alessandria (c. 250 a.C.) costruì la prima macchina a controllo autonomo: un orologio ad acqua dotato di un regolatore che manteneva il flusso d’acqua che lo attraversava costante e predicibile. Quest’invenzione cambiò la definizione di quello che potevano eseguire gli artefatti. In precedenza, solo gli esseri viventi potevano modificare il loro comportamento per reagire a cambiamenti nell’ambiente. Altri esempi di sistemi di controllo che si “auto-regolano” attraverso la retroazione includono il regolatore per motori a vapore di James Watt

(1736–1819) e il termostato inventato da Cornelis Drebbel (1572–1633), che è anche l'inventore del sottomarino. La teoria matematica dei sistemi stabili retroazionati fu sviluppata nel XVIII secolo.

La figura centrale nello sviluppo di quella che oggi viene chiamata **teoria del controllo** fu Norbert Wiener (1894–1964). Wiener fu un brillante matematico che lavorò tra gli altri con Bertrand Russell prima di sviluppare un interesse per i sistemi di controllo biologici e meccanici e la loro relazione con la cognizione. Come Craik (che pure usò i sistemi di controllo come modelli psicologici), Wiener e i suoi colleghi Arturo Rosenblueth e Julian Bigelow sfidarono l'ortodossia behaviorista (Rosenblueth et al., 1943). Nella loro concezione, il comportamento volontario scaturiva da un meccanismo di regolazione che cerca di minimizzare l'“errore”, ovvero la differenza tra lo stato corrente del sistema e quello desiderato. Alla fine degli anni '40 Wiener, insieme a Warren McCulloch, Walter Pitts e John von Neumann, organizzò una serie di incontri dedicati ai nuovi modelli cognitivi matematici e computazionali che influenzarono molti altri ricercatori nelle scienze comportamentali. Il libro di Wiener *Cybernetics* (1948) divenne un bestseller e rivelò al grande pubblico la possibilità di realizzare macchine intelligenti.

La moderna teoria del controllo, in particolare la branca nota come controllo ottimo stocastico, ha come scopo la progettazione di sistemi che massimizzano nel tempo una **funzione obiettivo**. Questo a grandi linee corrisponde alla nostra idea di IA: la costruzione di sistemi che agiscono in modo ottimo. Ma allora perché l'IA e la teoria del controllo sono due discipline differenti, a maggior ragione vista l'affinità dei rispettivi fondatori? La risposta sta nelle diverse tecniche matematiche padroneggiate dagli studiosi dei due campi e la loro influenza sulle rispettive visioni del mondo e sui relativi problemi. L'analisi e l'algebra delle matrici, gli strumenti della teoria del controllo, si prestano bene a sistemi che si possono descrivere con insiemi finiti di variabili continue; inoltre, l'analisi esatta si può tipicamente effettuare solo nel caso di sistemi *lineari*. L'IA fu fondata in parte proprio per superare le limitazioni della matematica tipica della teoria del controllo degli anni '50. Strumenti come l'inferenza logica e la computazione permisero agli studiosi di IA di affrontare problemi, come il linguaggio naturale, la visione e la pianificazione, che si ponevano completamente fuori dal campo d'azione della teoria del controllo.

teoria del controllo

Cybernetics

funzione obiettivo

Linguistica (1957–presente)

- ◆ Qual è il collegamento tra linguaggio e pensiero?

Nel 1957 B. F. Skinner pubblicò *Il comportamento verbale*: si trattava di un resoconto vasto e dettagliato dell'approccio behaviorista all'apprendimento del linguaggio, scritto dal più grande esperto del campo. Curiosamente, una recensione del libro divenne famosa quanto il libro stesso, ed ebbe come conseguenza la quasi totale scomparsa del behaviorismo. L'autore della recensione era Noam Chom-

sky, che aveva appena pubblicato un lavoro dedicato alla propria teoria, *Strutture sintattiche*. Chomsky mostrò che la teoria behaviorista non considerava l'aspetto creativo del linguaggio: non spiegava com'è possibile che un bambino possa comprendere e creare frasi che non ha mai sentito prima. La teoria di Chomsky, basata su modelli sintattici che risalgono al linguista indiano Panini (circa 350 a.C.) poteva spiegare la creatività, e a differenza delle teorie precedenti era abbastanza formale da poter essere, in via di principio, programmata.

La linguistica moderna e l'intelligenza artificiale, quindi, sono "nate" nello stesso momento, e sono cresciute insieme, mescolandosi in un campo di studi ibrido denominato **linguistica computazionale** o **elaborazione del linguaggio naturale**. Il problema della comprensione del linguaggio naturale si rivelò presto molto più complesso di quanto sembrasse nel 1957, perché richiede la comprensione dell'argomento trattato e del contesto, e non solamente della struttura delle frasi. Oggi questo può sembrare ovvio, ma non fu pienamente compreso fino agli anni '60. Gran parte del lavoro iniziale sulla **rappresentazione della conoscenza** (lo studio di come esprimere la conoscenza in una forma che possa essere usata da un computer per ragionare) fu collegato al linguaggio e si appoggiò alla ricerca dei linguisti, che a loro volta facevano riferimento a decenni di lavoro sull'analisi filosofica del linguaggio.

1.3 La storia dell'intelligenza artificiale

Dopo aver delineato sufficientemente il suo background culturale, possiamo finalmente parlare dello sviluppo dell'intelligenza artificiale.

La gestazione dell'intelligenza artificiale (1943–1955)

Il primo lavoro oggi generalmente considerato appartenente all'IA fu svolto da Warren McCulloch e Walter Pitts (1943) pescando da tre fonti: la conoscenza delle basi della fisiologia e della funzione dei neuroni nel cervello, un'analisi formale della logica proposizionale di Russell e Whitehead e la teoria della computazione di Turing. I due proposero un modello neuronale artificiale in cui ogni neurone era caratterizzato dallo stato "acceso" o "spento", e la cui accensione si verificava in risposta allo stimolo da parte di un numero sufficiente di neuroni adiacenti. Lo stato del neurone veniva così concepito come "di fatto equivalente alla proposizione corrispondente agli stimoli adeguati". McCulloch e Pitts mostrarono, tra le altre cose, che ogni funzione computabile poteva essere calcolata da una rete di neuroni collegati e che tutti gli operatori logici (and, or, not, eccetera) potevano essere implementati con semplici strutture a rete. Suggerirono inoltre che reti neurali adeguatamente definite potessero essere capaci di apprendere. Donald Hebb

(1949) formulò una semplice regola di aggiornamento per la modifica dei pesi delle connessioni tra i neuroni: la sua regola, chiamata oggi **apprendimento hebbiano**, rimane a tutt'oggi un modello importante.

Marvin Minsky e Dean Edmonds, due studenti di dottorato del dipartimento di matematica di Princeton, costruirono il primo computer basato su rete neurale nel 1951. Lo SNARC, com'era chiamato, utilizzava 3000 tubi a vuoto e un sistema automatico di pilotaggio riciclato da un bombardiere B-24 per simulare una rete di 40 neuroni. La commissione di dottorato di Minsky non era convinta che questo tipo di lavoro si potesse considerare matematica, ma sembra che von Neumann abbia ribattuto: "Se non lo è ora, lo sarà un giorno". Minsky più tardi avrebbe dimostrato teoremi importanti circa le limitazioni della ricerca sulle reti neurali.

Ci furono diversi altri esempi di lavori che possono essere considerati come intelligenza artificiale, ma una visione completa dell'IA fu espressa per la prima volta da Alan Turing nel suo articolo del 1950 *Computing Machinery and Intelligence*. L'articolo introduceva il test di Turing, l'apprendimento automatico, gli algoritmi genetici e l'apprendimento per rinforzo.

La nascita dell'intelligenza artificiale (1956)

A Princeton si trovava un'altra figura importante per l'IA, John McCarthy. Dopo la laurea passò al Dartmouth College, che sarebbe diventato il luogo di nascita ufficiale della disciplina. McCarthy convinse Minsky, Claude Shannon e Nathaniel Rochester ad aiutarlo a riunire i ricercatori americani interessati alla teoria degli automi, alle reti neurali e allo studio dell'intelligenza. Nell'estate del 1956 venne organizzato un workshop di due mesi a Dartmouth: in tutto parteciparono dieci persone, tra cui Trenchard More da Princeton, Arthur Samuel dall'IBM, Ray Solomonoff e Oliver Selfridge dal MIT.

Si può dire che due ricercatori della Carnegie Tech,¹³ Allen Newell e Herbert Simon, rubarono la scena. Benché gli altri avessero portato idee e in certi casi programmi per applicazioni particolari come il gioco della dama, Newell e Simon potevano già mostrare un programma capace di ragionare, il Logic Theorist (LT), circa il quale Simon dichiarò "abbiamo inventato un programma per computer capace di pensare in modo non numerico, e abbiamo quindi risolto l'antichissimo problema mente-corpo".¹⁴ Subito dopo il workshop, il programma fu in grado di

¹³ Ora Carnegie Mellon University (CMU).

¹⁴ Newell e Simon inventarono anche un linguaggio per l'elaborazione di liste, IPL, per scrivere LT. Non avevano compilatore, e dovevano tradurlo a mano in codice macchina. Per evitare di fare errori lavoravano in parallelo, dicendo insieme ad alta voce i numeri binari corrispondenti a ogni istruzione per assicurarsi che coincidessero.

dimostrare la maggior parte dei teoremi nel secondo capitolo dei *Principia Mathematica* di Russell e Whitehead. Si dice che Russell fosse deliziato quando Simon gli mostrò che il programma aveva escogitato una dimostrazione di un teorema più breve di quella inclusa nei *Principia*. Gli editor del *Journal of Symbolic Logic* furono meno impressionati e rifiutarono l'articolo che indicava come coautori Newell, Simon e Logic Theorist.

Il workshop di Dartmouth non portò a particolari innovazioni, ma servì a far incontrare tutti i protagonisti principali della disciplina. Per i successivi vent'anni, il campo sarebbe stato dominato da queste persone, i loro studenti e i loro colleghi al MIT, CMU, Stanford e IBM. Forse la conseguenza più duratura del workshop fu l'adozione del nome coniato da McCarthy per la nuova scienza: **intelligenza artificiale**. Forse “razionalità computazionale” sarebbe stata una denominazione più accurata, ma “IA” ha attecchito.

Leggendo la *proposal* per il workshop di Dartmouth (McCarthy et al., 1955) possiamo vedere perché era necessario che l'IA diventasse una branca scientifica separata. Perché tutto il lavoro di ricerca nel campo dell'IA non avrebbe potuto essere svolto come parte della teoria del controllo, della ricerca operativa o della teoria delle decisioni, che dopotutto hanno obiettivi molto simili? E anche, perché l'IA non è semplicemente una branca della matematica? La prima risposta è che l'IA, fin dall'inizio, ha accolto l'idea di riprodurre facoltà umane come la creatività, il miglioramento di sé e l'uso del linguaggio; nessuna delle discipline citate trattava questi argomenti. La seconda risposta sta nella metodologia: l'IA è l'unico campo di ricerca definito chiaramente come una branca dell'informatica (sebbene anche la ricerca operativa ponga molta enfasi sulle simulazioni al computer), ed è l'unica scienza che si propone di costruire macchine che funzionino autonomamente in ambienti complessi e mutevoli.

Primi entusiasmi, grandi aspettative (1952–1969)

I primi anni dell'IA furono pieni di successi, ancorché limitati. Con i computer e gli strumenti di programmazione primitivi disponibili all'epoca, e dato che solo qualche anno prima non si pensava che i calcolatori potessero svolgere molto più che operazioni aritmetiche, ogni volta che un computer faceva qualsiasi cosa anche vagamente ingegnosa era considerato stupefacente. Gli intellettuali, per la maggior parte, preferirono continuare a pensare che “una macchina non potrà mai fare *X*”. Naturalmente, i ricercatori dell'IA risposero dimostrando una *X* dietro l'altra. John McCarthy battezzò questo periodo come l'era del “Guarda mamma, senza mani!”.

Ai primi successi di Newell e Simon fece seguito il General Problem Solver, o GPS. A differenza di Logic Theorist, questo programma era stato progettato fin dal principio per imitare i procedimenti umani di risoluzione dei problemi. All'interno della classe limitata dei problemi che poteva risolvere, l'ordine in cui il pro-

gramma considerava i sotto-oggetti e le possibili azioni era simile a quello adottato dagli esseri umani. GPS quindi fu probabilmente il primo programma ad adottare l'approccio del "pensare umanamente". Il successo di GPS e dei suoi successori come modelli della cognizione portarono Newell e Simon (1976) a formulare la famosa ipotesi del **sistema fisico simbolico**, che afferma che "un sistema fisico simbolico ha i mezzi necessari e sufficienti per agire in modo generalmente intelligente". Quello che intendevano è che ogni sistema (umano o artificiale) che dimostra di possedere intelligenza deve funzionare manipolando strutture dati composte di simboli. Vedremo in seguito che quest'ipotesi è stata contestata da più parti.

All'IBM, Nathaniel Rochester e i suoi colleghi svilupparono alcuni dei primi programmi di intelligenza artificiale. Herbert Gelernter (1959) scrisse il Geometry Theorem Prover, capace di dimostrare teoremi che avrebbero dato qualche grattacapi a molti studenti di matematica. A partire dal 1952, Arthur Samuel scrisse una serie di programmi per il gioco della dama che arrivarono a giocare al livello di un buon dilettante. Durante lo sviluppo dimostrò la falsità dell'idea che i computer possano fare solo quello che viene loro detto: il software infatti imparò presto a giocare meglio del suo creatore. Questo programma venne mostrato alla televisione nel febbraio del 1956, suscitando forte impressione. Come per Turing, anche per Samuel fu difficile trovare un computer che avesse tempo libero per lui: lavorando di notte, poté utilizzare le macchine che si trovavano ancora in fase di test nella fabbrica dell'IBM.

John McCarthy passò da Dartmouth al MIT e lì diede all'IA tre contributi fondamentali in un solo anno, il 1958. Nel "AI LabMemo No.1" definì il linguaggio di alto livello **Lisp**, che sarebbe diventato il più importante linguaggio di programmazione per l'IA. È il secondo più vecchio linguaggio di alto livello ancora in uso, superato di un solo anno dal FORTRAN. Con Lisp McCarthy aveva a disposizione uno strumento potente, ma rimaneva il problema dell'accesso ai computer, ancora pochi e molto costosi. In risposta, insieme ad alcuni colleghi del MIT inventò il *time sharing*, la condivisione di tempo nell'accesso alle risorse hardware. Sempre nel 1958 McCarthy pubblicò un articolo intitolato *Programs with Common Sense*, nel quale descrisse Advice Taker, un ipotetico programma che può essere considerato come il primo sistema completo di IA. Come Logic Theorist e Geometry Theorem Prover, il programma di McCarthy sfruttava la conoscenza per cercare la soluzione ai problemi. A differenza degli altri, tuttavia, il suo scopo era incorporare conoscenza generale del mondo. Ad esempio, con pochi semplici assiomi, il programma sarebbe stato capace di generare un piano per guidare fino all'aeroporto e prendere un aereo. Il programma era anche progettato in modo da poter accettare nuovi assiomi durante la normale esecuzione, acquisendo così competenze in nuove aree *senza essere riprogrammato*. Così Advice Taker incarnava il

sistema fisico
simbolico

principio fondamentale della rappresentazione della conoscenza e del ragionamento: è utile avere una rappresentazione formale ed esplicita del mondo e del modo in cui le azioni di un agente lo modificano, ed essere in grado di manipolare queste rappresentazioni per mezzo di processi deduttivi. È stupefacente quanta parte di questo articolo del 1958 sia valida ancor oggi.

Il 1958 è anche l'anno in cui Marvin Minsky passò al MIT. La sua collaborazione iniziale con McCarthy, comunque, ebbe vita breve: quest'ultimo rivolgeva l'attenzione alla rappresentazione e al ragionamento all'interno di una logica formale, mentre a Minsky interessava soprattutto far funzionare i programmi, tanto che a un certo punto arrivò a sviluppare una prospettiva anti-logica. Nel 1963, McCarthy fondò il laboratorio di IA a Stanford. Il suo progetto di usare la logica per costruire una versione definitiva di Advice Taker poteva ora avvalersi della scoperta del metodo di risoluzione da parte di J. A. Robinson (si tratta di un algoritmo completo per la dimostrazione di teoremi per la logica del primo ordine; v. Capitolo 9). Il lavoro a Stanford si concentrava soprattutto su metodi di uso generale per il ragionamento logico. Tra le applicazioni della logica c'erano i sistemi di domanda-risposta e di pianificazione di Cordell Green (Green, 1969b) e il progetto di robotica Shakey, presso il nuovo Stanford Research Institute (SRI). Quest'ultimo fu il primo progetto a dimostrare un'integrazione completa tra il ragionamento logico e l'attività fisica.

Minsky supervisionò un gruppo di studenti che scelse di occuparsi di problemi circoscritti la cui soluzione sembrava richiedere un certo grado di intelligenza. Questi domini limitati divennero famosi come **micromondi**. Il programma SAINT (1963a) di James Slagle fu capace di risolvere problemi di integrali chiusi tipici dei corsi universitari di primo anno. Il programma ANALOGY (1968) di Tom Evans poteva risolvere problemi basati su analogie geometriche come quelli che appaiono nei test per il quoziente intellettivo (v. Figura 1.4). Il programma STUDENT (1967) di Daniel Bobrow risolveva problemi di algebra espressi in forma di racconto, come il seguente:

Se il numero di clienti di Tom è due volte il quadrato del venti per cento del numero degli spot pubblicitari che commissiona, e gli spot commissionati sono 45, quanti clienti avrà Tom?

Il micromondo più famoso era il mondo dei blocchi, che consisteva in una serie di blocchetti solidi disposti su un tavolo (o, più spesso, la sua simulazione), come si può vedere nella Figura 1.5. Un obiettivo tipico, in questo mondo, è sistemare i blocchi in una certa configurazione, usando un braccio meccanico che può prendere un blocco per volta. Il mondo dei blocchi fu l'ambiente sperimentale per il progetto di visione artificiale di David Huffman (1971), il lavoro di David Waltz (1975) dedicato alla visione e alla propagazione dei vincoli, la teoria dell'apprendimento di Patrick Winston (1970), il programma per la comprensione del linguaggio naturale di Terry Winograd (1972) e il pianificatore di Scott Fahlman (1974).

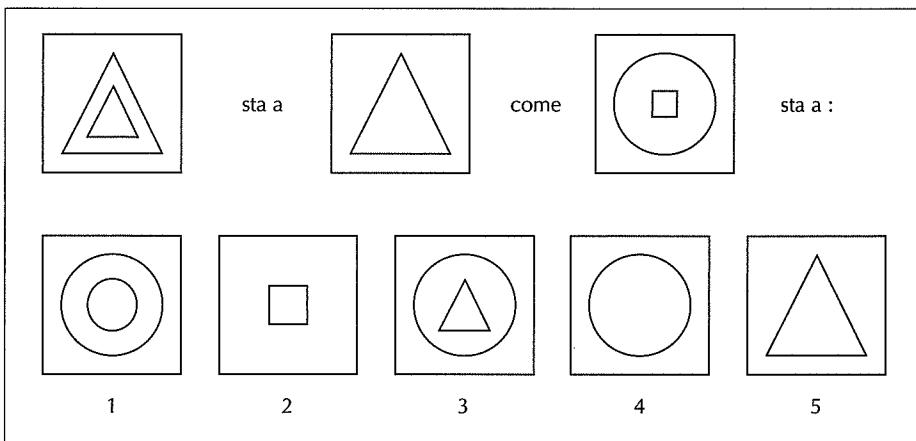


Figura 1.4
Un esempio di problema che può essere risolto dal programma di Evans ANALOGY.

Anche i primi lavori sulle reti neurali di McCulloch e Pitts vennero ripresi e approfonditi: il lavoro di Winograd e Cowan (1963) mostrò in che modo si poteva usare un grande numero di elementi per rappresentare collettivamente un singolo concetto, con un incremento di robustezza e parallelismo. I metodi di apprendimento di Hebb furono migliorati da Bernie Widrow (Widrow e Hoff, 1960; Widrow, 1962), che battezzò le proprie reti adaline, e da Frank Rosenblatt (1962) con i percettroni. Rosenblatt dimostrò il teorema di convergenza del percettone, mostrando che il suo algoritmo di apprendimento poteva modificare la forza delle connessioni di un percettone in modo da corrispondere a qualsiasi possibile input, a patto che tale corrispondenza esistesse.

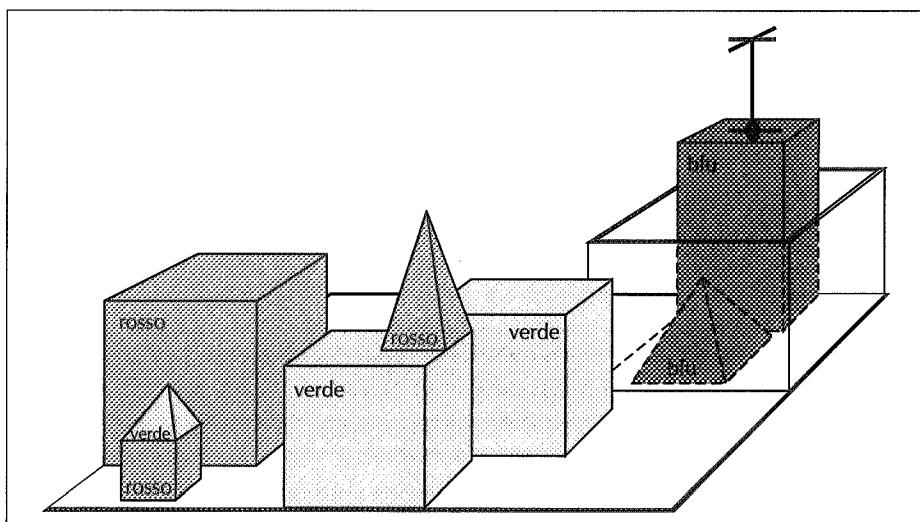


Figura 1.5
Una scena nel mondo dei blocchi. Il programma SHRDLU (Winograd, 1972) ha appena completato l'esecuzione del comando "trova un blocco più alto di quello che stai portando e spostalo all'interno della scatola".

Una dose di realtà (1966–1973)

Fin dagli inizi, i ricercatori di intelligenza artificiale non hanno avuto paura di fare predizioni sui loro futuri successi. Si cita spesso il seguente brano di Herbert Simon, risalente al 1957:

Il mio scopo non è stupire o sbalordire – ma il modo più semplice in cui posso esprimerti è dicendo che ora nel mondo esistono macchine che possono pensare, imparare e creare. Inoltre, la loro abilità nel fare queste cose aumenterà rapidamente finché, in un futuro vicino, il campo dei problemi che potranno gestire avrà la stessa estensione di quello a cui si è applicata la mente umana.

Termini come “un futuro vicino” si possono interpretare in molti modi, ma Simon ha anche fatto predizioni più concrete: che entro dieci anni un computer sarebbe stato campione mondiale di scacchi, e che una macchina avrebbe dimostrato un teorema matematico di notevole importanza. Queste predizioni si sono (approssimativamente) avvurate in quarant’anni anziché dieci. L’eccessiva sicurezza di Simon era dovuta alle promettenti prestazioni dei primi sistemi IA applicati a semplici esempi. In quasi tutti i casi, però, tali sistemi hanno fallito miseramente quando sono stati applicati a problemi più complessi o appartenenti a tipologie differenti.

Il primo tipo di difficoltà sorse perché i primi sistemi contenevano poca o nessuna informazione specifica riguardante l’argomento delle loro elaborazioni; il loro successo era basato su semplici manipolazioni sintattiche. Una storia tipica appartiene al campo della traduzione automatica del linguaggio naturale, che fu generosamente sovvenzionato dal Consiglio Nazionale per la Ricerca degli Stati Uniti nel tentativo di accelerare la traduzione degli articoli scientifici russi subito dopo il lancio dello Sputnik nel 1957. All’inizio si pensava che delle semplici trasformazioni sintattiche basate sulla grammatica del russo e dell’inglese, unite all’uso di un dizionario elettronico, sarebbero state sufficienti a preservare il significato preciso delle frasi. Di fatto una traduzione richiede una conoscenza generale degli argomenti trattati per risolvere le ambiguità e determinare il significato di una frase. Le difficoltà incontrate sono ben esemplificate dalla famosa traduzione della frase “lo spirito è forte ma la carne è debole” con “la vodka è buona ma la bistecca è marcia”. Nel 1966 il rapporto di un comitato consultivo dichiarò che “non c’è stata alcuna traduzione automatica di testi scientifici di carattere generale e non se ne prevede alcuna nell’immediato futuro”. I finanziamenti americani ai progetti accademici di traduzione del linguaggio naturale furono totalmente cancellati. Oggi la traduzione automatica è ancora uno strumento imperfetto, ma viene ampiamente usata per documenti tecnici, commerciali, governativi e su Internet.

Il secondo ostacolo era l’intrattabilità di molti dei problemi che l’IA stava cercando di risolvere. Molti dei primi programmi giungevano alla soluzione provando varie combinazioni di passi. Questa strategia inizialmente funzionava, perché i micromondi contenevano pochi oggetti e quindi le azioni possibili erano molto

poche e le sequenze risolutive corte. Prima dello sviluppo della teoria della complessità computazionale, l'idea diffusa era che la scalabilità verso problemi di maggiori dimensioni fosse solo una questione di hardware più potente e memorie più grandi. L'ottimismo che aveva accompagnato lo sviluppo dei dimostratori automatici di teoremi, ad esempio, fu presto smorzato quando i ricercatori non riuscirono a dimostrare teoremi che coinvolgessero più di qualche dozzina di fatti. Il fatto che un programma possa, in via di principio, trovare una soluzione, non significa affatto che il programma contenga i meccanismi necessari per trovarla nella pratica.

L'illusione di una potenza computazionale illimitata non rimase confinata ai programmi dedicati alla risoluzione di problemi. I primi esperimenti di evoluzione automatica (ora chiamati algoritmi genetici) (Friedberg, 1958; Friedberg et al., 1959) erano basati sull'idea, certamente corretta, che apportando una serie appropriata di piccole modifiche a un programma, sarebbe stato possibile generarne un altro con prestazioni particolarmente buone per una qualsiasi particolare attività. L'idea era di provare delle mutazioni casuali applicando poi un processo di selezione per mantenere solo quelle che sembravano utili. Nonostante migliaia di ore di tempo della CPU, fu praticamente impossibile registrare alcun progresso. Gli algoritmi genetici moderni usano rappresentazioni migliori e hanno avuto più successo.

La mancanza di una soluzione al problema della "esplosione combinatoria" fu una delle critiche principali all'IA contenute nel rapporto Lighthill (Lighthill, 1973), che fu alla base della decisione, da parte del governo britannico, di tagliare i finanziamenti per l'IA in tutte le università tranne due (la tradizione orale racconta una storia piuttosto differente e molto più colorita, con ambizioni politiche e contrasti personali la cui narrazione va oltre i nostri scopi).

Un terzo problema sorse a causa di alcuni limiti fondamentali delle strutture di base usate per generare comportamento intelligente. Ad esempio, il libro di Minsky e Papert *Percetroni* (1969) dimostrò che benché i percetroni (una forma semplice di reti neurali) potessero apprendere tutto ciò che erano in grado di rappresentare, potevano in effetti rappresentare ben poco. In particolare, un percetrone a due input non poteva imparare a distinguere quando i suoi due input erano diversi. Benché questi risultati non si applicassero a reti più complesse a diversi livelli, il finanziamento alla ricerca sulle reti neurali presto si ridusse fin quasi a zero. Ironicamente i nuovi algoritmi di apprendimento con retropropagazione per reti a più livelli, che avrebbero causato un grande ritorno alla ricerca sulle reti neurali alla fine degli anni '80, furono in realtà scoperti inizialmente nel 1969 (Bryson e Ho, 1969).



evoluzione automatica

metodi deboli

Sistemi basati sulla conoscenza: la chiave per il potere? (1969–1979)

La visione generale della risoluzione di problemi, che aveva preso forma durante il primo decennio di studi sull'IA, era quella di un meccanismo di ricerca di uso generale che cercava di mettere insieme dei passi elementari di ragionamento, uno dopo l'altro, fino a trovare soluzioni complete. Questi approcci sono stati chiamati metodi deboli perché, sebbene siano generali, non sono in grado di "scalare" verso l'alto e risolvere istanze di problemi molto grandi o difficili. L'alternativa ai metodi deboli è usare conoscenza più potente, specifica del dominio, che permette di intraprendere passi di ragionamento più ampi e può gestire più facilmente i casi tipici di aree ristrette di esperienza. Si potrebbe dire che, per risolvere un problema difficile, bisogna quasi conoscerne già la soluzione.

Il programma DENDRAL (Buchanan et al., 1969) è uno delle prime applicazioni di quest'approccio. Fu sviluppato a Stanford, quando Ed Feigenbaum (un ex-studente di Herbert Simon), Bruce Buchanan (un filosofo trasformato in informatico) e Joshua Lederberg (un genetista vincitore del premio Nobel) unirono i loro sforzi per risolvere il problema della ricostruzione dell'informazione molecolare partendo dai dati forniti da uno spettrometro di massa. L'input del programma consisteva nella formula elementare di una molecola (ad esempio $C_6H_{13}NO_2$) e in uno spettro, che forniva le masse dei vari frammenti di molecola generati in seguito a un bombardamento di elettroni. Ad esempio, lo spettro di massa avrebbe potuto contenere un picco per $m = 15$, che corrisponde alla massa di un frammento di metile (CH_3).

La versione iniziale e "ingenua" del programma generava tutte le strutture possibili a partire dalla formula, quindi prediceva quale spettro sarebbe stato osservato per ognuna, confrontando infine le predizioni con lo spettro effettivamente misurato. Come si può facilmente intuire, questo problema diventa intrattabile non appena le molecole crescono di dimensioni. I ricercatori del progetto DENDRAL consultarono dei chimici analitici e scoprirono che il loro metodo di lavoro consisteva nel cercare dei pattern ben conosciuti di picchi nello spettro, che avrebbero indicato delle strutture di occorrenza comune nella molecola. Ad esempio, per riconoscere un sottogruppo chetone ($C=O$), che ha peso 28, si usa la seguente regola:

if ci sono due picchi in x_1 e x_2 tali che
(a) $x_1 + x_2 = M + 28$ (M è la massa dell'intera molecola);
(b) $x_1 - 28$ è un picco;
(c) $x_2 - 28$ è un picco;
(d) Almeno uno tra x_1 e x_2 è alto
then c'è un sottogruppo chetone

Sapere che la molecola contiene una particolare struttura riduce enormemente il numero di possibili candidate. DENDRAL era potente perché

Tutta la conoscenza teorica importante per risolvere questi problemi è stata trasferita a partire dalla sua forma generale nel [componente dedicato alla predizione dello spettro] ("principî fondamentali") in forme speciali di grande efficienza ("ricette di cucina") (Feigenbaum et al., 1971).

DENDRAL rappresentò un passo avanti importante perché fu il primo sistema *a conoscenza intensiva* di successo: la sua capacità era dovuta a un grande numero di regole speciali. I sistemi successivi incorporarono anche il principio fondamentale dell'approccio che McCarthy aveva applicato ad Advice Taker: la separazione netta tra la conoscenza (espressa sotto forma di regole) e il ragionamento.

Forti di quest'esperienza, Feigenbaum e altri a Stanford diedero inizio all'Heuristic Programming Project (HPP), il cui scopo era investigare come la nuova metodologia dei sistemi esperti poteva essere applicata ad altre aree della conoscenza umana. Il più importante sviluppo successivo si ebbe nell'area della diagnosi medica: Feigenbaum, Buchanan e il Dott. Edward Shortliffe svilupparono MYCIN per diagnosticare le infezioni del sangue. Con circa 450 regole, MYCIN poteva offrire prestazioni pari a quelle di molti esperti, e certamente migliori di quelle dei dottori neolaureati. Rispetto a DENDRAL c'erano due grandi differenze. Prima di tutto, diversamente da quello MYCIN non conteneva alcun modello teorico generale da cui poter dedurre le regole, che dovevano essere acquisite con una lunga serie di interviste agli esperti, che a loro volta le avevano apprese dai libri, dai colleghi e con l'esperienza personale. In secondo luogo, le regole dovevano rispecchiare l'incertezza intrinsecamente associata alla conoscenza medica: MYCIN incorporava per questo un metodo di calcolo di incertezza denominato fattori di certezza che in quel periodo sembrava corrispondere bene alle modalità con le quali i dottori utilizzavano le informazioni disponibili durante il processo diagnostico.

L'importanza della conoscenza del dominio divenne evidente anche nel campo della comprensione del linguaggio naturale. Benché il sistema SHIRDLU di Winograd avesse suscitato parecchio interesse, la sua dipendenza dall'analisi sintattica gli causava alcuni dei problemi che avevano afflitto i primi programmi di traduzione automatica. Il sistema poteva superare certe ambiguità e capire a cosa si riferivano i pronomi, ma ciò era dovuto principalmente al fatto che era stato progettato specificatamente per un'area di applicazione – il mondo dei blocchi. Molti ricercatori, tra cui Eugene Charniak, un compagno di studi di Winograd al MIT, suggerirono che la comprensione del linguaggio richiedesse una conoscenza generale del mondo nonché di un metodo generale per utilizzarla.

A Yale, Roger Schank, un linguista trasformato in ricercatore di intelligenza artificiale, per enfatizzare questo punto arrivò a sostenere che "non esiste una cosa come la sintassi", affermazione che scatenò le ire di molti linguisti, ma ebbe il pre-

~~DENDRAL~~

sistemi esperti

~~MYCIN~~

gio di suscitare un'utile discussione. Schank costruì una serie di programmi insieme ai suoi studenti (Schank e Abelson, 1977; Wilensky, 1978; Schank e Riesbeck, 1981; Dyer, 1983), tutti dedicati alla comprensione del linguaggio naturale. L'enfasi tuttavia non era posta sul linguaggio in sé, ma più che altro sui problemi connessi alla rappresentazione e al ragionamento applicati alla conoscenza richiesta per la sua comprensione. I problemi spaziavano dalla rappresentazione di situazioni tipiche (Cullingford, 1981), alla descrizione dell'organizzazione della memoria umana (Rieger, 1976; Kolodner, 1983) fino alla comprensione di piani e obiettivi (Wilensky, 1983).

La sempre maggiore crescita di applicazioni a problemi reali provocò un parallelo incremento della domanda di schemi efficaci per la rappresentazione della conoscenza. Furono sviluppati molti linguaggi diversi per la rappresentazione e il ragionamento. Alcuni erano basati sulla logica: ad esempio Prolog divenne popolare in Europa, e la famiglia di linguaggi PLANNER negli Stati Uniti. Altri, seguendo le strutture che Minsky aveva denominato frame (1975), adottarono un approccio più strutturato, raccogliendo fatti che riguardavano particolari tipi di oggetti ed eventi e organizzando tali tipi in una grande gerarchia che ricordava la tassonomia biologica.

L'IA diventa un'industria (1980–presente)

Il primo sistema esperto ad avere successo commerciale fu R1, che cominciò ad operare presso la Digital Equipment Corporation (McDermott, 1982). Il programma aiutava la configurazione degli ordini di nuovi computer; si stima che nel 1986 la compagnia risparmiasse grazie a lui circa 40 milioni di dollari all'anno. Nel 1988 la DEC aveva messo in opera 40 sistemi esperti e ne stava producendo altri. La Du Pont ne contava 100 operativi e altri 500 in fase di sviluppo, che le permettevano di risparmiare circa 10 milioni all'anno. Quasi ogni compagnia americana di un certa grandezza aveva un proprio gruppo di IA e stava utilizzando o considerando l'uso di uno o più sistemi esperti.

Nel 1981 i giapponesi annunciarono il progetto della "Quinta Generazione", un ambizioso piano decennale con l'obiettivo di costruire computer intelligenti usando Prolog. In risposta, gli Stati Uniti formarono la Microelectronics and Computer Technology Corporation (MCC), un consorzio di ricerca che avrebbe dovuto assicurare la competitività nazionale. In entrambi i casi l'AI faceva parte di uno sforzo a largo raggio, che includeva la progettazione di chip e la ricerca sulle interfacce uomo-macchina. Tuttavia, le componenti di MCC e di Quinta Genera-

zione più specificatamente vicine all'IA non riuscirono a raggiungere i loro ambiziosi obiettivi. In Inghilterra, il rapporto Alvey ripristinò i finanziamenti tagliati al tempo del rapporto Lighthill.¹⁵

Globalmente l'industria dell'IA conobbe un boom, passando da pochi milioni di dollari nel 1980 a miliardi di dollari nel 1988. Poco dopo sopravvenne il periodo che fu chiamato "inverno dell'IA", durante il quale molte compagnie soffrirono per l'impossibilità di mantenere le promesse fatte, spesso alquanto stravaganti.

Il ritorno delle reti neurali (1986–presente)

Sebbene alla fine degli anni '70 l'informatica avesse abbandonato quasi completamente il campo delle reti neurali, la ricerca era continuata in seno ad altre discipline. Fisici come John Hopfield (1982) usavano tecniche di meccanica statistica per analizzare le proprietà di memorizzazione e l'ottimizzazione delle reti, trattando collezioni di nodi come fossero insiemi di atomi. Psicologi come David Rumelhart e Geoff Hinton continuavano a studiare i modelli della memoria basati su reti neurali. Ma il vero ritorno si ebbe a metà degli anni '80, quando almeno quattro gruppi differenti reinventarono l'algoritmo di apprendimento con retropropagazione scoperto nel 1969 da Bryson e Ho. L'algoritmo fu applicato a molti problemi di apprendimento in informatica e psicologia, e la pubblicazione dei risultati nella collezione *Parallel Distributed Processing* (Rumelhart e McClelland, 1986) suscitò grande scalpore.

Questi modelli di sistemi intelligenti, chiamati **conessionisti**, furono considerati da qualcuno in diretta opposizione sia ai modelli simbolici di Newell e Simon, sia all'approccio logistico di McCarthy e altri (Smolensky, 1988). Può sembrare ovvio che in qualche modo gli esseri umani manipolino dei simboli: in effetti il libro di Terrence Deacon *The Symbolic Species* (1997) sostiene che questa sia la **caratteristica peculiare** degli uomini. I conessionisti più radicali, però, si domandavano se la manipolazione simbolica fosse veramente utile per spiegare i modelli di cognizione più dettagliati. La questione rimane senza risposta, anche se l'opinione corrente è che gli approcci connessioneista e simbolico siano complementari e non in competizione.

conessionismo

¹⁵ Per evitare imbarazzo, dato che l'Intelligenza Artificiale era stata formalmente cancellata venne inventato un nuovo campo di ricerca, chiamato IKBS (*Intelligent Knowledge-Based Systems*).

L'IA diventa una scienza (1987–presente)

In anni recenti si è verificata una rivoluzione nel campo dell'IA, sia nei contenuti che nelle metodologie di lavoro.¹⁶ Oggi è più comune partire da teorie esistenti che avanzarne di nuove, basare le proprie proposte su teoremi rigorosi o prove sperimentali piuttosto che sull'intuito, e proporre applicazioni reali anziché esempi limitati.

L'IA nacque in parte come movimento di protesta contro le limitazioni imposte da campi di ricerca preesistenti come la teoria del controllo e la statistica, ma ora sta abbracciando quegli stessi campi. Come ha scritto David McAllester (1998),

Nei primi tempi dell'IA sembrava plausibile che nuove forme di calcolo simbolico, come i frame e le reti semantiche, rendessero obsoleta gran parte della teoria classica. Questo portò a una forma di isolazionismo per cui l'IA si trovò in gran parte separata dal resto dell'informatica. L'isolamento è ora superato: si riconosce che l'apprendimento automatico non dovrebbe essere disgiunto dalla teoria dell'informazione, il ragionamento incerto dalla modellazione stocastica, la ricerca dall'ottimizzazione e dal controllo classici, il ragionamento automatico dai metodi formali e dall'analisi statica.

Per quanto riguarda la metodologia, l'IA si pone oggi completamente nell'ambito del metodo scientifico: per essere accettate, le ipotesi devono essere sottoposte a rigorosi esperimenti, e i risultati devono essere convalidati da un'analisi statistica (Cohen, 1995). Grazie all'uso di Internet e all'esistenza di depositi condivisi di codice e dati di test, oggi è possibile replicare gli esperimenti.

Tutto questo è ben esemplificato dal campo del riconoscimento vocale. Negli anni '70, in questo settore furono provate una gran varietà di architetture e approcci diversi: molte di queste erano realizzare *ad hoc* e alquanto fragili, funzionando efficacemente solamente con alcuni esempi accuratamente selezionati. In anni recenti, il campo è stato rapidamente dominato dagli approcci basati sui modelli nascosti di Markov (HMM, *hidden Markov models*). Di questi sono importanti due aspetti: prima di tutto si fondano su una rigorosa teoria matematica; questo ha permesso ai ricercatori di avvalersi dei risultati ottenuti in decenni di lavoro in altri campi. In secondo luogo questi modelli sono generati mediante un processo di apprendimento basato su una grande mole di dati reali. Questo assicu-



¹⁶ Alcuni hanno descritto questo cambiamento come la vittoria dei *neats* (precisini), quelli che pensano che le teorie dell'IA debbano essere fondate sul rigore matematico, nei confronti degli *scruffies* (scapigliati), che invece preferirebbero mettere alla prova un sacco di idee diverse, scrivere un po' di programmi e poi valutare quello che sembra funzionare meglio. Entrambi gli approcci sono importanti: che ora si tenda verso la precisione significa che la disciplina ha raggiunto un certo livello di stabilità e maturità. Resta ancora da vedere se questa stabilità non sia destinata a essere messa a soqquadro da una nuova idea scapigliata.

ra prestazioni robuste, e gli HMM hanno dato prova di affidabilità sempre maggiore nei test rigorosi condotti negli ultimi anni. Il riconoscimento vocale, così come il campo a esso attiguo relativo alla scrittura manuale, sta già compiendo la transizione verso un uso quotidiano da parte dell'industria e degli utenti privati.

Anche le reti neurali hanno seguito un'evoluzione simile: gran parte del lavoro svolto negli anni '80 è stato dedicato a investigare cosa poteva essere fatto e imparare in che modo le reti neurali differiscono dalle tecniche "tradizionali". Usando una metodologia più robusta e avvalendosi di infrastrutture teoriche, il settore è giunto a un grado di maturazione per cui le reti neurali oggi possono essere confrontate con tecniche analoghe nel campo della statistica, del riconoscimento di pattern e dell'apprendimento automatico, in modo da applicare a ogni problema la tecnica più promettente. In seguito a questi sviluppi è nata una nuova, vigorosa industria che si appoggia alla tecnologia denominata **data mining**.

In seguito a un ritorno di interesse riassunto dall'articolo di Peter Cheeseman (1985) "In Defense of Probability", il libro *Probabilistic Reasoning in Intelligent Systems*, di Judea Pearl (1988), portò a una nuova considerazione della probabilità e della teoria delle decisioni nell'IA. Il formalismo delle reti Bayesiane fu inventato per permettere di rappresentare la conoscenza incerta in modo efficiente e ragionare intorno a essa con rigore. Quest'approccio supera gran parte dei problemi che affliggevano i sistemi di ragionamento probabilistico negli anni '60 e '70 e oggi domina la ricerca sul ragionamento incerto e sui sistemi esperti: permettendo di apprendere dall'esperienza, riunisce il meglio dell'IA classica e delle reti neurali. I lavori di Judea Pearl (1982a) e di Eric Horvitz e David Heckerman (Horvitz e Heckerman, 1986; Horvitz et al., 1986) promossero l'idea dei sistemi esperti normativi, capaci di agire razionalmente secondo le leggi della teoria delle decisioni e non cercando di imitare i passi logici degli esperti umani. Il sistema operativo WindowsTM include diversi sistemi esperti normativi di diagnostica per la risoluzione dei problemi.

Analoghe "rivoluzioni gentili" si sono verificate nel campo della robotica, della visione e della rappresentazione della conoscenza. Una miglior comprensione dei problemi e della loro complessità, unita al raffinamento degli strumenti matematici, ha portato al rafforzamento della ricerca e allo sviluppo di metodi robusti. In molti casi, la formalizzazione e la specializzazione hanno anche portato a una frammentazione: argomenti come la visione e la robotica sono sempre più isolati dall'IA "principale". Questi campi disparati possono essere riunificati dalla visione comune dell'intelligenza artificiale come attività di progettazione di agenti razionali.

La comparsa degli agenti intelligenti (1995–presente)

Forse incoraggiati dai successi ottenuti nella risoluzione di problemi ristretti, i ricercatori di intelligenza artificiale hanno ricominciato a considerare il problema degli agenti nella loro interezza. Il lavoro di Allen Newell, John Laird e Paul Rosenbloom su SOAR (Newell, 1990; Laird et al., 1987) è l'esempio meglio conosciuto di architettura completa per un agente. Il cosiddetto "movimento situato" (*situated movement*) punta alla comprensione del funzionamento di agenti situati in ambienti reali con input sensori continui. Uno degli ambienti più importanti per gli agenti intelligenti è Internet; i sistemi di IA sono diventati così comuni nelle applicazioni web che il suffisso “-bot” è entrato nel linguaggio di tutti i giorni. Inoltre, le tecnologie proprie dell'IA sono alla base di molti strumenti come i motori di ricerca, i sistemi di personalizzazione commerciale (*recommender systems*) e quelli per la costruzione di siti Web.

Oltre alla prima edizione di questo libro (Russell e Norvig, 1995), anche altri testi recenti hanno adottato la prospettiva degli agenti (Poole et al., 1998; Nilsson, 1998). Una conseguenza del tentativo di costruire agenti completi è la comprensione che potrebbe essere necessario riorganizzare in qualche modo i settori dell'IA precedentemente isolati, in modo da rendere possibile l'integrazione dei diversi risultati. In particolare, oggi è opinione diffusa che i sistemi sensori (visione, sonar, riconoscimento vocale etc.) non possano fornire informazioni sull'ambiente totalmente affidabili. Di conseguenza, i sistemi di ragionamento e pianificazione devono essere in grado di gestire l'incertezza. Una seconda importante conseguenza della prospettiva basata sugli agenti è l'avvicinamento dell'IA alle altre discipline che li trattano, come la teoria del controllo o l'economia.

Lo stato dell'arte

Cosa può fare oggi l'IA? È difficile dare una risposta sintetica, dato che le attività sono così tante, e suddivise in molti campi. Ecco una serie di esempi applicativi; nel resto del libro ne incontreremo altri.

Pianificazione e scheduling autonomi: a centocinquanta milioni di chilometri dalla Terra, il programma Remote Agent della NASA divenne il primo sistema di pianificazione integrato autonomo per la gestione dello scheduling delle operazioni di un veicolo spaziale (Jonsson et al., 2000). Remote Agent generava i piani partendo da obiettivi di alto livello inviati da terra e monitorava le operazioni del veicolo spaziale durante la loro esecuzione, rilevando, diagnosticando e recuperando dagli errori non appena si verificavano.

Giochi: Deep Blue, sviluppato da IBM, è stato il primo programma per computer capace di sconfiggere il campione del mondo battendo Garry Kasparov con un punteggio di 3.5 a 2.5 (Goodman e Keene, 1997). Kasparov dichiarò che, dal-

l'altra parte della scacchiera, aveva percepito "un nuovo tipo di intelligenza". Il settimanale *Newsweek* descrisse il match come "l'ultima resistenza del cervello umano". Il valore delle azioni IBM aumentò di 18 miliardi di dollari.

Controllo autonomo: il sistema di visione robotica ALVINN fu addestrato a guidare una macchina in modo che rimanesse sempre nella sua corsia. Montato sul furgoncino NAVLAB della CMU, un veicolo controllato dal computer, poté girare per gli Stati Uniti per 2850 miglia mantenendo il controllo dello sterzo per il 98% del tempo. Un essere umano prese la guida per il restante 2%, più che altro sulle rampe di entrata/uscita delle autostrade. NAVLAB ha delle telecamere che trasmettono immagini della strada a ALVINN, che calcola la miglior angolazione di sterzo in base all'esperienza accumulata con il precedente addestramento.

Diagnosi: in molte aree della medicina, i programmi di diagnosi medica basati sull'analisi probabilistica si sono dimostrati capaci di performance pari a quelle di un medico esperto. Heckerman (1991) descrive un caso in cui un famoso esperto di patologie dei linfonodi aveva manifestato scetticismo sulla diagnosi fatta da un programma riguardo un caso particolarmente difficile. I creatori del programma lo hanno invitato a chiedere al computer di giustificare la diagnosi: la macchina ha spiegato gli aspetti principali che avevano influenzato la sua decisione, evidenziando la sottile interazione di molti dei sintomi osservati. Alla fine, l'esperto ha dovuto dichiararsi d'accordo col programma.

Pianificazione logistica: durante la Guerra del Golfo del 1991, le forze armate americane hanno utilizzato DART, Dynamic Analysis e Replanning Tool (Cross e Walker, 1994), per automatizzare la pianificazione logistica e i trasporti. Il problema comprendeva la gestione contemporanea di persone, approvvigionamenti e 50.000 veicoli, e tra tutti gli altri parametri doveva prendere in considerazione i punti di partenza, quelli di arrivo, i collegamenti esistenti e la risoluzione dei conflitti. Le tecniche di pianificazione dell'IA permisero la generazione, nel giro di poche ore, di un piano che con i metodi tradizionali avrebbe richiesto settimane. La DARPA (*Defense Advanced Research Project Agency*) affermò che questa singola applicazione aveva già più che ripagato i suoi investimenti trentennali nel campo dell'IA.

Robotica: oggi molti microchirurghi si avvalgono dell'assistenza di robot per operare. HipNav (DiGioia et al., 1996) applica delle tecniche di visione robotica per creare un modello tridimensionale dell'anatomia interna di un paziente e poi usa un controllo robotico per guidare l'inserimento di una protesi all'anca.

Comprensione del linguaggio e risoluzione di problemi: PROVERB (Littman et al., 1999) è un programma per computer che risolve i cruciverba meglio della gran parte degli esseri umani usando vincoli sulle parole, un grande database di schemi risolti e molte altre fonti, tra cui dizionari e database online come quelli che riportano tutti i film con i rispettivi attori. Il programma, ad esempio, è in

grado di risolvere la definizione “una storia a Nizza” con “ETAGE”, perché il suo database include la coppia definizione/soluzione “storia in Francia/ETAGE” e il programma riconosce che i pattern “X a Nizza” e “X in Francia” hanno spesso la stessa soluzione. Il programma non sa che Nizza è una città della Francia, ma può lo stesso risolvere il cruciverba.

Questi sono solo alcuni esempi di sistemi di intelligenza artificiale oggi esistenti. Non si tratta di magia né di fantascienza bensì di scienza, ingegneria e matematica, a cui questo libro vuole fornire un’introduzione.

1.4 Riepilogo

Questo capitolo fornisce una definizione dell’IA e descrive il background culturale in cui si è sviluppata. Questi sono alcuni dei punti più importanti.

- ❖ Persone differenti hanno concetti diversi dell’IA. Due domande importanti sono: vi interessa più il pensiero o il comportamento? E anche, volete usare come modello gli esseri umani o fare riferimento a uno standard ideale?
- ❖ In questo libro adottiamo l’idea che l’intelligenza riguardi principalmente l’azione razionale. Idealmente, un agente intelligente intraprende in ogni situazione la migliore azione possibile. Studieremo il problema di costruire agenti che sono “intelligenti” secondo questa particolare accezione.
- ❖ I filosofi, a partire da 400 a.C., hanno reso concepibile lo sviluppo dell’IA proponendo che la mente sia per certi aspetti simile a una macchina, che funzioni sulla base di conoscenze rappresentate in qualche forma di linguaggio interno e che si possa usare il pensiero per scegliere quale azione compiere.
- ❖ I matematici hanno fornito gli strumenti per manipolare proposizioni dotate di significato logico sia in condizioni di certezza che di incertezza. Inoltre hanno sviluppato la teoria necessaria a comprendere la computazione e analizzare gli algoritmi.
- ❖ Gli economisti hanno formalizzato il problema di prendere decisioni che massimizzino il risultato atteso dal decisore.
- ❖ Gli psicologi hanno adottato l’idea che gli umani e gli animali possano essere considerati macchine che elaborano informazioni. I linguisti hanno mostrato che l’uso del linguaggio si adatta a questo modello.
- ❖ Gli ingegneri informatici hanno creato gli artefatti che rendono possibile applicare l’IA. I programmi di IA tendono a essere complessi, e non potrebbero funzionare senza i grandi passi avanti dei computer in termini di velocità di elaborazione e quantità di memoria.

- ◆ La teoria del controllo si occupa della progettazione di dispositivi che agiscono in modo ottimo sulla base della retroazione fornita dall'ambiente. All'inizio gli strumenti matematici della teoria del controllo differivano molto da quelli propri dell'IA, ma ora i due campi si stanno avvicinando.
- ◆ La storia dell'IA è stata segnata da cicli in cui al successo ha fatto seguito un eccessivo ottimismo, seguito da cadute d'entusiasmo e tagli di fondi. Ci sono anche stati cicli in cui l'introduzione di nuovi approcci creativi si è alternata con il raffinamento e la sistematizzazione dei migliori tra essi.
- ◆ I passi avanti dell'IA nell'ultimo decennio sono stati favoriti dall'uso diffuso del metodo scientifico, che ha guidato la sperimentazione e il confronto tra approcci alternativi.
- ◆ I recenti progressi nella comprensione delle basi teoriche dell'intelligenza sono andati di pari passo con l'aumento delle capacità dei sistemi esistenti. Settori diversi dell'IA sono arrivati a un'integrazione e l'IA stessa ha trovato un terreno comune con altre discipline.

Note storiche e bibliografiche

Lo stato metodologico dell'intelligenza artificiale è discusso in *The Sciences of the Artificial*, di Herb Simon (1981), che esamina le aree di ricerca che riguardano gli artefatti complessi. Simon spiega come l'IA possa essere considerata contemporaneamente come scienza e matematica. Cohen (1995) fornisce una panoramica della metodologia sperimentale all'interno dell'IA. Ford e Hayes (1995) offrono una visione personale sull'utilità del test di Turing.

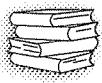
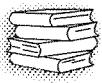
Artificial Intelligence: The Very Idea, di John Haugeland (1985), è un resoconto di piacevole lettura sui problemi filosofici e pratici dell'IA. Le scienze cognitive sono ben descritte da molti testi recenti (Johnson-Laird, 1988; Stillings et al., 1995; Thagard, 1996) e dall'*Encyclopedia of the Cognitive Sciences* (Wilson e Keil, 1999). Baker (1989) copre la parte sintattica della linguistica moderna, mentre Chierchia e McConnell-Ginet (1990) la semantica. Jurafsky e Martin (2000) trattano l'argomento della linguistica computazionale.

I primi anni di sviluppo dell'IA sono descritti in *Computers and Thought* di Feigenbaum e Feldman (1963), in *Semantic Information Processing* di Minsky (1968) e nella serie *Machine Intelligence* curata da Donald Michie. Un gran numero di articoli importanti sono stati raccolti in antologia da Webber e Nilsson (1981) e da Luger (1995). I primi lavori sulle reti neurali sono raccolti in *Neurocomputing* (Anderson e Rosenfeld, 1988). La *Encyclopedia of AI* (Shapiro, 1992) contiene articoli generali su quasi ogni argomento dell'IA e fornisce un buon corredo bibliografico per chi volesse approfondire la relativa letteratura scientifica.

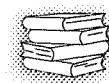
I lavori più recenti sono raccolti negli atti dei congressi più importanti: la *International Joint Conference on AI* (IJCAI) è biennale, la *European Conference on AI* (ECAI) è annuale, e c'è poi la *National Conference on AI*, più nota come AAAI dall'acronimo dell'organizzazione che la promuove. Le riviste più importanti per l'IA in generale sono *Artificial Intelligence*, *Computational Intelligence*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *IEEE Intelligent Systems* e, in formato elettronico, il *Journal of Artificial Intelligence Research*. Ci sono anche molti congressi e riviste dedicate a settori specifici, che citeremo nei capitoli appropriati. Le organizzazioni più importanti sono la American Association for Artificial Intelligence (AAAI), l'ACM Special Interest Group in Artificial Intelligence (SIGART) e la Society for Artificial Intelligence and Simulation of Behaviour (AISB). *AI Magazine*, pubblicato dalla AAAI, contiene molti articoli specializzati e tutorial; il suo sito web, aaai.org, fornisce notizie aggiornate e varie informazioni.

Esercizi

Lo scopo di questi esercizi è stimolare la discussione: alcuni potrebbero anche essere assegnati agli studenti come progetti più articolati. Alternativamente si potrebbe cercare di dare ora una risposta preliminare, e riprendere gli esercizi alla fine del corso o della lettura del libro.

- 
- 
- 1.1 Definite con parole vostre: (a) intelligenza, (b) intelligenza artificiale, (c) agente.
 - 1.2 Leggete il lavoro originale di Turing sull'IA (Turing, 1950). Nell'articolo Turing considera diverse possibili obiezioni alla sua proposta e al test per l'intelligenza. Quali obiezioni sono ancora valide oggi? Le confutazioni di Turing sono valide? Potete pensare a nuove obiezioni, emerse grazie agli sviluppi successivi alla scrittura dell'articolo? Turing prediceva che, nell'anno 2000, un computer avrebbe avuto una possibilità del 30% di passare un test di Turing della durata di cinque minuti con un esaminatore inesperto. Quali possibilità pensate che potrebbe avere oggi un computer? E tra cinquant'anni?
 - 1.3 Ogni anno viene assegnato il premio Loebner al programma che arriva più vicino a superare una versione del test di Turing. Fate una ricerca sugli ultimi vincitori del premio Loebner: che tecniche hanno usato? In che modo queste ricerche hanno fatto progredire l'IA?
 - 1.4 Ci sono classi ben note di problemi che per la loro difficoltà risultano intrattabili per i computer, mentre per altre si è dimostrata la non decidibilità. Questo significa che l'IA è impossibile?

- 1.5 Supponiamo di estendere il programma ANALOGY di Evans in modo che riesca a ottenere 200 punti in un test standard per il Q.I. Questo significherebbe aver creato un programma che è più intelligente di un essere umano? Spiegate le vostre ragioni.
- 1.6 In che modo può essere inaccurata l'introspezione, il cui scopo è riportare i propri pensieri interni? Possiamo sbagliarci su quello che stiamo pensando? Discutete.
- 1.7 Prendete in esame la letteratura riguardante l'IA e scoprite se le seguenti attività possono essere presentemente svolte dai computer:
- giocare decentemente a ping-pong
 - guidare in centro nella città del Cairo
 - fare la spesa settimanale al mercato
 - fare la spesa settimanale sul Web
 - giocare a bridge a un buon livello
 - scoprire e dimostrare nuovi teoremi matematici
 - scrivere una storiella intenzionalmente divertente
 - fornire consigli legali competenti in una branca specializzata del diritto
 - tradurre inglese parlato in svedese parlato, in tempo reale
 - eseguire una difficile operazione chirurgica.



Per le attività oggi impossibili cercate di identificare quali sono le difficoltà e di predire se e quando saranno superate.

- 1.8 Alcuni autori hanno sostenuto che le percezioni e le abilità motorie costituiscono la parte più importante dell'intelligenza, e che le capacità "di alto livello" sono in qualche modo parassite: delle semplici aggiunte alle abilità di base. Certo, la maggior parte dell'evoluzione e una gran parte del cervello sono state dedicate alla percezione e alle abilità motorie, mentre l'IA ha trovato più facile, sotto molti aspetti, occuparsi di attività come il gioco degli scacchi e l'inferenza logica piuttosto che di percezione e azione nel mondo reale. Ritenete che la tradizionale attenzione dell'IA sulle abilità cognitive di livello più alto sia mal riposta?
- 1.9 Perché l'evoluzione dovrebbe tendere a produrre sistemi che agiscono razionalmente? Quali sono gli obiettivi per cui sono progettati sistemi siffatti?
- 1.10 I riflessi automatici (come togliere la mano da una padella calda) sono razionali? Sono intelligenti?
- 1.11 "Sicuramente i computer non possono essere intelligenti: infatti possono fare solo quello che dice loro il programmatore". Secondo voi la seconda parte della frase è vera, e pensate che implichia la prima parte?

- 1.12 “Sicuramente gli animali non possono essere intelligenti: infatti possono fare solo quello che gli dicono i geni”. Secondo voi la seconda parte della frase è vera, e pensate che implichi la prima parte?
- 1.13 “Sicuramente gli animali, gli esseri umani e i computer non possono essere intelligenti: infatti sono tutti composti di atomi e possono fare solo quello che gli dicono le leggi della fisica”. Secondo voi la seconda parte della frase è vera, e pensate che implichi la prima parte?

Capitolo 2

Agenti intelligenti

Nel quale discutiamo la natura degli agenti, perfetti o no, la diversità degli ambienti, e il risultante assortimento di tipi di agenti.

Nel Capitolo 1 abbiamo introdotto il concetto di **agente razionale**, fondamentale per il nostro approccio all'intelligenza artificiale. In questo capitolo lo definiremo più concretamente. Come vedremo, il concetto di razionalità si può applicare a una grande varietà di agenti che operano in ogni ambiente immaginabile. Lo scopo di questo libro è applicare tale nozione per sviluppare un piccolo insieme di principî di progettazione per la costruzione di agenti di successo, che possano chiamarsi a buon diritto **intelligenti**.

Cominceremo con l'esaminare gli agenti, gli ambienti e le relazioni tra essi. L'osservazione che alcuni agenti si comportano meglio di altri porta naturalmente alla formulazione dell'idea di agente razionale: quello che si comporta nel modo migliore. Quanto bene si può comportare un agente dipende dalla natura dell'ambiente; alcuni presentano più difficoltà di altri. Forniremo una semplice suddivisione degli ambienti in categorie e mostreremo come le rispettive caratteristiche influenzano il progetto di agenti adeguati. Descriveremo anche un certo numero di semplici progetti schematici di agente che saranno sviluppati ulteriormente nel resto del libro.

ambiente
sensori
attuatori

percezione
sequenza percettiva



funzione agente

programma agente

2.1 Agenti e ambienti

Un agente è qualsiasi cosa possa essere vista come un sistema che percepisce il suo ambiente attraverso dei sensori e agisce su di esso mediante attuatori. Questa semplice idea è illustrata nella Figura 2.1. Un agente umano possiede come sensori occhi, orecchie e altri organi e può usare come attuatori mani, gambe, bocca e altre parti del corpo. Un agente robotico potrebbe avere telecamere e dispositivi a infrarossi per sensori e diversi motori per attuatori. Un agente software riceve come input sensori le battute dei tasti, il contenuto dei file e dei pacchetti di rete e può intervenire sull'ambiente cambiando la visualizzazione su uno schermo, scrivendo file e inviando pacchetti di dati. In generale presumeremo che ogni agente possa percepire le sue stesse azioni (ma non sempre i relativi effetti).

Useremo il termine **percezione** (*percept*) per indicare gli input percettivi dell'agente in un dato istante. La **sequenza percettiva** (*percept sequence*) è la storia completa di tutto ciò che l'agente ha percepito nella sua esistenza. In generale, *la scelta dell'azione di un agente in un qualsiasi istante può dipendere dall'intera sequenza percettiva osservata fino a quel momento*. Se possiamo specificare l'azione prescelta dall'agente per ogni possibile sequenza percettiva, allora abbiamo descritto l'agente in modo completo. In termini matematici diciamo quindi che il comportamento di un agente è descritto dalla **funzione agente**, che descrive la corrispondenza tra una qualsiasi sequenza percettiva e una specifica azione.

Possiamo immaginare di realizzare *in forma di tabella* la funzione agente che descrive un certo agente. Nella maggior parte dei casi questa tabella sarebbe molto grande, di fatto infinita, a meno di non specificare una lunghezza massima delle sequenze percettive che vogliamo prendere in considerazione. Volendo analizzare un agente, in via di principio è possibile ricostruire la sua tabella provando tutte le possibili sequenze percettive e registrando l'azione prescelta dall'agente come risposta.¹ La tabella, naturalmente, è una definizione *esterna* dell'agente. Internamente, la funzione di un agente artificiale sarà implementata da un **programma agente**. È importante tenere questi due concetti ben distinti: la funzione è una descrizione matematica astratta; il programma è una sua implementazione concreta, in esecuzione sull'architettura dell'agente.

¹ Se l'agente applica un certo grado di casualità nella scelta delle azioni, potremmo dover provare ogni sequenza più volte per ricostruire la probabilità di ogni scelta. Si potrebbe pensare che agire casualmente sia alquanto stupido, ma vedremo più avanti in questo capitolo che può essere un comportamento molto intelligente.

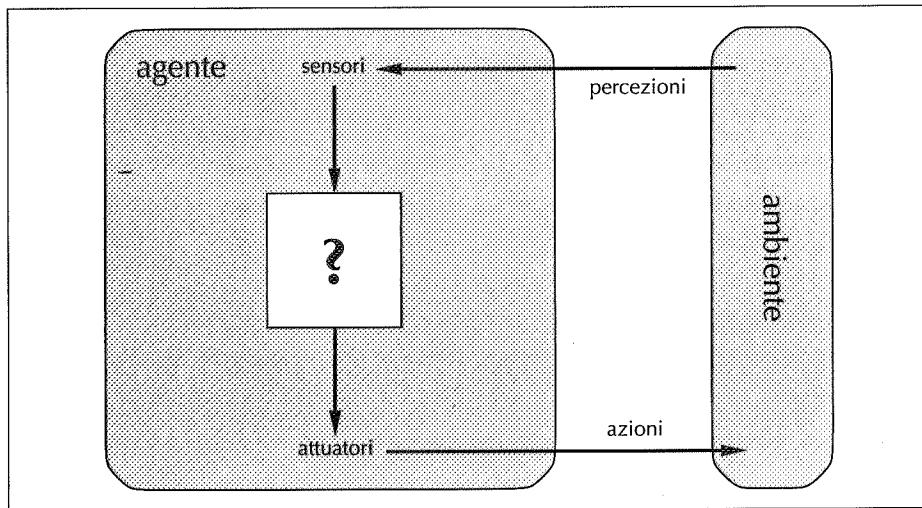


Figura 2.1
Gli agenti interagiscono con l'ambiente attraverso sensori e attuatori.

Per illustrare queste idee faremo ricorso a un esempio molto semplice: il mondo dell'aspirapolvere mostrato nella Figura 2.2. Questo mondo è così semplice che possiamo descrivere tutto quello che succede; è anche totalmente inventato, per cui possiamo escogitare molte variazioni. Ci sono solo due posizioni: i riquadri *A* e *B*. L'agente aspirapolvere percepisce in quale riquadro si trova e se c'è dello sporco in tale locazione. Può scegliere di muoversi a sinistra, a destra, aspirare lo sporco o non fare nulla. Una funzione agente molto semplice è “se il riquadro corrente è sporco, aspira, altrimenti muoviti nell'altro riquadro”. La Figura 2.3 ne mostra una tabulazione parziale: forniremo più avanti in questo capitolo, nella Figura 2.8, un programma agente altrettanto semplice che la implementa.

Guardando la Figura 2.3, vediamo che si possono definire vari agenti del mondo aspirapolvere semplicemente riempiendo la colonna di destra in modi diversi. La domanda che sorge naturalmente, allora, è questa: qual è il modo più cor-

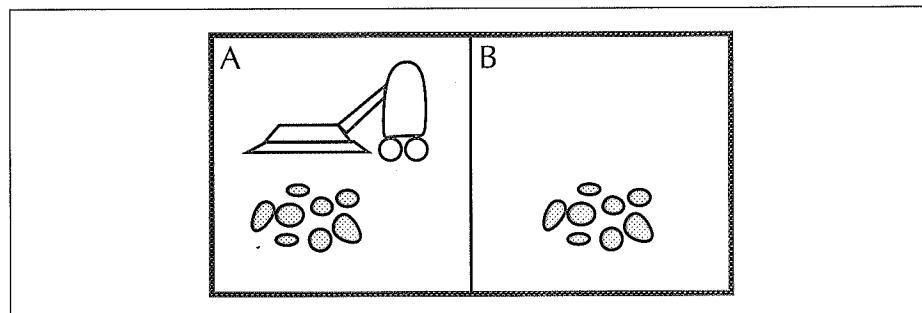


Figura 2.2
Un mondo dell'aspirapolvere con due sole posizioni.

Figura 2.3
 Tabulazione
 parziale di una
 semplice funzione
 agente per il
 mondo
 dell'aspirapolvere
 mostrato nella
 Figura 2.2.

Sequenza percettiva	Azione
[A, Pulito]	Destra
[A, Sporco]	Aspira
[B, Pulito]	Sinistra
[B, Sporco]	Aspira
[A, Pulito], [A, Pulito]	Destra
[A, Pulito], [A, Sporco]	Aspira
:	:
[A, Pulito], [A, Pulito], [A, Pulito]	Destra
[A, Pulito], [A, Pulito], [A, Sporco]	Aspira
:	:

retto di progettare la tabella? In altre parole, che cosa rende un agente buono o cattivo, intelligente o stupido? Risponderemo a queste domande nel prossimo paragrafo.

Prima di concludere vogliamo porre l'accento sul fatto che il concetto di agente è solo uno strumento adatto all'analisi di sistemi, e non una caratterizzazione assoluta che divide il mondo in agenti e non-agenti. Si potrebbe anche dire che una calcolatrice portatile è un agente che decide di intraprendere l'azione di visualizzare “4” sul display quando gli viene fornita la sequenza percettiva “2 + 2 =”, ma una siffatta analisi servirebbe ben poco a comprendere il suo funzionamento.

2.2 Comportarsi correttamente: il concetto di razionalità

agente razionale

Un agente razionale è un agente che fa la cosa giusta: dal punto di vista teorico, si potrebbe dire che ogni riga nella tabella della funzione d'agente è scritta correttamente. È ovvio che è meglio fare la cosa giusta di quella sbagliata, ma che cosa significa precisamente? In prima approssimazione diremo che l'azione giusta è quella che fa sì che l'agente ottenga il massimo grado di successo. Di conseguenza, avremo bisogno di qualche tecnica per misurare il successo: quest'ultima, insieme alle descrizioni dell'ambiente, dei sensori e degli attuatori dell'agente, fornirà la specifica completa dell'attività che l'agente è chiamato a svolgere. A questo punto potremo definire più precisamente che cosa significa essere razionali.

Misure di prestazione

Una misura di prestazione (*performance measure*) rappresenta il criterio in base al quale valutare il successo del comportamento di un agente. Quando si lascia libero un agente in un dato ambiente, esso genererà una sequenza di azioni in base alle percezioni ricevute. Questa sequenza di azioni farà sì che l'ambiente attraversi una sequenza di stati: se tale sequenza è desiderabile, allora l'agente si è comportato bene. Naturalmente, non c'è una misura fissa applicabile a tutti gli agenti. Potremmo chiedere all'agente stesso un'opinione soggettiva di quanto sia contento della propria prestazione, ma alcuni agenti non sarebbero in grado di rispondere, e altri potrebbero illudersi.² Noi ricorreremo quindi a una misura di prestazione oggettiva, tipicamente stabilita dal progettista dell'agente stesso.

misura di prestazione

Consideriamo ancora l'agente aspirapolvere del paragrafo precedente. Come misura potremmo proporre la quantità di sporco aspirato in un singolo lasso di tempo di otto ore. Nel caso di un agente razionale, naturalmente, si ottiene sempre quello che si chiede: l'aspirapolvere potrebbe massimizzare la misura pulendo un po' di sporco, quindi sbattendolo nuovamente sul pavimento per poi aspirarlo ancora e così via. Una misura di prestazione più adeguata farebbe riferimento al grado di pulizia del pavimento: ad esempio, si potrebbe assegnare un punto per ogni riquadro di pavimento pulito in ogni passo temporale (magari assegnando una penalità per il consumo di elettricità e il rumore generato). Come regola generale, è meglio progettare le misure di prestazione in base all'effetto che si desidera ottenere sull'ambiente piuttosto che su come si pensa che debba comportarsi l'agente.

Scegliere una misura di prestazione non è sempre una cosa facile: ad esempio, il concetto di "pulizia del pavimento" che abbiamo espresso nel paragrafo precedente è basato sulla pulizia media in un lasso di tempo. Eppure la stessa pulizia media può essere ottenuta da due agenti ben diversi, uno che pulisce in modo mediocre ma costante, l'altro che pulisce con energia prendendosi però delle lunghe pause. Quale delle due soluzioni sia preferibile può sembrare un problema che riguarda solo bidelli e portinai, ma è effettivamente un profondo dilemma filosofico con vaste implicazioni. È meglio una vita smodata fatta di altissimi picchi e profonde cadute, o una sicura ma monotona? È meglio un'economia in cui tutti vivono in moderata povertà, o una in cui alcuni sono ricchissimi e altri molto poveri? Lasceremo le risposte a queste domande come esercizio per i lettori più diligenti.



² Gli agenti umani in particolare sono noti per il loro comportamento che ricorda la famosa volpe con l'uva: dopo che non hanno ottenuto qualcosa si convincono di non averla desiderata, come in "Oh beh, pace, dopotutto non è che desiderassi tanto quello stupido premio Nobel".

Razionalità

In un dato momento, ciò che è razionale dipende da quattro fattori:

- ♦ la misura di prestazione che definisce il criterio del successo
- ♦ la conoscenza pregressa dell'ambiente da parte dell'agente
- ♦ le azioni che l'agente può effettuare
- ♦ la sequenza percettiva dell'agente fino all'istante corrente.

Questo porta alla seguente definizione di agente razionale:

per ogni possibile sequenza di percezioni, un agente razionale dovrebbe scegliere un'azione che massimizzi la sua misura di prestazione attesa, date le informazioni fornite dalla sequenza percettiva e da ogni ulteriore conoscenza dell'agente.

Consideriamo il semplice agente aspirapolvere che pulisce un riquadro se questo è sporco, muovendosi nell'altro in caso contrario; la sua tabella è rappresentata nella Figura 2.3. Quest'agente è razionale? Dipende! Dobbiamo prima di tutto dire qual è la sua misura di prestazione, che cosa sa dell'ambiente, quali sensori e attuatori possiede. Supponiamo che:

- ♦ la misura di prestazione assegna un punto per ogni riquadro pulito all'inizio di ogni passo temporale, per una "vita" dell'agente di 1000 passi;
- ♦ la "geografia" dell'ambiente sia nota *a priori* (Figura 2.2) ma non la posizione iniziale dell'agente, né la distribuzione dello sporco. I riquadri puliti rimangono tali, mentre l'aspirazione dello sporco pulisce il riquadro corrente. Le azioni *Sinistra* e *Destra* muovono l'agente nelle corrispondenti direzioni, tranne quando ciò porterebbe l'agente fuori dall'ambiente, nel qual caso l'agente non si muove;
- ♦ le sole azioni disponibili sono *Sinistra*, *Destra*, *Aspira* e *NoOp* (non fare nulla);
- ♦ l'agente percepisce correttamente la propria posizione e se il riquadro corrente è sporco o meno.

Noi sosteniamo che *date queste condizioni* l'agente è effettivamente razionale; le sue prestazioni sono buone almeno quante quelle di qualsiasi altro agente. L'Esercizio 2.4 vi chiederà di darne una dimostrazione.

Si vede chiaramente che lo stesso agente, date altre condizioni, non sarebbe più razionale. Ad esempio, una volta pulito tutto lo sporco quest'agente continua a muoversi avanti e indietro; se la misura di prestazione prevedesse una penalità di un punto per ogni movimento, l'agente se la caverebbe maluccio. In questo caso sarebbe meglio non fare nulla, una volta assicurarsi che tutti i riquadri siano puliti. D'altra parte se le locazioni pulite potessero sporcarsi nuovamente l'agente dovrebbe controllare ogni tanto, e nel caso ripulire ove necessario. Se la geografia del-

l'ambiente fosse sconosciuta, l'agente dovrebbe esplorarlo invece di rimanere per sempre nei riquadri *A* e *B*. L'Esercizio 2.4 vi chiederà di progettare agenti adatti a tutti questi casi.

Onniscienza, apprendimento e autonomia

Dobbiamo distinguere accuratamente il concetto di razionalità e quello di onniscienza. Un agente onnisciente conosce il risultato effettivo delle sue azioni e può agire di conseguenza; ma nella realtà l'onniscienza è impossibile. Consideriamo il seguente esempio: sto camminando un bel giorno lungo gli Champs Elysées e vedo un vecchio amico dall'altra parte della strada. Non ci sono macchine vicino e non ho nulla da fare per cui, razionalmente, comincio ad attraversare la strada. Nel frattempo, a undicimila metri di quota, il portellone di un aereo si stacca³ e vengo spiaccicato prima di arrivare dall'altra parte della strada. È stato irrazionale, da parte mia, decidere di attraversare? È molto improbabile che il mio necrologio si intitoli "Un idiota cerca di attraversare i Campi Elisi".

Quest'esempio dimostra che razionalità non significa perfezione. Infatti la razionalità massimizza il risultato atteso, mentre la perfezione massimizza quello reale. Non richiedere prestazioni perfette non è solo una questione di generosità verso gli agenti: per pretendere che un agente faccia quella che solo a posteriori si può rivelare la migliore azione dovremmo utilizzare sfere di cristallo o macchine del tempo.

La nostra definizione di razionalità non richiede quindi l'onniscienza, perché la scelta razionale dipende solo dalla sequenza percettiva fino al momento corrente. Dobbiamo anche assicurarci di non aver inavvertitamente permesso all'agente di intraprendere con risolutezza attività poco intelligenti. Ad esempio, se un agente non guarda in entrambe le direzioni prima di attraversare una strada trafficata, la sua sequenza percettiva non gli dirà che c'è un TIR che si avvicina a gran velocità. Date queste premesse, secondo la nostra definizione sarebbe razionale attraversare la strada in questo momento? Nient'affatto! Prima di tutto, con una sequenza percettiva così scarsa di informazioni il rischio di incidenti sarebbe troppo grande. In secondo luogo, un agente razionale dovrebbe scegliere l'azione "guarda" prima di incominciare l'attraversamento, perché così facendo potrebbe massimizzare la prestazione attesa. Intraprendere azioni mirate a modificare le percezioni future – ciò che viene talvolta chiamato, con un termine inglese, information gathering – è una parte importante della razionalità, che tratteremo nel dettaglio nel Capito-

onniscienza

information gathering

³ Cfr. N. Henderson, "Urgente necessità di nuove chiusure a scatto per i portelli dei jumbo jet Boeing 747" *Washington Post*, 24 agosto 1989.

esplorazione

imparare

autonomia

lo 16. Un secondo esempio di information gathering è rappresentato dall'esplorazione che dev'essere intrapresa da un agente aspirapolvere posto in un ambiente inizialmente sconosciuto.

La nostra definizione richiede che un agente razionale non si limiti a raccogliere informazioni, ma sia anche in grado di imparare il più possibile sulla base delle proprie percezioni. La sua configurazione iniziale potrebbe riflettere una conoscenza pregressa, che però può modificarsi e aumentare man mano che l'agente accumula esperienza. Ci sono casi limite nei quali l'ambiente è completamente conosciuto a priori: in tali situazioni l'agente non ha bisogno di percepire o imparare; può semplicemente agire nel modo corretto. Naturalmente, il comportamento di agenti simili è molto fragile. Consideriamo l'umile scarabeo stercoario: dopo aver scavato la sua tana e aver deposto le uova, l'insetto recupera una pallina di sterco dalle vicinanze per bloccarne l'ingresso. Se la palla di sterco viene rimossa dalle sue zampe durante il tragitto, lo scarabeo continua a camminare e compie gli stessi gesti come se stesse tappando la tana con la palla inesistente, non accorgendosi mai della sua scomparsa. L'evoluzione ha selezionato il comportamento dello scarabeo basandosi su un assunto, e quando questo è violato, ne consegue un comportamento erroneo. La vespa sphex è leggermente più intelligente. Le femmine scavano una tana, quindi escono e vanno a pungere un bruco che trascinano fino alla tana stessa. Una volta giunte entrano nuovamente per controllare che tutto sia a posto, quindi trascinano dentro il bruco e finalmente depongono le uova. Fin qui tutto bene, ma se un entomologo sposta il bruco qualche centimetro più in là mentre la vespa sta facendo il controllo, questa ritornerà indietro fino al passo "trascina il bruco" del suo piano, dopodiché lo riprenderà senza modifiche, anche dopo che il bruco è stato spostato dozzine di volte. La vespa sphex non riesce a capire che il suo piano istintivo sta fallendo, e così non può cambiarlo.

Gli agenti di successo suddividono il calcolo della funzione agente in tre periodi differenti: al momento della progettazione dell'agente, una parte dei calcoli viene svolta dai suoi programmatore; quando l'agente sta decidendo la sua prossima azione, svolge altri calcoli; imparando dall'esperienza, infine, compie ulteriori elaborazioni per decidere come modificare il suo comportamento.

Quando un agente si appoggia alla conoscenza pregressa inserita dal programmatore invece che sulle proprie percezioni, diciamo che l'agente manca di autonomia. Un agente razionale dovrebbe essere autonomo, e imparare il più possibile per compensare la presenza di conoscenza parziale o erronea. Ad esempio, l'aspirapolvere che impara a prevedere quando e dove apparirà il nuovo sporco si comporterà meglio degli altri. Nella pratica, raramente si richiede che un agente sia completamente autonomo fin dall'inizio: finché la sua esperienza è limitata o nulla un simile agente sarebbe costretto ad agire casualmente, a meno di non ricevere aiuto dal suo progettista. Così, proprio come l'evoluzione ha fornito agli animali riflessi innati per sopravvivere abbastanza a lungo da imparare come comportarsi, è ragionevole fornire a un agente intelligente artificiale, oltre all'abilità di

imparare, anche un po' di conoscenza iniziale. Dopo aver accumulato una sufficiente esperienza dell'ambiente, il comportamento dell'agente razionale può diventare a tutti gli effetti *indipendente dalla conoscenza pregressa*. Incorporare l'apprendimento nel suo progetto, quindi, permette di sviluppare un agente razionale capace di muoversi efficacemente in una grande varietà di ambienti differenti.

2.3 La natura degli ambienti

Ora che abbiamo una definizione di razionalità, siamo quasi pronti a occuparci della creazione di agenti razionali. Prima, però, dobbiamo considerare gli ambienti, che sono essenzialmente i "problemi" di cui gli agenti razionali rappresentano le "soluzioni". Cominceremo a vedere, con l'aiuto di numerosi esempi, come definire un ambiente. Mostreremo poi che ci sono molte tipologie di ambienti, che influenzano direttamente la progettazione del programma agente più appropriato.

ambienti

Specificare un ambiente

Quando abbiamo discusso la razionalità del nostro semplice agente aspirapolvere, abbiamo dovuto specificare la misura di prestazione, l'ambiente esterno e gli attuatori e i sensori dell'agente. Tutto ciò può essere raggruppato nel termine specifico **task environment**, che potremmo tradurre "ambiente operativo" o "ambiente della missione". Se vi piacciono gli acronimi, potete parlare anche di descrizione PEAS (*Performance, Environment, Actuators, Sensors*). Quando si progetta un agente, il primo passo dovrebbe sempre corrispondere alla specifica del task environment, la più ricca possibile.

PEAS

Il mondo dell'aspirapolvere era un esempio semplice. Consideriamone ora uno più complesso, che ci accompagnerà per tutto il resto del capitolo: un pilota automatico per taxi. Prima che qualche lettore si allarmi, precisiamo che un taxi completamente automatizzato è al di là delle possibilità della tecnologia esistente: per saperne di più potete leggere a pag. 38 la descrizione di un robot pilota esistente, o consultare gli atti degli ultimi congressi sui cosiddetti *Intelligent Transportation Systems*. L'attività della guida, nelle sua interezza, è estremamente aperta. Non ci sono limiti alle possibili nuove combinazioni di circostanze che si possono verificare; questa è un'altra ragione per cui abbiamo deciso di prenderla come esempio. La Figura 2.4 presenta la descrizione PEAS dell'ambiente operativo del taxi. Ne discuteremo ogni elemento nei prossimi paragrafi.

Prima di tutto a quale misura di prestazione dovrebbe aspirare il nostro pilota automatico? Gli aspetti desiderabili includono: arrivare alla destinazione corretta; minimizzare il consumo di carburante e l'usura; minimizzare la durata temporale del viaggio e/o il suo costo; minimizzare le violazioni al codice della strada e il

Figura 2.4
Descrizione PEAS
del task
environment
di un taxi
automatizzato.

tipo di agente	misura di prestazione	ambiente	attuatori	sensori
guidatore di taxi	sicuro, veloce, ligo alla legge, viaggio confortevole, profitti massimi	strada, altri veicoli nel traffico, pedoni, clienti	sterzo, acceleratore, freni, frecce, clacson, schermo di interfaccia	telecamere, sonar, tachimetro, GPS, contachilometri, accelerometro, sensori sullo stato del motore, tastiera

disturbo per gli altri guidatori; massimizzare la sicurezza e il comfort dei passeggeri; massimizzare i profitti. Alcuni di questi obiettivi sono chiaramente in contrasto, per cui occorrerà trovare una qualche forma di compromesso.

Il punto successivo è: qual è l'ambiente nel quale il taxi dovrà operare? Ogni tassista dev'essere in grado di muoversi in una varietà di strade, che vanno dai sentieri di campagna alle strade urbane, fino alle autostrade a più corsie. Le strade contengono altri veicoli, pedoni, animali randagi, lavori in corso, macchine della polizia, pozzanghere e buche nell'asfalto. Il taxi deve anche interagire con i passeggeri, potenziali e non. Ci sono anche scelte aggiuntive: il taxi potrebbe operare in California, dove c'è raramente la neve, oppure in Alaska, dove è raro che non ci sia. Potremmo accontentarci di un taxi che guida a destra, o potremmo desiderarne uno abbastanza flessibile da tenere la sinistra quando si trova in Inghilterra o Giappone. Naturalmente, restringendo l'ambiente la progettazione diventa più semplice.

Gli attuatori disponibili al pilota automatico saranno più o meno identici a quelli usati da un autista umano: acceleratore, sterzo e freno. Oltre a questi, sarà necessario uno schermo o un sistema di sintesi vocale per comunicare con i passeggeri, e forse anche qualche dispositivo per comunicare con gli altri veicoli, in modo più o meno cortese.

Per raggiungere i suoi obiettivi nell'ambiente, il taxi dovrà sapere dove si trova, cosa c'è sulla strada e quanto velocemente si sta muovendo. I suoi sensori dovranno quindi includere almeno una o più telecamere, il tachimetro e il contachilometri. Per controllare il veicolo in modo aconci, specialmente in curva, dovrebbe avere anche un accelerometro; per conoscere lo stato meccanico del veicolo sarà anche necessaria la presenza dei consueti sensori per motore e sistema elettrico. Il nostro taxi potrebbe poi disporre di strumenti che non sono disponibili all'autista medio: un sistema di posizionamento globale satellitare (GPS) che fornisca

tipo di agente	misura di prestazione	ambiente	attuatori	sensori
sistema di diagnosi medica	paziente sano, minimizzare i costi e le denunce	paziente, ospedale, staff medico	schermo per visualizzare domande, test, diagnosi, trattamenti, documentazione	inserimento da tastiera dei sintomi, dei risultati dei test, delle risposte del paziente
sistema di analisi di immagini satellitari	categorizzazione corretta della immagini	collegamento verso terra di un satellite orbitante	visualizzazione della categorizzazione della scena	array di pixel sensibili al colore
robot selezionatore di parti meccaniche	percentuale di pezzi inseriti nei contenitori giusti	nastro trasportatore con parti meccaniche; contenitori	braccio meccanico con manipolatore snodato	telecamera, sensori di posizionamento del braccio meccanico
controllore industriale per una raffineria	massimizzare la purezza, il volume della produzione, sicurezza del processo	raffineria, operai specializzati	valvole, pompe, elementi di riscaldamento, visori	temperatura, pressione, sensori chimici
tutor interattivo per lo studio dell'inglese	massimizzare i risultati del test dello studente	insieme di studenti, scuola erogatrice di test	visore per proporre esercizi, suggerimenti, correzioni	input da tastiera

Figura 2.5
Esempi di tipi di agente e loro descrizioni PEAS.

la posizione accurata in una mappa elettronica, sensori a infrarossi o sonar per calcolare la distanza con gli altri veicoli ed eventuali ostacoli. Infine, il passeggero avrà bisogno di un microfono o di una tastiera per indicare la sua destinazione.

Nella Figura 2.5 abbiamo abbozzato gli elementi base PEAS per diversi altri tipi di agente. Ulteriori esempi compaiono nell'Esercizio 2.5. Alcuni lettori potrebbero sorprendersi che nella nostra lista siano inclusi alcuni programmi che operano nell'ambiente interamente artificiale definito da una tastiera per l'input e uno schermo per l'output. "Certamente", si potrebbe dire, "questo non è un ambiente reale, no?". In effetti, quello che importa non è la distinzione tra ambienti "reali" e "artificiali", ma la complessità delle relazioni tra il comportamento dell'agente, la sequenza percettiva generata dall'ambiente e la misura di prestazione. Al-

agenti software
softbot

cuni ambienti “reali” sono di fatto abbastanza semplici. Ad esempio, un robot progettato per ispezionare le parti meccaniche che gli scorrono davanti su un nastro trasportatore potrà assumere molte ipotesi semplificative: l’illuminazione sarà sempre costante, le uniche cose sul nastro saranno parti meccaniche di cui è a conoscenza, e ci saranno solo due possibili azioni (accetta o rifiuta il pezzo corrente).

Al contrario, alcuni agenti software (detti anche “software robot”, o softbot) operano in domini complessi e senza confini. Immaginate un softbot progettato per agire all’interno del simulatore di volo di un grande aereo commerciale. Il simulatore rappresenta un ambiente complesso e molto dettagliato, che include altri velivoli e operazioni a terra, e l’agente software dovrà scegliere in tempo reale tra una vasta gamma di possibili azioni. Immaginate anche un softbot progettato per consultare diversi siti Internet informativi e mostrare ai suoi utenti le notizie a cui sono più interessati. Per lavorare bene dovrà possedere una certa capacità di comprendere il linguaggio naturale, imparare gli argomenti che interessano maggiormente ogni utente ed essere capace di cambiare dinamicamente i propri piani, ad esempio quando un sito non è più raggiungibile o quando ne vengono aperti di nuovi. Internet è un ambiente la cui complessità rivaleggia con quella del mondo reale e i cui abitanti includono molti agenti artificiali.

Proprietà degli ambienti

La varietà di ambienti operativi nell’IA è naturalmente molto vasta. È comunque possibile identificare un numero relativamente piccolo di proprietà in base a cui suddividerli in categorie. Queste proprietà determinano in gran parte la progettazione di agenti appropriati e l’applicabilità delle principali tecniche alla loro implementazione. Prima di tutto elencheremo le dimensioni, in seguito analizzeremo diversi task environment esemplificativi. Le definizioni qui sotto sono informali: nei capitoli successivi ne forniremo di più precise, insieme ad altri esempi di ogni tipo di ambiente.

completamente osservabile

- ◆ **Completamente osservabile/parzialmente osservabile:** se i sensori di un agente gli danno accesso allo stato completo dell’ambiente in ogni momento, allora diciamo che l’ambiente operativo è completamente osservabile.⁴ In effetti, perché un task environment sia completamente osservabile basta che i sensori dell’agente misurino tutti gli aspetti che sono rilevanti per la scelta

⁴ La prima edizione di questo libro usava i termini *accessible* e *inaccessible* al posto di *totalmente e parzialmente osservabile*; *non deterministico* invece di *stocastico* e *non episodico* invece di *sequenziale*. La nuova terminologia è più vicina all’uso comune.

dell'azione; la rilevanza a sua volta dipende dalla misura di prestazione. Gli ambienti completamente osservabili sono comodi, perché l'agente non deve memorizzare uno stato interno per tenere traccia del mondo. Un ambiente potrebbe essere parzialmente osservabile a causa di sensori inaccurati o per la presenza di rumore, o semplicemente perché una parte dei dati non viene rilevata dai sensori: ad esempio, un agente aspirapolvere con un sensore locale di rilevazione della sporcizia non potrà sapere se ci sono altri riquadri sporchi, e un pilota automatico per taxi non potrà in ogni caso sapere le intenzioni degli altri guidatori.

- ◆ Deterministico/stocastico: se lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente, allora si può dire che l'ambiente è deterministico; in caso contrario si dice che è stocastico. In via di principio, un agente non si deve preoccupare dell'incertezza se si trova in un ambiente completamente osservabile e deterministico. Tuttavia, se l'ambiente è solo parzialmente osservabile potrebbe sembrare stocastico. Questo è particolarmente vero nel caso di ambienti complessi, che rendono difficile tener traccia di tutti gli aspetti non osservati. Di conseguenza, spesso conviene pensare che un ambiente sia deterministico o stocastico dal punto di vista dell'agente. In questi termini, guidare un taxi è chiaramente un'attività stocastica, perché nessuno può prevedere esattamente l'andamento del traffico; inoltre le forature e i guasti del motore si possono verificare senza preavviso. Il mondo dell'aspirapolvere come l'abbiamo descritto è deterministico, ma si potrebbero facilmente descrivere delle varianti che includono elementi stocastici, come sporco che appare casualmente o un meccanismo di aspirazione poco affidabile (Esercizio 2.12). Se l'ambiente è deterministico in tutto tranne che per le azioni di altri agenti, si dice allora che è strategico.
- ◆ Episodico/sequenziale⁵: in un ambiente operativo episodico, l'esperienza dell'agente è divisa in episodi atomici. Ogni episodio consiste nella percezione dell'agente seguita dall'esecuzione di una singola azione. L'aspetto fondamentale è che un episodio non dipende dalle azioni intraprese in quelli precedenti: negli ambienti episodici, la scelta dell'azione dipende solo dall'episodio corrente. Molte attività di classificazione sono episodiche. Per esempio, un agente che deve identificare i pezzi difettosi in una catena di montaggio basa ogni decisione solamente sul pezzo esaminato in quel momento, indipendentemente dalle decisioni prese in precedenza; inoltre, la de-

S. S. 2014 - 2015 - A. 2014
deterministico
stocastico

strategico
episodico
sequenziale

⁵ La parola "sequenziale" è anche usata nell'informatica in contrapposizione a "parallelo". I due significati sono indipendenti.

Sequenziali

cisione corrente non renderà più o meno probabile che il pezzo successivo sia difettoso. Negli ambienti sequenziali, al contrario, ogni decisione può influenzare tutte quelle successive. Gli scacchi e la guida dei taxi sono sequenziali: in entrambi i casi le azioni a breve termine possono avere conseguenze a lungo termine. Gli ambienti episodici sono molto più semplici di quelli sequenziali, perché l'agente non deve "pensare avanti".

statico
dinamico

Taxi

Statico/dinamico: se l'ambiente può cambiare mentre un agente sta pensando, allora diciamo che è dinamico per quell'agente; in caso contrario diciamo che è statico. Gli ambienti statici sono più facili da trattare perché l'agente non deve continuare a osservare il mondo mentre decide l'azione successiva e non si deve preoccupare del passaggio del tempo. Gli ambienti dinamici, invece, chiedono continuamente all'agente quello che vuole fare; se questo non risponde in tempo, è come se avesse deciso di non fare nulla. Se l'ambiente stesso non cambia al passare del tempo, ma la valutazione della prestazione dell'agente sì, allora diciamo che l'ambiente è semidinamico. Guidare un taxi è chiaramente dinamico: le altre macchine e il taxi stesso continuano a muoversi mentre l'algoritmo di guida pondera la mossa successiva. Gli scacchi sono semidinamici se giocati con l'orologio. I cruciverba sono statici.

semidinamico

discreto
continuo

◆ **Discreto/continuo:** la distinzione tra discreto e continuo può essere applicata allo stato dell'ambiente, al modo in cui è gestito il tempo, alle percezioni e azioni dell'agente. Ad esempio, un ambiente a stati discreti, come una scacchiera, ha un numero finito di stati distinti. Gli scacchi hanno anche un insieme discreto di percezioni e azioni. La guida di un taxi è un problema con stato e tempo continui: la velocità e la posizione del taxi e degli altri veicoli cambiano con continuità al passare del tempo. In questo caso sono continue anche le azioni (pensiamo ad esempio all'angolo di sterzo delle ruote). L'input proveniente da una telecamera digitale, strettamente parlando, è discreto, ma tipicamente si considera che rappresenti intensità e posizioni che variano con continuità.

agente singolo
multiagente

◆ **Agente singola/multiagente:** la distinzione tra ambienti ad agente singolo e multiagente può sembrare piuttosto semplice. Ad esempio, un agente che da solo risolve un cruciverba chiaramente opera in un ambiente ad agente singolo, laddove un agente che gioca a scacchi si trova in un ambiente a due agenti. Tuttavia la questione è sottile. Innanzitutto, abbiamo descritto in che modo un'entità può essere vista come agente, ma non abbiamo spiegato quali entità devono essere considerate tali. Un agente *A* (l'autista del taxi, per esempio) deve considerare l'oggetto *B* (un altro veicolo) come un agente, oppure lo può trattare come un semplice oggetto che si comporta stocasticamente, simile alle onde sul bagnasciuga o a foglie spinte dal vento? La distinzione chiave è se si può descrivere il comportamento di *B* come il tentativo di massimizzare una misura di prestazione il cui valore dipende dal comportamento dell'agente *A*.

Per esempio, negli scacchi, l'avversario *B* sta cercando di massimizzare una misura di prestazione che, per le regole degli scacchi, minimizza quella dell'agente *A*. Gli scacchi, quindi, sono un ambiente multiagente competitivo. Nell'ambiente del traffico, al contrario, evitare gli incidenti massimizza la misura delle prestazioni di tutti gli agenti, per cui possiamo dire che è un ambiente multiagente parzialmente cooperativo. È anche parzialmente competitivo perché, tra le altre cose, una sola macchina può occupare un parcheggio libero. I problemi di progettazione che sorgono negli ambienti multiagente sono spesso molto differenti da quelli ad agente singolo: ad esempio, spesso nei primi può emergere come comportamento razionale la comunicazione. In alcuni ambienti competitivi parzialmente osservabili, il comportamento stocastico è razionale perché permette di evitare la predicibilità delle azioni.

competitivo

cooperativo

Come ci si potrebbe aspettare, il caso più difficile è quello parzialmente osservabile, stocastico, sequenziale, dinamico, continuo e multiagente. La maggior parte delle situazioni reali, poi, sono così complesse che è inutile chiedersi se in realtà sono deterministiche: per tutti gli usi pratici vengono effettivamente trattate come stocastiche. Guidare un taxi è difficile sotto tutti questi punti di vista.

La Figura 2.6 elenca le proprietà di diversi ambienti di esempio. Notate che le risposte non sono sempre sicure al cento per cento. Ad esempio, abbiamo scritto che gli scacchi sono un ambiente completamente osservabile; questo non è strettamente vero perché le regole dell'arrocco, della cattura *en passant* e della patta per ripetizione richiedono di avere memoria della storia *del gioco* e non sono osservabili come parte dello stato della scacchiera. Queste eccezioni all'osservabilità sono ovviamente di piccola entità rispetto a quelle che devono essere gestite dall'autista di taxi, dal maestro d'inglese o dal sistema di diagnosi medica.

Alcune altre risposte nella tabella dipendono dalla definizione del task environment. Abbiamo indicato l'attività di diagnosi medica ad agente singolo perché non c'è ragione di modellare come agente il processo patologico in atto nel paziente; il sistema di diagnosi medica però potrebbe dover gestire pazienti recalcitranti e medici scettici, e questo potrebbe essere rappresentato come un aspetto multiagente. Inoltre, la diagnosi medica è episodica se si immagina di formularne semplicemente una sulla base di una lista di sintomi; il problema diventa sequenziale se l'attività prevede di proporre una serie di test, valutare l'efficacia di una terapia e così via. Può anche darsi che un ambiente sia episodico a livelli più alti di quelli delle singole azioni dell'agente. Ad esempio, un torneo di scacchi consiste in una serie di partite; ognuna costituisce un episodio, perché le mosse non sono influenzate da quelle eseguite nelle partite precedenti. D'altra parte all'interno di una singola partita le decisioni sono certamente sequenziali..

task environment	osservabile	deterministico	episodico	statico	discreto	agenti
cruciverba scacchi con orologio	completamente completamente	deterministico strategico	sequenziale sequenziale	statico semi	discreto discreto	singolo multi
poker - backgammon	parzialmente completamente	stocastico stocastico	sequenziale sequenziale	statico statico	discreto discreto	multi multi
autista di taxi diagnosi medica	parzialmente parzialmente	stocastico stocastico	sequenziale sequenziale	dinamico dinamico	continuo continuo	multi singolo
analizzatore di immagini robot di selezione per parti meccaniche	completamente parzialmente	deterministico stocastico	episodico episodico	semi dinamico	continuo continuo	singolo singolo
controllore per una raffineria maestro di inglese interattivo	parzialmente parzialmente	stocastico stocastico	sequenziale sequenziale	dinamico dinamico	continuo discreto	singolo multi

Figura 2.6 Esempi di task environment e loro caratteristiche.

classe di ambienti

La collezione di codice associata a questo libro (aima.cs.berkeley.edu) include l'implementazione di un certo numero di ambienti, insieme a un simulatore di uso generale che pone uno o più agenti all'interno di un ambiente simulato, osserva il suo comportamento per un certo lasso di tempo e lo valuta sulla base di una data misura di prestazione. Questi esperimenti spesso non vengono eseguiti per un singolo ambiente, ma per più elementi di una classe di ambienti. Per valutare un agente tassista immerso nel traffico, ad esempio, vorremo eseguire molte simulazioni variando il traffico, le condizioni di luce e quelle meteorologiche. Progettando l'agente per un singolo scenario potremmo avvantaggiarci delle caratteristiche specifiche del caso particolare prescelto, ma d'altra parte non potremmo identificare un buon progetto applicabile alla guida in generale. Per questa ragione il codice include anche un generatore di ambienti per ogni classe, che seleziona ambienti particolari in cui mettere alla prova un agente. Ad esempio, il generatore di ambienti per il mondo aspirapolvere inizializza casualmente la configurazione dello sporco e la posizione dell'agente. Quello che ci interessa, allora, sono le prestazioni medie dell'agente su tutta la classe di ambienti: un agente razionale per una data classe di ambienti massimizza tali prestazioni medie. Gli Esercizi dal 2.7 al 2.12 vi guideranno attraverso il processo di sviluppo di una classe di ambienti e la valutazione di diversi agenti al suo interno.

generatore di ambienti

2.4 La struttura degli agenti

Fin qui, parlando degli agenti, ci siamo limitati a descrivere il loro comportamento, ovvero l'azione eseguita in corrispondenza di una data sequenza di percezioni. Ora dovremo fare un passo avanti e cominciare a descrivere il loro funzionamento interno. Il compito dell'IA è progettare il programma agente che implementa la funzione agente, mettendo in relazione percezioni e azioni. Diamo per scontato che questo programma sarà eseguito da un computer dotato di sensori fisici e attuatori; questa prende il nome di architettura:

programma agente

architettura

I agente = architettura + programma.

Naturalmente, il programma prescelto dovrà essere appropriato per l'architettura: ad esempio, se il programma prevede azioni come Cammina, dovrà essere fornita di gambe. L'architettura potrebbe essere costituita da un semplice PC di uso quotidiano, o un veicolo robotico dotato di vari computer, telecamere e altri sensori. In generale l'architettura si occupa di rendere le percezioni disponibili al programma, eseguire il programma stesso e passare le azioni da esso prescelte agli attuatori man mano che vengono generate.

Programmi agente

I programmi agente che progetteremo nel corso del libro hanno tutti la stessa struttura: prendono come input la percezione corrente dei sensori e restituiscono un'azione agli attuatori.⁶ Notate la differenza tra il programma agente, che prende come input solamente la percezione corrente, e la funzione agente, il cui input è costituito dall'intera storia delle percezioni. Il programma agente si basa sulla sola percezione corrente perché l'ambiente non può fornirgli nulla di più; se le sue azioni dipendono dalla sequenza percettiva precedente, l'agente stesso dovrà preoccuparsi di memorizzarla.

Descriveremo i programmi agente con il semplice pseudocodice definito nell'Appendice B (il deposito online di codice contiene implementazioni in vari linguaggi di programmazione reali). Ad esempio, la Figura 2.7 mostra un programma agente abbastanza banale che tiene traccia della sequenza percettiva e poi la usa per selezionare l'azione da intraprendere in una tabella. La tabella rappresenta in modo esplicito la funzione agente implementata dal programma. Per costruire un

⁶ Ci sono altre possibili scelte per lo schema di un programma agente; ad esempio questo potrebbe essere realizzato mediante coroutine in esecuzione asincrona insieme all'ambiente. Ogni coroutine avrebbe porte di input e output e consisterebbe in un ciclo infinito che legge dalla porta di input e scrive azioni su quella di output.

```

function AGENTE-CON-TABELLA(percezione) returns un'azione
  static: percezioni, una sequenza inizialmente vuota
  tabella, una tabella di azioni, indicizzata per sequenze percettive, pienamente
  specificata dall'inizio
    append percezione alla fine di percezioni
    azione  $\leftarrow$  LOOKUP(percezioni, tabella)
  return azione

```

Figura 2.7 Il programma AGENTE-CON-TABELLA viene invocato per ogni nuova percezione e restituisce ogni volta l'azione da eseguire. Per tener traccia della sequenza percettiva utilizza una struttura dati privata.

agente razionale con questa tecnica, i progettisti devono costruire una tabella che contenga l'azione appropriata per ogni possibile sequenza percettiva.

È utile comprendere subito perché l'approccio basato su tabelle esplicite sia destinato al fallimento. Sia P l'insieme di possibili percezioni e T la durata della vita dell'agente (ovvero il numero totale di percezioni che riceverà). La tabella dovrà contenere $\sum_{t=1}^T |P|^t$ righe. Considerate il caso del taxi automatico: l'input visuale di una singola telecamera corrisponde a un flusso di informazioni di circa 27 me-gabyte al secondo (30 fotogrammi a una risoluzione di 640×480 pixel con 24 bit di profondità di colore). Per un'ora di guida, questo significa che la tabella dovrà avere $10^{250.000.000.000}$ righe. Anche la tabella per il gioco degli scacchi – un frammento molto piccolo del mondo reale, che varia secondo modalità ben conosciute – richiederebbe almeno 10^{150} righe. L'enorme dimensione di queste tabelle (il numero di atomi dell'intero universo osservabile è meno di 10^{80}) significa che (a) nessun agente fisico nell'universo avrà mai lo spazio necessario per memorizzarle, (b) il progettista non avrà mai il tempo di crearle, (c) nessun agente avrà mai il tempo di imparare le azioni in base all'esperienza e (d) anche se l'ambiente fosse abbastanza piccolo da permettere la creazione di una tabella di dimensioni accettabili, il progettista continuerebbe a non avere alcun criterio per riempirne le righe.

Nonostante tutto ciò, il nostro AGENTE-CON-TABELLA *fa* quello che vogliamo, implementando effettivamente la funzione agente desiderata. La sfida principale dell'IA sta nel trovare il modo di scrivere programmi che, nella massima misura possibile, producano comportamento razionale con una piccola quantità di codice invece che con la grande quantità di righe di un'enorme tabella. Molti casi dimostrano che quest'obiettivo può essere ottenuto in altre discipline: ad esempio, le grandi tabelle di radici quadrate usate sia dai bambini che dagli ingegneri prima degli anni '70 sono state sostituite da un programma di cinque righe che imple-

menta il metodo di Newton nelle calcolatrici tascabili. La domanda è questa: è possibile che l'IA faccia per il comportamento intelligente cio che Newton ha fatto per le radici quadrate? Noi crediamo che la risposta sia sì.

Nel resto di questo paragrafo delineeremo quattro tipi base di programma agente che rappresentano i principi alla base di quasi tutti i sistemi intelligenti:

- ◆ agenti reattivi semplici
- ◆ agenti reattivi basati su modello
- ◆ agenti basati su obiettivi
- ◆ agenti basati sull'utilità.

Spiegheremo dopo, in termini generali, come convertire tutti questi tipi in agenti capaci di apprendere.

Agenti reattivi semplici

Il tipo più semplice è l'agente reattivo semplice. Questi agenti scelgono le azioni sulla base della percezione corrente, ignorando tutta la storia percettiva precedente. Ad esempio, l'aspirapolvere della Figura 2.3 è un agente reattivo semplice, perché la sua decisione è basata unicamente sulla posizione corrente e se questa contiene dello sporco o no. La Figura 2.8 mostra uno dei suoi possibili programmi agente.

Notate che il programma agente dell'aspirapolvere è molto più piccolo della tabella corrispondente. La riduzione più importante deriva dall'aver ignorato la storia delle percezioni, cosa che riduce il numero di possibilità da 4⁷ a solo 4. Un'ulteriore, piccola riduzione deriva dal fatto che, quando il riquadro corrente è sporco, l'azione non dipende dalla posizione.

Immaginatevi ora alla guida del taxis automatico. Se la macchina davanti a voi comincia a frenare, e si accendono le relative luci di segnalazione, dovreste notare questo fatto e cominciare anche voi a frenare. In altre parole, devono essere svolti

{agente reattivo
semplice}

function AGENTE-REATTIVO-ASPIRAPOLVERE([posizione, stato]) returns un'azione

```

if stato = Sporco then return Aspira
else if posizione = A then return Destra
else if posizione = B then return Sinistra

```

Figura 2.8 Il programma agente per l'agente reattivo semplice nell'ambiente dell'aspirapolvere a due stati. Questo programma implementa la funzione agente corrispondente alla tabella della Figura 2.3.

regola
condizione-azione

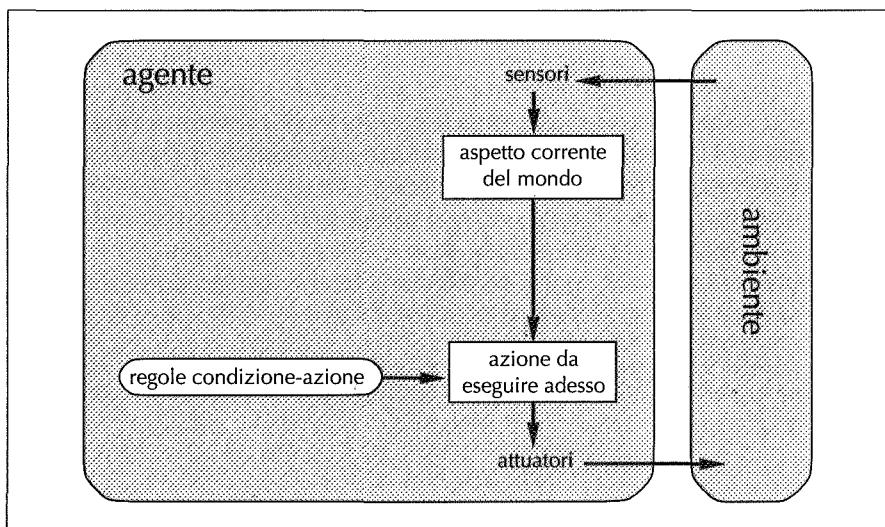
alcuni calcoli sull'input visivo per stabilire la condizione che descriviamo con la frase "la macchina davanti sta frenando". Questo farà poi scattare, all'interno del programma agente, una connessione con l'azione "inizia a frenare". Questa connessione prende il nome di regola condizione-azione,⁷ e si scrive

if la-macchina-davanti-frena then inizia-a-frenare.

Anche gli esseri umani fanno uso di connessioni analoghe, alcune delle quali derivano dall'apprendimento (come nel caso della guida), altre invece da riflessi innati (come chiudere le palpebre quando qualcosa si avvicina rapidamente agli occhi). Nel corso del libro vedremo molti modi diversi di apprendere queste connessioni e implementarle in un programma.

Il programma nella Figura 2.8 è scritto ad hoc, specificatamente per un ambiente. Un approccio più generale e flessibile consiste nel costruire per prima cosa un interprete di regole condizione-azione di uso generale e poi scrivere gli insiemi di regole specifiche per i vari ambienti operativi. La Figura 2.9 illustra in forma schematica la struttura di questo programma generale, mostrando come le regole condizione-azione permettono all'agente di stabilire una connessione da percezione ad azione (se tutto questo vi sembra banale, non preoccupatevi; ben presto si

Figura 2.9
Diagramma schematico di un agente reattivo semplice.



⁷ Queste regole sono anche chiamate regole situazione-azione, produzioni o regole if-then.

```
function AGENTE-REATTIVO-SEMPLICE(percezione) returns un'azione
```

static: *regole*, un insieme di regole condizione-azione

stato \leftarrow INTERPRETA-INPUT(*percezione*)

regola \leftarrow REGOLA-CORRISPONDENTE(*stato*, *regole*)

azione \leftarrow REGOLA-AZIONE[*regola*]

return *azione*

Figura 2.10 Un agente reattivo semplice che agisce secondo una regola la cui condizione corrisponde allo stato corrente, dedotto dalla percezione.

farà più interessante). Per denotare lo stato interno corrente del processo decisionale usiamo dei rettangoli, mentre gli ovali rappresentano le informazioni di base usate nel processo. Il programma agente corrispondente, anch'esso molto semplice, è mostrato nella Figura 2.10. La funzione INTERPRETA-INPUT genera una descrizione astratta dello stato corrente partendo dalla percezione, mentre la funzione REGOLA-CORRISPONDENTE restituisce la prima regola dell'insieme che corrisponde a tale descrizione. Notate che la descrizione del programma in termini di "regole" e "corrispondenza" è totalmente astratta; le effettive implementazioni possono essere molto più semplici, come una serie di porte logiche che implementano in hardware un circuito booleano.

Gli agenti reattivi semplici hanno l'ammirevole proprietà di essere, appunto, semplici, ma la loro intelligenza è molto limitata. L'agente nella Figura 2.10 funzionerà solo se si può selezionare la decisione corretta in base alla sola percezione corrente, ovvero solo nel caso in cui l'ambiente sia completamente osservabile. Anche una minima parte di non-osservabilità può causare grandi problemi. Ad esempio, la regola per frenare che abbiamo formulato poco fa dà per scontato che la condizione la-macchina-davanti-frena possa essere determinata partendo dalla percezione corrente (l'immagine video) e presumendo che la macchina davanti a noi abbia una luce di segnalazione della frenata montata centralmente. Sfortunatamente, i modelli più vecchi hanno diverse configurazioni di luci di posizione, frecce e segnalatori di frenata, e non sempre si può stabilire da una singola immagine se la macchina sta frenando. Un agente reattivo semplice posto alla guida dietro una macchina siffatta continuerebbe a frenare senza ragione o, ancora peggio, non frenerebbe mai.

Possiamo riscontrare un problema simile anche nel mondo dell'aspirapolvere. Supponiamo che un agente reattivo semplice sia privato del suo sensore di posizione e abbia solo il rilevatore di sporco. Quest'agente avrebbe due sole possibili percezioni: [Sporco] e [Pulito]. In risposta a [Sporco] l'agente può Aspirare; ma cosa dovrebbe fare in risposta a [Pulito]? Se per caso è partito nel riquadro A, muoversi a



randomizzare

Sinistra fallirebbe (per sempre); alla stessa maniera sarebbe errato (per sempre) scegliere di muoversi a *Destra* se l'agente è inizialmente posizionato nel riquadro *B*. Spesso gli agenti reattivi semplici non sono in grado di evitare cicli infiniti quando operano in ambienti parzialmente osservabili.

Evitare i cicli infiniti è possibile quando l'agente è in grado di randomizzare le sue azioni, aggiungendo alla scelta una componente casuale. Ad esempio, quando l'agente aspirapolvere percepisce *[Pulito]*, potrebbe tirare una moneta per decidere se muoversi a *Sinistra* o a *Destra*. È facile dimostrare che, in media, l'agente raggiungerà l'altro riquadro in due passi. A questo punto, se il riquadro è sporco lo potrà pulire, e la sua attività sarà terminata. Di conseguenza un agente reattivo semplice randomizzato può comportarsi meglio del suo corrispondente deterministico.

Abbiamo menzionato nel Paragrafo 2.3 che, in alcuni sistemi multiagente, un comportamento che include adeguate componenti casuali può essere razionale. Negli ambienti ad agente singolo la randomizzazione solitamente non è razionale: si può sfruttare come "trucco" per aiutare un agente reattivo semplice a districarsi in certe situazioni, ma nella maggior parte dei casi si può fare molto meglio ricorrendo ad agenti deterministici più sofisticati.

Agenti reattivi basati su modello

Il modo più efficace di gestire l'osservabilità parziale, per un agente, è tener traccia della parte del mondo che non può vedere nell'istante corrente. Questo significa che l'agente deve memorizzare una sorsa di stato interno che dipende dalla storia delle percezioni e che quindi riflette almeno una parte degli aspetti non osservabili dello stato corrente. Ritornando al problema della frenata, lo stato interno non è troppo complesso: basta mantenere in memoria il fotogramma precedente "scattato" dalla telecamera, permettendo così all'agente di accorgersi quando due luci rosse alle estremità laterali del veicolo si accendono o spengono contemporaneamente. Per altri compiti, come il cambio di corsia, l'agente deve tener traccia della posizione di tutte le macchine che non può vedere direttamente.

Aggiornare l'informazione di stato al passaggio del tempo richiede che il programma agente possieda due tipi di conoscenza. Prima di tutto, deve avere qualche informazione sull'evoluzione del mondo indipendentemente dalle sue azioni; ad esempio, una macchina che ci sta superando sarà in genere più vicina di quanto fosse un momento fa. In secondo luogo, sono necessarie delle informazioni sull'effetto che hanno sul mondo le azioni dell'agente stesso: ad esempio, occorre sapere che quando l'agente gira il volante a destra la macchina svolta in quella direzione, o che dopo aver guidato in autostrada verso nord per cinque minuti la nostra posizione sarà più o meno cinque miglia più a nord di dove eravamo cinque minuti fa. Questa conoscenza sul "funzionamento del mondo", implementata mediante semplici circuiti logici o sviluppata in una teoria scientifica completa, viene chiamata

stato interno

2.4 La struttura degli agenti

67

...pende dello stato di percezione

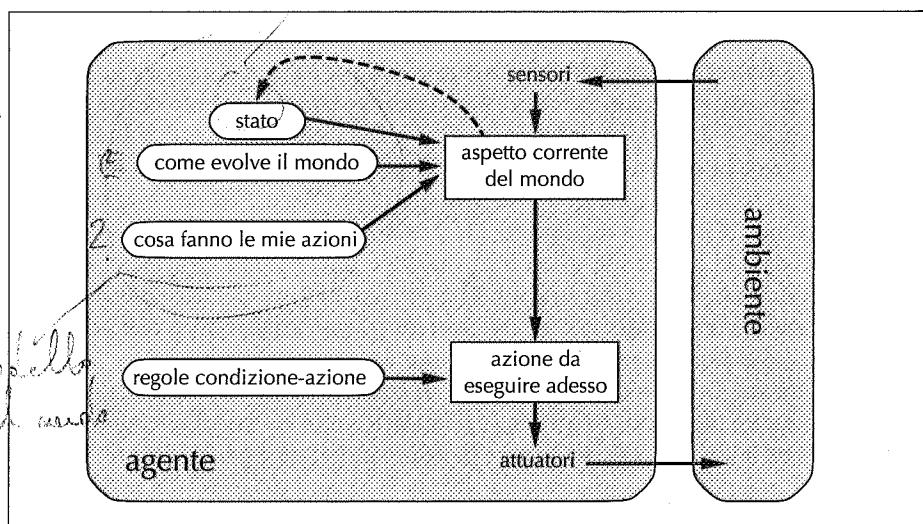


Figura 2.11
Un agente reattivo basato su modello.

modello del mondo. Una agente che si appoggia a un simile modello prende appunto il nome di **agente basato su modello**.

La Figura 2.11 illustra la struttura di un agente reattivo dotato di stato interno, mostrando come la descrizione aggiornata dello stato scaturisce dalla combinazione del vecchio stato interno e della percezione corrente. Il programma agente è riportato nella Figura 2.12. La parte interessante è la funzione AGGIORNA-STATO, responsabile della creazione del nuovo stato interno. Oltre a interpretare la nuova percezione alla luce della conoscenza preesistente dello stato, la funzione utilizza informazione sull'evoluzione del mondo per tener traccia delle parti di esso che non sono presentemente visibili. La funzione deve anche conoscere l'effetto delle azioni dell'agente sullo stato del mondo.

agente basato su
modello

Imp.

```

function AGENTE-REATTIVO-CON-STATO(percezione) returns un'azione
  static: stato, una descrizione dello stato corrente del mondo
  regole, un insieme di regole condizione-azione
  azione, l'azione più recente, inizialmente nessuna

  stato  $\leftarrow$  AGGIORNA-STATO(stato, azione, percezione)
  regola  $\leftarrow$  REGOLA-CORRISPONDENTE(stato, regole)
  azione  $\leftarrow$  REGOLA-AZIONE[regola]
  return azione

```

Figura 2.12 Un agente reattivo basato su modello, che tiene traccia dello stato corrente del mondo mediante uno stato interno. A parte questo, l'agente sceglie l'azione da eseguire come un normale agente reattivo.

obiettivo

Agenti basati su obiettivi

Conoscere lo stato corrente dell'ambiente non sempre basta a decidere che cosa fare. Ad esempio, arrivati a un incrocio il taxi può girare a sinistra, a destra oppure continuare dritto; la decisione corretta dipende dalla sua destinazione. In altre parole, oltre che della descrizione corrente dello stato l'agente ha bisogno di qualche tipo di informazione riguardante il suo obiettivo (*goal*) che descriva situazioni desiderabili come ad esempio raggiungere la destinazione richiesta dal passeggero. Il programma agente può unire quest'informazione a quella che riguarda i risultati delle possibili azioni (la stessa usata anche per aggiornare lo stato interno in un agente reattivo) per scegliere quelle che portano al soddisfacimento dell'obiettivo. La Figura 2.13 illustra la struttura di un agente basato su obiettivi.

Talvolta scegliere un'azione in base a un obiettivo è molto semplice, quando questo può essere raggiunto in un solo passo. Altre volte è più difficile, quando l'agente deve considerare lunghe sequenze di azioni alternative per trovare il "cammino" che porta al risultato agognato. La ricerca (Capitoli dal 3 al 6) e la pianificazione (Capitoli 11 e 12) sono dei sottocampi dell'IA dedicati proprio a identificare le sequenze di azioni che permettono a un agente di raggiungere i propri obiettivi.

Notate che questo tipo di decisioni non ha nulla a che vedere con le regole condizione-azione che abbiamo descritto prima, perché ora dobbiamo prendere in considerazione il futuro sotto due aspetti: "cosa accadrà se faccio così e così?" e anche "se faccio questo sarò soddisfatto?" Nella progettazione degli agenti reattivi, quest'informazione non viene rappresentata esplicitamente, perché le regole interne mettono direttamente in corrispondenza percezioni e azioni. L'agente reattivo frena quando vede le luci di frenata. Un agente basato su obiettivi, in via di principio, potrebbe ragionare che se la macchina di fronte ha le luci di frenata accese, starà rallentando. Dato il modo in cui evolve normalmente il mondo, l'unica azione che permetterebbe di raggiungere l'obiettivo di/non tamponare altre macchine sarà allora quella di frenare.

Benché un agente basato su obiettivi sembri meno efficiente, d'altra parte è più flessibile, perché la conoscenza che motiva le sue decisioni è rappresentata esplicitamente e può essere modificata. Se comincia a piovere, l'agente può aggiornare la propria conoscenza circa l'efficienza dei propri freni; questo automaticamente farà variare tutti i comportamenti relativi in modo da adattarli alle nuove condizioni. Nel caso dell'agente reattivo, invece, dovremmo riscrivere daccapo molte regole condizione-azione. Il comportamento dell'agente basato su obiettivi può essere facilmente alterato per farlo andare verso una destinazione diversa. Le regole dell'agente reattivo che governano quando sterzare e quando andare dritti, invece, funzioneranno solo per una singola destinazione finale; per andare da qualche altra parte dovranno essere tutte riscritte.

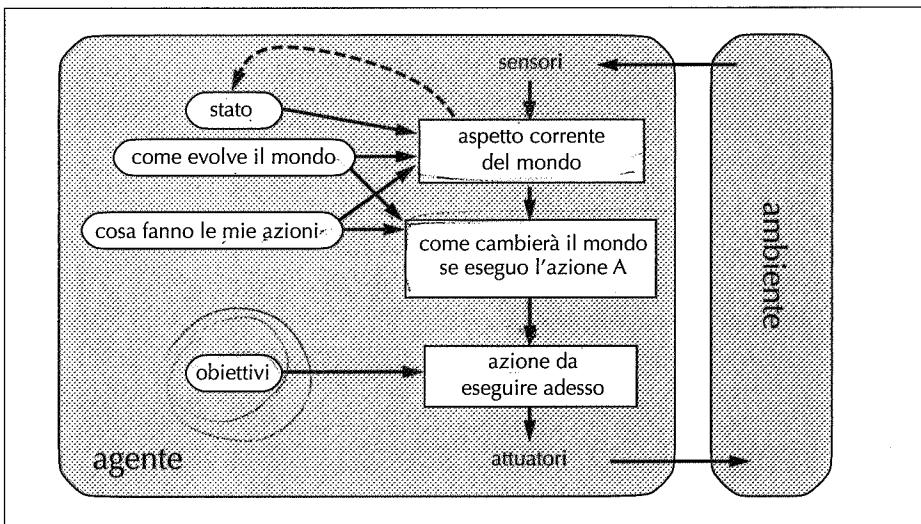


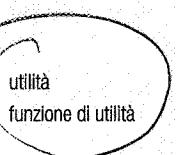
Figura 2.13 Un agente dotato di modello del mondo e basato su obiettivi. Oltre a tener traccia dello stato dell'ambiente, l'agente memorizza un insieme di obiettivi e sceglie l'azione che lo porterà (a un certo punto) a soddisfarli.

Agenti basati sull'utilità

Nella maggior parte degli ambienti gli obiettivi, da soli, non bastano a generare un comportamento di alta qualità. Ad esempio, ci sono molte sequenze di azioni che porteranno un taxi alla sua destinazione (soddisfando così il suo obiettivo) ma alcune sono più veloci, sicure, affidabili o economiche di altre. Gli obiettivi forniscono solamente una distinzione binaria tra stati "contenti" e "scontenti", laddove una misura di prestazione più generale dovrebbe permettere di confrontare stati del mondo differenti e misurare precisamente quanto sarebbe contento l'agente se riuscisse a raggiungerli. Dato che "contentezza" non suona molto scientifico, nella terminologia corrente si dice che uno stato del mondo preferibile a un altro ha, per l'agente, una maggiore utilità.

Una funzione di utilità assegna a uno stato (o a una sequenza di stati) un numero reale che quantifica il grado di contentezza a esso associato. Una specifica completa della funzione di utilità permette di prendere decisioni razionali in due categorie di casi in cui non bastano i soli obiettivi. Prima di tutto, quando ci sono più obiettivi in conflitto che non si possono ottenere tutti insieme (per esempio, velocità e sicurezza), la funzione di utilità specifica quali privilegiare. In secondo luogo, quando ci sono più obiettivi raggiungibili ma nessuno può essere ottenuto con certezza, il concetto di utilità fornisce un mezzo per confrontare le probabilità di successo e l'importanza degli obiettivi.

Nel Capitolo 16 (nel 2° vol.) vedremo che ogni agente razionale deve comportarsi come se possedesse una funzione di utilità di cui cerca di massimizzare il valore atteso. Un agente che possiede una funzione di utilità esplicita può quindi pren-



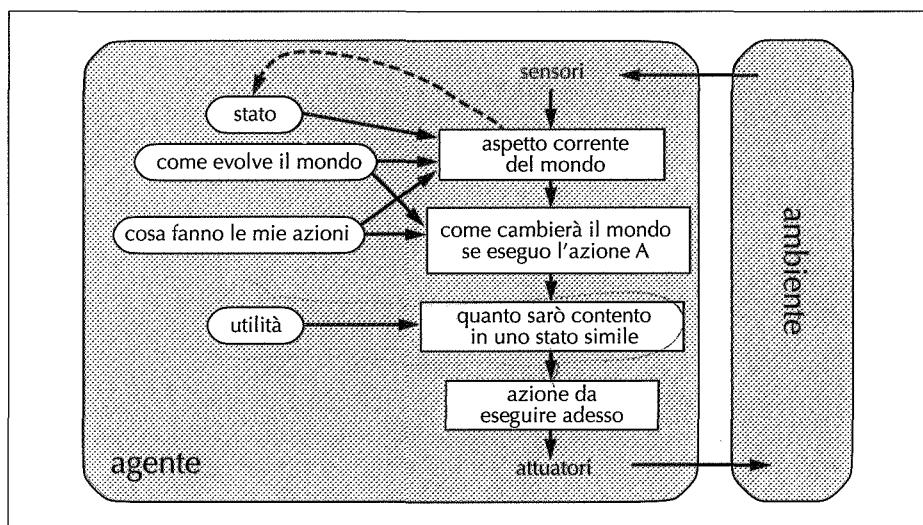


Figura 2.14 Un agente dotato di modello del mondo e basato sull'utilità. Oltre a tener traccia dello stato dell'ambiente, l'agente memorizza una funzione utilità che misura le sue preferenze tra i vari stati del mondo. L'azione prescelta è quella che massimizza l'utilità attesa, calcolata come media pesata dei valori degli stati possibili per la rispettiva probabilità di verificarsi.

dere decisioni razionali, e lo può fare seguendo un algoritmo generale che non dipende dalla specifica funzione di utilità che si desidera massimizzare. In questo modo la definizione “globale” di razionalità, che definisce razionali quelle funzioni agente che hanno le prestazioni migliori, viene trasformata in un vincolo “locale” in progetti di agenti razionali che possono essere espressi in un semplice programma.

La Figura 2.14 mostra la struttura di un agente basato sull'utilità. I programmi agente basati sull'utilità saranno trattati nella Parte V, in cui progetteremo agenti capaci di prendere decisioni che considerano l'incertezza intrinseca negli ambienti parzialmente osservabili.

Agenti capaci di apprendere

Fin qui abbiamo descritto programmi agente che usano vari metodi per scegliere le azioni. Non abbiamo ancora spiegato in che modo nascono i programmi agente. Nel suo famoso vecchio articolo, Turing (1950) considera l'idea di programmare a mano le sue macchine intelligenti. Dopo aver stimato il lavoro necessario, concludeva “sembra auspicabile un metodo più rapido”. Il metodo proposto dallo stesso Turing è costruire macchine capaci di apprenderne e poi addestrarle. In molti campi dell'IA, questo è oggi il metodo preferito per creare sistemi allo stato dell'arte. Come abbiamo già notato, l'apprendimento presenta un altro vantaggio: permette

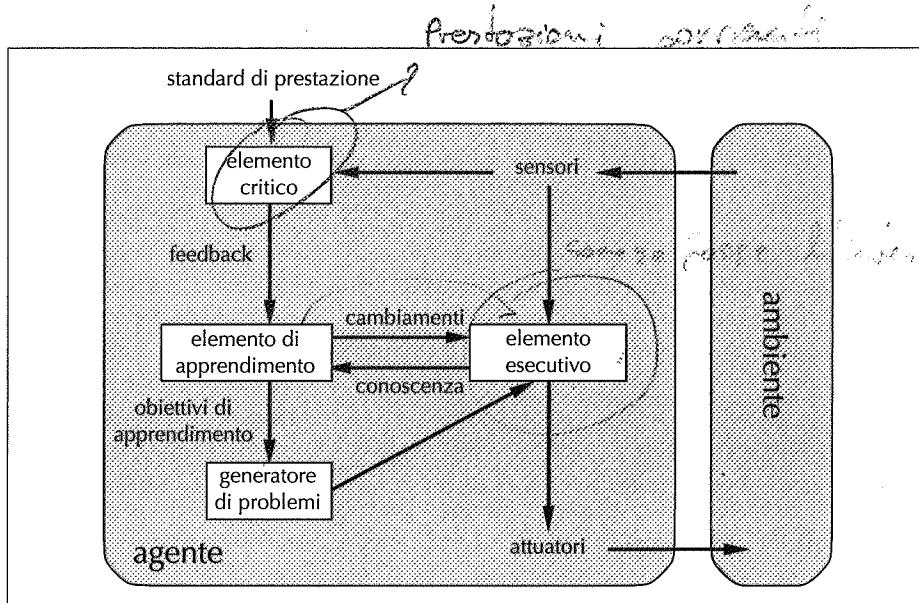


Figura 2.15
Un modello generale di agente capace di apprendere.

agli agenti di operare in ambienti inizialmente sconosciuti, diventando col tempo più competenti di quanto fossero all'inizio, allorché si basavano sulla sola conoscenza iniziale. In questo paragrafo presenteremo brevemente i concetti principali dell'apprendimento. Quasi tutti i prossimi capitoli includeranno commenti sulla possibilità e i metodi per l'apprendimento in particolari categorie di agenti. Nella Parte VI analizzeremo in modo approfondito gli algoritmi stessi.

Un agente capace di apprendere può essere diviso in quattro componenti astratti, come si vede nella Figura 2.15. La distinzione più importante è tra l'elemento di apprendimento (*learning element*), che è responsabile del miglioramento interno, e l'elemento esecutivo (*performance element*), che si occupa della selezione delle azioni esterne. Quest'ultimo è ciò che abbiamo considerato fin qui come se costituisse l'intero agente: prende in input le percezioni e decide le azioni. L'elemento di apprendimento utilizza informazione proveniente dall'elemento critico riguardo le prestazioni correnti dell'agente e determina se e come modificare l'elemento esecutivo affinché in futuro si comporti meglio.

Il progetto dell'elemento di apprendimento dipende molto da quello dell'elemento esecutivo. Quando si cerca di progettare un agente che impara a svolgere una certa attività, la prima domanda da porsi non è "come faccio a fargli imparare questa cosa?" ma "quale tipo di elemento esecutivo permetterà al mio agente di fare questa cosa, una volta che l'avrà imparata?". Qualsiasi progetto di agente può essere migliorato in ogni sua parte dall'aggiunta di meccanismi di apprendimento.

L'elemento critico dice a quello di apprendimento come si sta comportando l'agente rispetto a uno standard di prestazione prefissato. Quest'elemento è neces-

elemento di apprendimento
elemento esecutivo

elemento critico

sario perché le percezioni, in sé, non forniscono alcuna indicazione del successo dell'agente. Un programma di scacchi potrebbe ricevere una percezione che indica che ha dato scacco matto all'avversario, ma ha bisogno di uno standard di prestazione per sapere che questa è una cosa buona; la percezione da sola non basta. È importante che lo standard sia prefissato: concettualmente lo si può pensare come un'entità del tutto separata dall'agente, dato che quest'ultimo non lo deve modificare per adattarlo al suo comportamento.

generatore di problemi

L'ultimo componente di un agente capace di apprendere è il generatore di problemi, il cui scopo è suggerire azioni che portino a esperienze nuove e significative. L'idea è che se si lasciasse mano libera all'elemento esecutivo, esso continuerebbe a ripetere le azioni che ritiene migliori date le conoscenze attuali. Ma se l'agente è disposto a esplorare qualche altra possibilità, e magari eseguire nel breve termine qualche azione subottima, potrebbe scoprire l'esistenza di azioni molto superiori a lungo termine. Scopo del generatore di problemi è di suggerire tali azioni esplorative. Questo è ciò che fanno gli scienziati quando svolgono esperimenti: Galileo non pensava che far cadere sassi dalla cima di una torre, a Pisa, fosse un'azione valida in sé. Non stava cercando di rompere i sassi, né di modificare il cervello di qualche sfortunato passante. Lo scopo era modificare il suo stesso cervello, formulando una teoria migliore sul moto degli oggetti.

Per dare concretezza a questo progetto generale, torniamo a considerare il taxi automatico. L'elemento esecutivo consiste nella collezione di conoscenze e procedure usate dal taxi per selezionare le possibili azioni di guida. Usando quest'elemento il taxi si mette in viaggio e guida. L'elemento critico osserva il mondo e passa delle informazioni a quello di apprendimento. Ad esempio, dopo che il taxi ha svolto a sinistra attraversando senza preavviso tre corsie piene di traffico, il critico osserva il linguaggio scurrile usato dagli altri guidatori. Da quest'esperienza l'elemento di apprendimento può formulare una regola che stabilisce che questa è stata una cattiva azione, di conseguenza l'elemento esecutivo viene modificato con l'inserimento di questa nuova regola. Il generatore di problemi potrebbe identificare certe aeree di comportamento che necessitano di qualche miglioria e suggerire esperimenti, come provare i freni su differenti superfici stradali e in diverse condizioni.

L'elemento di apprendimento può modificare uno qualsiasi dei componenti "di conoscenza" mostrati nei diagrammi degli agenti (Figure 2.9, 2.11, 2.13 e 2.14). Nei casi più semplici l'apprendimento scaturisce direttamente dalla sequenza percettiva. L'osservazione di coppie di stati successivi dell'ambiente permette all'agente di imparare "come evolve il mondo", e l'osservazione dei risultati delle azioni effettuate permette all'agente di imparare "cosa fanno le mie azioni". Ad esempio, se il taxi esercita una pressione eccessiva sui freni mentre sta guidando su una strada bagnata, imparerà ben presto quanta poca decelerazione si può ottenere. Ovviamente le due attività di apprendimento citate diventano più difficili se l'ambiente è solo parzialmente osservabile.

Le forme di apprendimento del precedente paragrafo non hanno bisogno di accedere allo standard di prestazione esterno: in un certo senso lo standard, sempre va-

lido, è quello di formulare predizioni che concordano con l'esperimento. La situazione è leggermente più complessa per un agente basato sull'utilità che desidera aumentare le informazioni riguardanti appunto quest'ultima. Ad esempio, supponiamo che l'agente tassista non riceva mai alcuna mancia dai passeggeri che sono stati rozzamente sballottati per tutto il viaggio. Lo standard di prestazione esterno deve informare l'agente che la perdita di tutte le mance rappresenta un contributo negativo alla sua performance globale; l'agente potrebbe allora imparare che la guida violenta non contribuisce alla sua utilità. In un certo senso, lo standard di prestazione caratterizza una parte delle percezioni entranti come ricompense (o rispettivamente penalità) che rappresentano un feedback diretto sulla qualità del comportamento dell'agente. In questo modo si possono comprendere standard di prestazione predefiniti, quali il dolore e la fame negli animali.

Riassumendo, gli agenti sono formati da una varietà di componenti che possono essere rappresentati in molti modi nel programma agente. Questo fa sì che anche nei metodi di apprendimento sia presente una grande diversità. C'è comunque un tema unificante: l'apprendimento in un agente intelligente può essere definito come il processo che modifica ogni suo componente affinché si accordi meglio con l'informazione di feedback disponibile, migliorando così le prestazioni globali dell'agente.

2.5 Riepilogo

Questo capitolo ha voluto presentare una sorta di panoramica estremamente succinta dell'IA, che abbiamo definito come la scienza della progettazione di agenti. I punti più importanti da ricordare sono i seguenti.

- ♦ Un agente è qualcosa che percepisce e agisce all'interno di un ambiente. La sua funzione agente specifica l'azione intrapresa in risposta a qualsiasi sequenza di percezioni.
- ♦ La misura di prestazione valuta il comportamento dell'agente in un ambiente. Un agente razionale agisce in modo da massimizzare il valore atteso della misura di performance, data la sequenza percettiva fino a quel momento.
- ♦ La specifica di un task environment, o ambiente operativo, include la misura di prestazione, l'ambiente esterno, gli attuatori e i sensori. Durante la progettazione di un agente il primo passo deve sempre consistere nella specifica più dettagliata possibile dell'ambiente operativo.
- ♦ Gli ambienti operativi possono essere classificati in base a molte proprietà significative. Possono essere completamente o parzialmente osservabili, deterministici o stocastici, episodici o sequenziali, statici o dinamici, discreti o continui, ad agente singolo o multiagente.

- ◆ Il **programma agente** implementa la funzione agente. Esistono diversi progetti base per i programmi agente, che riflettono il tipo di informazione esplicitata e utilizzata nel processo decisionale. I progetti variano in efficienza, compattezza e flessibilità; quello più appropriato per un dato programma agente dipende dalla natura dell'ambiente.
- ◆ Gli **agenti reattivi semplici** rispondono direttamente alle percezioni, mentre gli **agenti reattivi basati su modello** memorizzano uno stato interno per tener traccia degli aspetti del mondo che non sono visibili nelle percezioni correnti. Gli **agenti basati su obiettivi** agiscono per raggiungere tali obiettivi, e gli **agenti basati sull'utilità** cercano di massimizzare la “contentezza” attesa.
- ◆ Tutti gli agenti possono migliorare le loro prestazioni mediante l'**apprendimento**.

Note storiche e bibliografiche

Il ruolo centrale dell'azione nell'intelligenza, ovvero il concetto di ragionamento pratico, risale almeno all'*Etica Nicomachea* di Aristotele. Il ragionamento pratico fu anche l'argomento dell'importante articolo di McCarthy's (1958) *Programs with common sense*. La robotica e la teoria del controllo, per loro stessa natura, si preoccupano principalmente della costruzione di agenti fisici. Il concetto di **controllore** della teoria del controllo è identico a quello di agente nell'IA. In modo forse sorprendente, per la maggior parte della sua storia l'IA si è concentrata su componenti isolati (sistemi di risposta, dimostratori di teoremi, sistemi di visione e così via) piuttosto che sugli agenti nella loro interezza. Un'importante eccezione fu rappresentata dalla discussione sugli agenti nel libro di Genesereth e Nilsson (1987). La visione basata su agenti oggi è ampiamente accettata e costituisce il tema centrale dei libri più recenti (Poole et al., 1998; Nilsson, 1998).

Nel Capitolo 1 abbiamo ritrovato le radici del concetto di razionalità nella filosofia e nell'economia. Nell'IA il concetto rivestì un interesse marginale fino alla metà degli anni '80, quando cominciò a diffondersi in molte discussioni riguardanti le basi tecniche della disciplina. In un articolo, Jon Doyle (1983) predisse che la progettazione di agenti razionali sarebbe arrivata a essere considerata la missione principale dell'IA, mentre altri argomenti popolari se ne sarebbero separati per formare nuove discipline.

L'attenzione verso le proprietà dell'ambiente e le loro conseguenze sulla progettazione di agenti razionali appare con più evidenza nella tradizione della teoria del controllo: ad esempio, i sistemi di controllo classici (Dorf e Bishop, 1999) gestiscono ambienti completamente osservabili e deterministici; il controllo ottimo stocastico (Kumar e Varaiya, 1986) si occupa di ambienti parzialmente osservabili e stocastici; il controllo ibrido (Henzinger e Sastry, 1998) è rivolto agli ambienti

che contengono sia elementi discreti che continui. La distinzione tra ambienti parzialmente e totalmente osservabili è centrale nella letteratura che riguarda la **programmazione dinamica**, sviluppata in seno alla ricerca operativa (Puterman, 1994). Ne discuteremo nel Capitolo 17.

Gli agenti reattivi sono stati il modello principale per gli psicologi behavioristi come Skinner (1953), che tentarono di ridurre totalmente la psicologia degli organismi a una corrispondenza input/output o stimolo/risposta. Il passaggio dal behaviorismo al funzionalismo, che fu almeno parzialmente dovuto all'applicazione della metafora dei computer agli agenti (Putnam, 1960; Lewis, 1966), introdusse lo stato interno degli agenti. La maggior parte degli autori di intelligenza artificiale ritiene che gli agenti reattivi puri, anche dotati di stato, siano troppo semplici per risolvere molti problemi: quest'assunto è stato però messo in discussione dai lavori di Rosenschein (1985) e Brooks (1986). Negli ultimi anni sono stati fatti molti sforzi per trovare algoritmi efficienti capaci di tener traccia di ambienti complessi (Hamscher et al., 1992). Il programma Remote Agent, che controllava il veicolo spaziale Deep Space One descritto a pag. 38, è un esempio particolarmente notevole (Muscettola et al., 1998; Jonsson et al., 2000).

Gli agenti basati su obiettivo sono già presenti nel concetto aristotelico di ragionamento pratico e attraversano la letteratura fino ai primi articoli di McCarthy sull'IA logica. Shakey il Robot (Fikes e Nilsson, 1971; Nilsson, 1984) fu la prima realizzazione robotica di un agente logico basato su obiettivi. Un'analisi completa della logica degli agenti a obiettivi apparì poi in Genesereth e Nilsson (1987), e una metodologia di programmazione basata sugli obiettivi, chiamata *agent-oriented programming*, fu sviluppata da Shoham (1993).

L'approccio basato su obiettivi domina anche la tradizione della psicologia cognitiva nell'area della risoluzione di problemi, a partire da *Human Problem Solving* (Newell e Simon, 1972), che ebbe un'enorme influenza, attraverso tutti i lavori più recenti di Newell (Newell, 1990). Gli obiettivi, ulteriormente suddivisi in *desideri* (generali) e *intenzioni* (perseguite in un dato momento) sono l'aspetto centrale della teoria degli agenti sviluppata da Bratman (1987), che ha avuto molta importanza nei campi della comprensione del linguaggio naturale e dei sistemi multiagente.

Horvitz et al. (1988) suggerisce specificatamente di usare come base dell'IA una razionalità concepita come massimizzazione dell'utilità attesa. Il testo di Pearl (1988) è stato il primo a considerare approfonditamente la probabilità e la teoria dell'utilità; la sua esposizione dei metodi pratici per ragionare e prendere decisioni in condizione di incertezza fu probabilmente il singolo fattore più importante nel rapido spostamento dell'attenzione, negli anni '90, verso gli agenti basati sull'utilità (v. Parte V, 2° vol.).

Il progetto generale per gli agenti capaci di apprendere rappresentato nella Figura 2.15 è un classico della letteratura sull'apprendimento automatico (Buchanan et al., 1978; Mitchell, 1997). Esempi di questo progetto, implementati sotto

forma di software, risalgono almeno al programma di Arthur Samuel (1959, 1967) per giocare a dama. Gli agenti capaci di apprendere sono discussi nella Parte VI.

Negli ultimi anni l'interesse negli agenti e nella loro progettazione è cresciuto rapidamente, anche grazie allo sviluppo di Internet e alla relativa necessità di softbot mobili e automatizzati (Etzioni e Weld, 1994). Gli articoli più importanti sono raccolti in *Readings in Agents* (Huhns e Singh, 1998) e *Foundations of Rational Agency* (Wooldridge e Rao, 1999). *Multiagent Systems* (Weiss, 1999) fornisce solide basi per molti aspetti della progettazione di agenti. I congressi dedicati agli agenti includono la International Conference on Autonomous Agents, l'International Workshop on Agent Theories, Architectures, and Languages e l'International Conference on Multiagent Systems. Infine, *Dung Beetle Ecology* (Hanski e Cambefort, 1991) fornisce una quantità di informazioni interessanti sul comportamento degli scarabei stercorari.

Esercizi

- 2.1 Definite con parole vostre i seguenti termini: agente, funzione agente, programma agente, razionalità, autonomia, agente reattivo, agente basato su modello, agente basato su obiettivi, agente basato sull'utilità, agente capace di apprendere.
- 2.2 Sia la misura di prestazione che la funzione di utilità misurano quanto bene si sta comportando un agente. Spiegate la differenza tra le due.
- 2.3 Quest'esercizio indaga le differenze tra funzioni agente e programmi agente.
 - a. Può esserci più di un programma agente che implementa la stessa funzione agente? Fornite un esempio o mostrate perché ciò è impossibile.
 - b. Esistono funzioni agente che non possono essere implementate da alcun programma agente?
 - c. Data un'architettura prefissata della macchina, ne risulta che ogni programma agente implementa esattamente una funzione agente?
 - d. Data un'architettura con n bit di memoria, quanti sono i possibili programmi agente?
- 2.4 Ora esaminiamo la razionalità di varie funzioni agente per l'aspirapolvere.
 - a. Mostrate che la semplice funzione agente descritta nella Figura 2.3 è effettivamente razionale date le ipotesi elencate a pag. 50.
 - b. Descrivete una funzione agente razionale per la misura di prestazione modificata che sottrae un punto per ogni movimento. L'agente corrispondente richiederà la presenza di uno stato interno?

- c. Discutete i possibili progetti per l'agente nei casi in cui i riquadri puliti possano diventare sporchi e la geografia dell'ambiente sia sconosciuta. Ha senso che in questi casi l'agente sia capace di apprendere dall'esperienza? Se è così, che cosa dovrebbe apprendere?
- 2.5 Per ognuno dei seguenti agenti, sviluppate una descrizione PEAS dell'ambiente operativo:
- calciatore robotico
 - agente per l'acquisto di libri su Internet
 - veicolo autonomo per l'esplorazione della superficie marziana
 - sistema di assistenza alla dimostrazione di teoremi per matematici.
- 2.6 Per ognuno dei tipi di agente dell'Esercizio 2.5, caratterizzate l'ambiente secondo le proprietà elencate nel Paragrafo 2.3 e scegliete un progetto adeguato per ogni agente.

I seguenti esercizi riguardano tutti l'implementazione di ambienti e agenti per il mondo dell'aspirapolvere.

- 2.7 Implementate un simulatore di ambienti che misuri le prestazioni degli agenti all'interno del mondo dell'aspirapolvere illustrato nella Figura 2.2 e specificato a pag. 50. La vostra implementazione dovrebbe essere modulare, in modo che i sensori, gli attuatori e le caratteristiche dell'ambiente (dimensioni, forma, posizione dello sporco, ecc.) possano essere cambiate facilmente (*nota*: per alcune combinazioni di linguaggi di programmazione e sistemi operativi sono già presenti implementazioni nel nostro deposito online di codice).
- 2.8 Implementate un semplice agente reattivo per l'ambiente descritto nell'Esercizio 2.7. Lanciate il simulatore di ambienti con quest'agente provando tutte le possibili configurazioni iniziali di sporco e posizione iniziale dell'agente. Registrate i punteggi dell'agente per ogni configurazione e il suo punteggio medio.
- 2.9 Considerate una versione modificata dell'ambiente dell'Esercizio 2.7 in cui l'agente sia penalizzato di un punto ogni volta che si muove.
- Un agente reattivo semplice può essere perfettamente razionale in quest'ambiente? Spiegate.
 - E un agente reattivo dotato di stato interno? Progettate.
 - Come cambiano le vostre risposte ai punti a e b se le percezioni dell'agente sono in grado di dirgli se ogni riquadro dell'ambiente è sporco o pulito?



- 2.10 Considerate una versione modificata dell’ambiente dell’Esercizio 2.7 nella quale la geografia dell’ambiente (estensione, confini, ostacoli) sia sconosciuta, così come la configurazione iniziale dello sporco (l’agente può andare *Su* e *Giù* oltre che a *Sinistra* e *Destra*).
- a. Un agente reattivo semplice può essere perfettamente razionale in quest’ambiente? Spiegate.
 - b. Un agente reattivo semplice con una funzione agente *randomizzata* può comportarsi meglio dell’agente del punto a? Progettate un agente siffatto e misurate le sue prestazioni all’interno di ambienti diversi.
 - c. Potete progettare un ambiente in cui le prestazioni dell’agente randomizzato siano particolarmente basse? Mostrate i vostri risultati.
 - d. Un agente reattivo dotato di stato può comportarsi meglio di un agente reattivo semplice? Progettate un agente siffatto e misurate le sue prestazioni all’interno di ambienti diversi. Potete progettare un agente razionale di questo tipo?
- 2.11 Ripetete l’Esercizio 2.10 nel caso in cui il sensore della posizione sia rimpiattato da un “paraurti” che percepisce i tentativi dell’agente di muoversi fuori dai confini dell’ambiente o contro un ostacolo. Supponete che questo sensore smetta di funzionare; come si dovrebbe comportare l’agente?
- 2.12 Tutti gli ambienti negli esercizi precedenti erano deterministici. Discutete i possibili programmi agente in ognuna delle seguenti versioni stocastiche.
- a. Legge di Murphy: il 25% delle volte, l’azione *Aspira* non riesce a pulire il pavimento se questo è sporco e addirittura lo sporca se è pulito. In che modo cambia il vostro programma agente se il sensore dello sporco dà una segnalazione sbagliata il 10% delle volte?
 - b. Bambini piccoli: a ogni passo temporale, ogni riquadro pulito ha il 10% di possibilità di diventare sporco. Potete escogitare un progetto di agente razionale per questo caso?

Risoluzione di problemi

Problem Solving

Parte Seconda

Capitolo 3

Risolvere i problemi con la ricerca

Nel quale vediamo come un agente può trovare una sequenza di azioni che raggiunge i suoi scopi, quando nessuna azione singola è sufficiente.

Gli agenti più semplici che abbiamo descritto nel Capitolo 2 sono quelli reattivi, che basano le loro azioni unicamente sullo stato corrente. Questi agenti non possono operare bene negli ambienti in cui la corrispondenza tra stati e azioni diventa troppo grande per essere memorizzata e troppo complessa da imparare. Gli agenti basati su obiettivi, d'altra parte, possono aver successo considerando le azioni future e valutando le loro conseguenze.

Questo capitolo descrive un tipo particolare di agente basato su obiettivi, chiamato agente risolutore di problemi. Questi agenti decidono cosa fare trovando sequenze di azioni che conducono a stati desiderabili. Cominceremo col definire precisamente gli elementi che costituiscono un "problema" e la sua "soluzione", fornendo diversi esempi per chiarimento. Descriveremo poi diversi algoritmi di ricerca di uso generale, confrontando i vantaggi di ognuno di essi. Gli algoritmi sono non informati, nel senso che non possiedono alcuna informazione specifica riguardante il problema oltre la sua definizione. Il Capitolo 4 si occuperà degli algoritmi di ricerca informati, che invece sanno dove cercare le soluzioni.

Questo capitolo utilizza nozioni di analisi degli algoritmi: i lettori che non hanno familiarità con i concetti di complessità asintotica (la notazione $O()$) e di NP-completezza dovrebbero prima leggere l'Appendice A.

agente risolutore di problemi

non informati

3.1 Agenti risolutori di problemi

Gli agenti intelligenti devono massimizzare la loro misura di prestazione. Come abbiamo detto nel Capitolo 2, talvolta questo è più facile se l'agente può scegliere un obiettivo e cercare di soddisfarlo. Vediamo innanzitutto come e perché potrebbe farlo.

Immaginiamo un agente che si sta godendo una vacanza nella città di Arad, in Romania. La sua misura di prestazione è composta da molti fattori: vuole farsi una bella tintarella, migliorare il suo rumeno, ammirare i panorami più belli, passare nottate divertenti, evitare i postumi di sbronza e così via. Il problema è complesso e richiede molti compromessi, nonché la lettura accurata della guida turistica. Ora supponiamo che l'agente abbia un biglietto aereo non rimborsabile per la partenza da Bucarest il giorno successivo. In questo caso, sembra sensato che l'agente adotti l'obiettivo di arrivare a Bucarest. Le sequenze di azioni che non portano a raggiungere Bucarest in tempo possono essere scartate senza altre considerazioni, cosicché il problema dell'agente risulta molto semplificato. Gli obiettivi aiutano a organizzare il comportamento limitando gli scopi che l'agente sta cercando di raggiungere. La formulazione dell'obiettivo, basata sulla situazione corrente e sulla misura di prestazione dell'agente, è il primo passo nella risoluzione di problemi.

Un obiettivo è composto da un insieme di stati del mondo: esattamente tutti e soli quegli stati in cui l'obiettivo è soddisfatto. Il compito dell'agente è trovare la sequenza di azioni che lo farà giungere in uno stato obiettivo. Prima di poter fare ciò, l'agente deve decidere quali tipi di azioni e stati prendere in considerazione. Se cercasse di considerare azioni al livello di "muovi il piede sinistro avanti di un centimetro", oppure "gira il volante a sinistra di un grado", l'agente con tutta probabilità non riuscirebbe neppure a uscire dal parcheggio, meno che meno arrivare a Bucarest: a quel grado di dettaglio c'è troppa incertezza nel mondo e una soluzione sarebbe composta da un numero eccessivo di passi. La formulazione del problema è il processo che porta a decidere, dato un obiettivo, quali azioni e stati considerare. Discuteremo più avanti questo processo: per ora, limitiamoci a presumere che l'agente esaminerà azioni al livello di guidare da una grande città a un'altra. Gli stati considerati, quindi, corrisponderanno a trovarsi in un particolare città.¹

Adesso il nostro agente ha adottato l'obiettivo di guidare fino a Bucarest, e sta valutando dove andare partendo da Arad. Ci sono tre strade che conducono fuori città: una verso Sibiu, una verso Timisoara e la terza verso Zerind. Nessuna di queste soddisfa l'obiettivo per cui l'agente, a meno che non abbia particolare familiari-

formulazione
dell'obiettivo

formulazione del
problema

¹ Notate che ognuno di questi "stati" in realtà corrisponde a un grande insieme di stati del mondo, perché questi ultimi contengono ogni aspetto della realtà. È importante tenere ben presente la differenza tra gli stati del mondo e quelli del problema.

rità con la geografia rumena, non saprà quale scegliere.² In altre parole, l'agente non saprà quale delle tre possibili azioni è preferibile, perché non possiede abbastanza informazioni sullo stato risultante da ognuna di esse. Se l'agente non avesse ulteriore conoscenza, sarebbe bloccato, e non potrebbe fare di meglio che scegliere a caso.

Ma supponiamo che l'agente possieda una mappa della Romania, su carta o nella sua memoria. Lo scopo di una mappa è fornire all'agente informazioni riguardo agli stati in cui si potrebbe trovare e alle sue possibili azioni. L'agente può usare queste informazioni per considerare i passi successivi di un viaggio ipotetico attraverso ognuna delle tre città, cercando di trovare un cammino che lo conduca finalmente a Bucarest. Una volta trovato sulla mappa un percorso che va da Arad a Bucarest, potrà raggiungere il suo obiettivo eseguendo le azioni di guida che corrispondono ai singoli passi sulla mappa. In generale, un agente che ha a disposizione diverse opzioni immediate di valore sconosciuto può decidere cosa fare esaminando diverse possibili sequenze di azioni che portano a stati di valore conosciuto, scegliendo quindi la sequenza migliore.

Questo processo di selezione è detto ricerca. Un algoritmo di ricerca prende un problema come input e restituisce una soluzione sotto forma di una sequenza di azioni. Una volta trovata una soluzione, l'agente può eseguire le azioni raccomandate: questa fase prende il nome di esecuzione. Il progetto dell'agente, quindi, ha la semplice struttura "formulazione, ricerca, esecuzione" mostrata nella Figura 3.1. Dopo aver formulato un obiettivo e un problema da risolvere, l'agente invoca una procedura di ricerca. A questo punto, usa la soluzione come guida per le sue azioni, facendo quello che la soluzione raccomanda come passo successivo (tipicamente la prima azione della sequenza) e rimuovendo poi quel passo dalla sequenza stessa. Una volta eseguita l'intera soluzione l'agente formulerà un nuovo obiettivo.

Ora descriveremo per prima cosa il processo di formulazione del problema, dedicando il resto del capitolo ai vari algoritmi per la funzione RICERCA. In questo capitolo non discuteremo ulteriormente le funzioni AGGIORNA-STATO e FORMULA-OBIETTIVO.

Prima di addentrarci nei dettagli facciamo una breve pausa per vedere dove si collocano gli agenti risolutori di problemi nella discussione del Capitolo 2 riguardante agenti e ambienti. Il progetto di agente nella Figura 3.1 dà per scontato che l'ambiente sia statico, perché la formulazione e la soluzione del problema sono eseguite senza preoccuparsi che nel frattempo l'ambiente possa cambiare. Il progetto



ricerca
soluzione

esecuzione

² Presumiamo che gran parte dei lettori si trovino nelle stesse condizioni del nostro agente e si possano facilmente immaginare altrettanto sperduti. Chiediamo scusa ai lettori rumeni, che non potranno trarre vantaggio da questo espediente pedagogico.

function SEMPLICE-AGENTE-RISOLTORE-PROBLEMI(*percezione*) **returns** un'azione

inputs: *percezione*, una percezione

static: *seq*, una sequenza di azioni, inizialmente vuota

stato, una descrizione dello stato corrente del mondo

goal, un obiettivo, inizialmente null

problema, una formulazione di problema

stato \leftarrow AGGIORNA-STATO(*stato*, *percezione*)

if *seq* è vuota **then do**

goal \leftarrow FORMULA-OBIETTIVO(*stato*)

problema \leftarrow FORMULA-PROBLEMA(*stato*, *goal*)

seq \leftarrow RICERCA(*problema*)

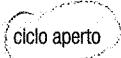
azione \leftarrow PRIMO-ELEMENTO(*seq*)

seq \leftarrow CODA(*seq*)

return *azione*

Figura 3.1 Un semplice agente risolutore di problemi. Per prima cosa l'agente formula un obiettivo e un problema, quindi cerca una sequenza di azioni in grado di risolverlo, per eseguire infine le azioni una per volta. Quando il ciclo è completo, l'agente formula un altro obiettivo e ricomincia. Notate che, durante l'esecuzione delle azioni, le percezioni vengono ignorate: l'agente dà per scontato che la soluzione trovata funzionerà sempre.

presume anche che lo stato iniziale sia conosciuto; ciò è facile se l'ambiente è osservabile. L'idea di esaminare "sequenze di azioni alternative" porta all'assunto che l'ambiente possa essere considerato discreto. Infine, e questo è l'aspetto più importante, il nostro progetto presuppone che l'ambiente sia deterministico. Le soluzioni sono sequenze uniche di azioni, per cui non possono gestire eventi inaspettati; inoltre sono eseguite senza nessuna attenzione alle percezioni! Un agente che attua i propri piani a occhi chiusi, per così dire, dev'essere ben sicuro di ciò che lo circonda (nella teoria del controllo questo sistema sarebbe chiamato a ciclo aperto, perché ignorare le percezioni spezza il ciclo di retroazione tra l'agente e l'ambiente). Tutti questi assunti significano che stiamo muovendoci nel più facile degli ambienti, e questa è la ragione per cui questo capitolo si trova all'inizio del libro. Nel Paragrafo 3.6 esamineremo brevemente quello che succede quando si rilassano le ipotesi di osservabilità e determinismo: i Capitoli 12 e 17 andranno molto più a fondo.



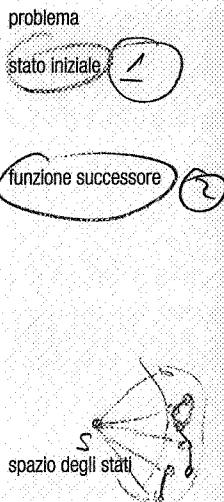
ciclo aperto

Problemi ben definiti e soluzioni

Un problema può essere definito formalmente da quattro componenti.

- ♦ Lo stato iniziale in cui si trova l'agente. Ad esempio, lo stato iniziale del nostro agente in Romania potrebbe essere descritto come *In(Arad)*.
- ♦ Una descrizione delle azioni possibili dell'agente. La formulazione più comune³ usa una funzione successore. Dato uno stato x , $\text{FUNZIONE-SUCCESSORE}(x)$ restituisce un insieme di coppie ordinate $\langle \text{azione}, \text{successore} \rangle$, ove ogni azione è una di quelle legali nello stato x e ogni successore è lo stato che può essere raggiunto da x eseguendo tale azione. Ad esempio, partendo dallo stato *In(Arad)*, la funzione successore per il problema del turista in Romania restituirebbe

$$\{\langle \text{Go}(Sibiu), \text{In}(Sibiu) \rangle, \langle \text{Go}(Timisoara), \text{In}(Timisoara) \rangle, \langle \text{Go}(Zerind), \text{In}(Zerind) \rangle\}$$

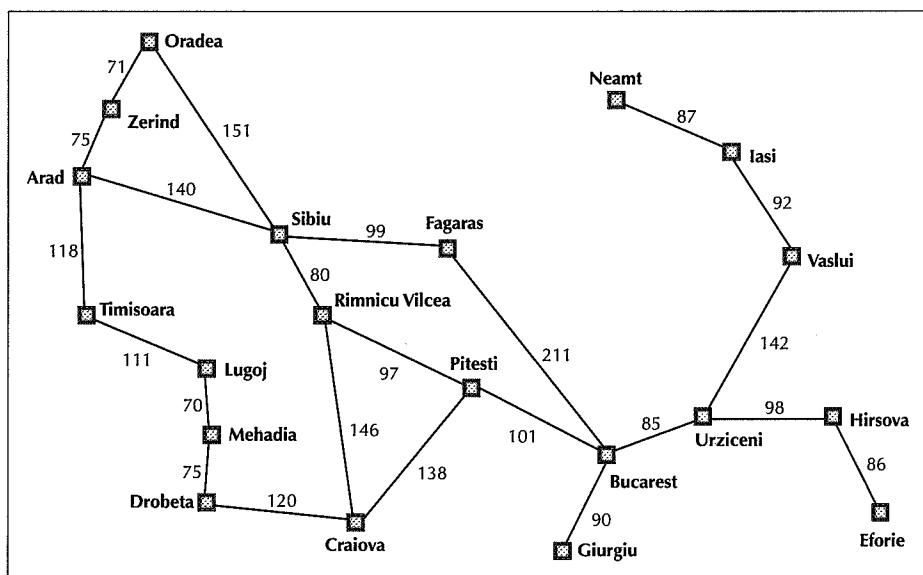


Insieme, lo stato iniziale e la funzione successore definiscono implicitamente lo spazio degli stati del problema, ovvero l'insieme di tutti gli stati raggiungibili a partire da quello iniziale. Lo spazio degli stati forma un grafo in cui i nodi rappresentano degli stati e gli archi delle azioni (la mappa della Romania mostrata nella Figura 3.2 può essere interpretata come un grafo dello spazio degli stati se consideriamo ogni strada come indicatore di due possibili azioni di guida, una per ogni direzione). Un cammino nello spazio degli stati è una sequenza di stati collegati da una sequenza di azioni.

- ♦ Il test obiettivo, che determina se un particolare stato è uno stato obiettivo. Talvolta esiste un insieme esplicito di possibili stati obiettivo, e il test si limita a verificare se quello dato appartiene all'insieme. L'obiettivo del turista in Romania è l'insieme singololetto *In(Bucarest)*. Altre volte l'obiettivo è specificato da una proprietà astratta e non da un insieme di stati esplicitamente enumerati. Ad esempio, negli scacchi, l'obiettivo è raggiungere lo stato chiamato "scacco matto", in cui il re avversario è sottoposto a uno scacco a cui non può sfuggire.
- ♦ La funzione costo di cammino, che assegna un costo numerico a ogni cammino. L'agente risolutore di problemi sceglie una funzione di costo che rispecchia la sua misura di prestazione. Per l'agente che cerca di arrivare a Bucarest, la cosa più importante è il tempo, cosicché il costo di un cammino potrebbe essere indicato dalla sua lunghezza in chilometri. In questo capitolo

³ Una formulazione alternativa usa un insieme di operatori che possono essere applicati a uno stato per generare i successori.

Figura 3.2
Una mappa stradale semplificata della Romania.



costo di passo

presumeremo sempre che il costo di un cammino possa essere definito come la somma dei costi delle azioni individuali che lo compongono. Il costo di passo corrispondente all'azione a che fa passare dallo stato x allo stato y è indicato $c(x, a, y)$. I costi dei passi in Romania sono indicati nella Figura 3.2 sotto forma di distanze chilometriche tra le città. Presumeremo sempre che i costi dei passi non siano mai negativi.⁴

I quattro elementi qui sopra definiscono un problema e possono essere riuniti insieme in una singola struttura dati passata come input a un algoritmo di risoluzione di problemi. Una soluzione di un problema è un cammino dallo stato iniziale a uno stato obiettivo. La qualità della soluzione è misurata dalla funzione di costo di cammino, e la soluzione ottima è quella che ha il costo minore di tutte.

soluzione ottima

La formulazione dei problemi

Nel paragrafo precedente abbiamo proposto una formulazione del problema di arrivare a Bucarest in termini di stato iniziale, funzione successore, test obiettivo e costo di cammino. Questa formulazione sembra ragionevole, eppure non definisce gran parte degli aspetti del mondo reale. Confrontate la semplice descrizione di stato che abbiamo scelto, $In(Arad)$, a un vero viaggio in macchina attraverso un paese,

⁴ Le conseguenze dell'eventuale presenza di costi negativi sono esaminate nell'Esercizio 3.17.

in cui lo stato del mondo include così tante cose: i compagni di viaggio, cosa c'è alla radio, il paesaggio fuori dal finestrino, se ci sono poliziotti nelle vicinanze, tra quanto tempo c'è la prossima area di servizio, la condizione della strada, le condizioni del tempo e così via. Tutte queste considerazioni sono trascurate nelle nostre descrizioni di stato perché sono irrilevanti al fine di trovare la strada per Bucarest. Il processo di rimozione dei dettagli da una rappresentazione prende il nome di astrazione.

Oltre ad astrarre le descrizioni di stato, dobbiamo astrarre le stesse azioni. Un'azione di guida ha molti effetti. Oltre a cambiare la posizione del veicolo e dei suoi occupanti, richiede tempo, consuma carburante, genera inquinamento e cambia lo stesso agente (come si dice, viaggiare apre la mente). Nella nostra formulazione abbiamo preso in considerazione solamente il cambiamento di posizione. Inoltre abbiamo proprio omesso di citare un gran numero di azioni: accendere la radio, guardare fuori dal finestrino, rallentare quando c'è una macchina della polizia e così via. E ovviamente non abbiamo specificato azioni a un livello di dettaglio come "gira il volante a sinistra di tre gradi".

È possibile essere più precisi sulla definizione di un livello di astrazione appropriato? Pensate che gli stati e le azioni astratte che abbiamo scelto corrispondano a grandi insiemi di stati del mondo e di sequenze di azioni dettagliate. Ora considerate una soluzione al problema astratto: ad esempio, il cammino da Arad a Sibiu, quindi a Rimnicu Vilcea e da lì a Pitesti, per arrivare infine a Bucarest. Questa soluzione astratta corrisponde a un gran numero di cammini più dettagliati. Ad esempio, potremmo guidare con la radio accesa tra Sibiu e Rimnicu Vilcea e poi spegnerla per il resto del viaggio. L'astrazione è valida se possiamo espandere ogni soluzione astratta in una soluzione nel mondo più dettagliato; una condizione sufficiente è che per ogni stato dettagliato "in Arad" ci sia un cammino dettagliato che porta a qualche stato che è "in Sibiu" e così via. L'astrazione è utile se eseguire ogni azione nella soluzione è più facile che nel problema originale; in questo caso le azioni sono abbastanza facili da poter essere eseguite da un agente guidatore medio senza ulteriore ricerca o pianificazione. La scelta di una buona astrazione prevede quindi che si rimuova quanto più dettaglio possibile mantenendo la validità e assicurandosi che le azioni astratte siano facili da eseguire. Se non fosse per la capacità di costruire astrazioni utili, gli agenti intelligenti sarebbero completamente soverchiati dalla complessità del mondo reale.

astrazione

3.2 Problemi esemplificativi

La risoluzione di problemi è stata applicata a una vasta gamma di task environment. Ne indicheremo ora qualcuno tra i più noti, distinguendo tra problem giocattolo e problem del mondo reale. Lo scopo di un problema giocattolo (*toy problem*) è illustrare o mettere alla prova diversi metodi di risoluzione di problemi.

problema giocattolo

problema del mondo reale

Può sempre essere descritto in modo preciso e sintetico: questo significa che può essere usato facilmente da ricercatori diversi per confrontare le prestazioni degli algoritmi. Un problema del mondo reale è uno di quelli le cui soluzioni interessano effettivamente alla gente. Questi problemi tendono a non avere una descrizione singola adottata da tutti: cercheremo di trasmettere il senso generale delle loro formulazioni.

Problemi giocattolo

Il primo esempio che esamineremo è il mondo dell'aspirapolvere presentato inizialmente nel Capitolo 2 (v. Figura 2.2). Può essere formulato come problema come segue.

- ◆ **Stati:** l'agente si trova in uno di due possibili riquadri, ognuno dei quali può contenere sporco oppure no. Quindi ci sono $2 \times 2^2 = 8$ possibili stati del mondo.
- ◆ **Stato iniziale:** ogni stato può essere designato come stato iniziale.
- ◆ **Funzione successore:** genera gli stati legali che risultano dal tentativo di intraprendere le tre azioni (*Sinistra*, *Destra* e *Aspira*). Lo spazio degli stati completo è mostrato nella Figura 3.3.
- ◆ **Test obiettivo:** controlla se entrambi i riquadri sono puliti.
- ◆ **Costo di cammino:** ogni passo costa 1, cosicché il costo di ogni cammino è pari al numero di passi che lo compongono.

Confrontato con il mondo reale questo problema giocattolo ha locazioni discrete, sporco discreto, aspirazione affidabile, e i riquadri puliti non diventano mai più sporchi (nel Paragrafo 3.6 rilasseremo queste ipotesi). È importante notare che lo stato è determinato sia dalla posizione dell'agente che da quella dello sporco. Un ambiente più grande, con n locazioni, avrebbe $n \cdot 2^n$ stati.

rompicapo a 8 tasselli

Il rompicapo a 8 tasselli, di cui è rappresentata un'istanza nella Figura 3.4, consiste in una plancia 3×3 con otto tasselli numerati e uno spazio vuoto. Un tassello adiacente allo spazio vuoto può scivolare nella posizione libera. Lo scopo è raggiungere un particolare stato obiettivo, come quello mostrato a destra nella figura. La formulazione standard è la seguente.

- ◆ **Stati:** la descrizione di ogni stato specifica la posizione di ognuno degli otto tasselli e dello spazio vuoto.
- ◆ **Stato iniziale:** ogni stato può essere designato come stato iniziale. Notate che, qualsiasi sia lo stato obiettivo, questo potrà essere raggiunto da esattamente metà dei possibili stati iniziali (Esercizio 3.4).
- ◆ **Funzione successore:** genera gli stati legali che risultano dal tentativo di eseguire le quattro possibili azioni (lo spazio vuoto si muove a *Sinistra*, *Destra*, *Su* o *Giù*).

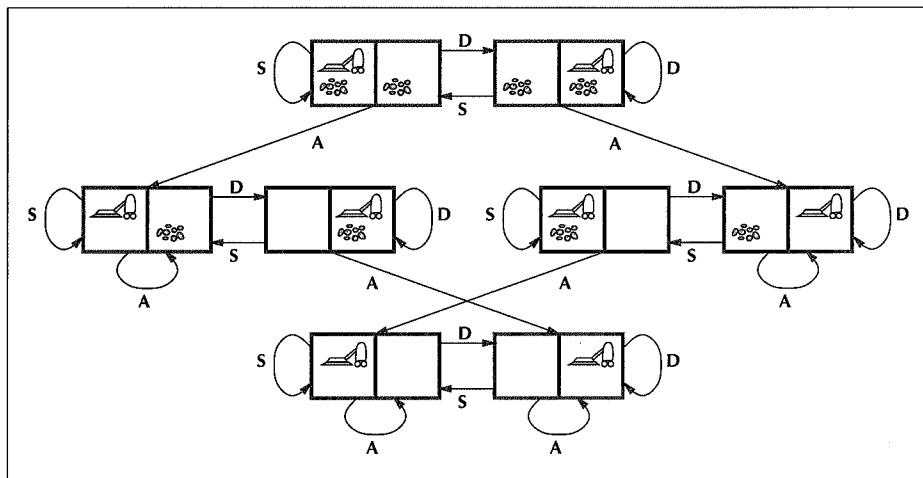


Figura 3.3
Lo spazio degli stati per il mondo dell'aspirapolvere.
Gli archi rappresentano azioni:
S = Sinistra,
D = Destra,
A = Aspira.

- ◆ **Test obiettivo:** verifica se lo stato corrente corrisponde alla configurazione obiettivo mostrata nella Figura 3.4 (naturalmente sono possibili altre configurazioni obiettivo).
- ◆ **Costo di cammino:** ogni passo costa 1, cosicché il costo di ogni cammino è pari al numero di passi che lo compongono.

Quali sono adesso le astrazioni? Le azioni considerano solo gli stati iniziali e finali, ignorando le posizioni intermedie in cui il blocco sta ancora scivolando. Abbiamo eliminato completamente azioni come scuotere la plancia quando i pezzi si incassano, oppure tirare fuori tutti i pezzi con un coltellino e rimetterli dentro. Quello che rimane è la descrizione delle regole del rompicapo, senza nessun riferimento ai dettagli della manipolazione fisica.

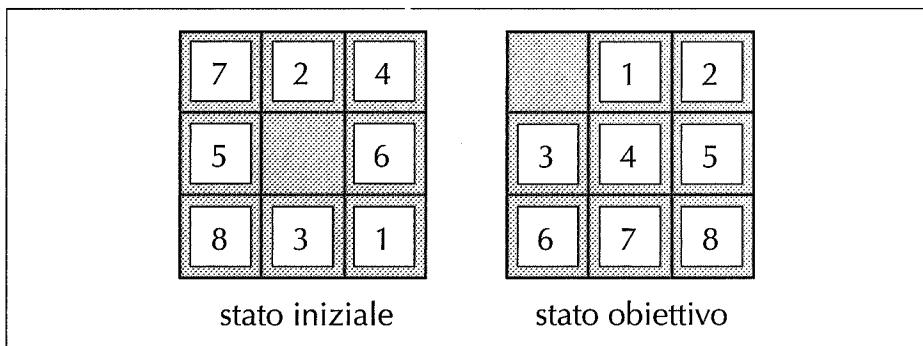


Figura 3.4
Una tipica istanza di rompicapo a 8 tasselli.

90 Capitolo 3 Risolvere i problemi con la ricerca

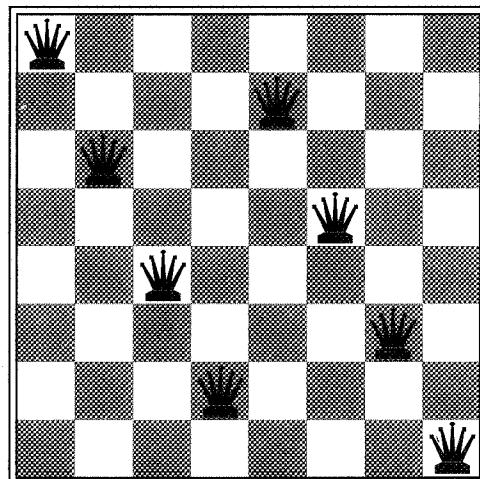
rompicapi a tasselli
mobiliproblema delle 8
regineformulazione
incrementaleformulazione a stato
completo

Il problema che abbiamo appena descritto appartiene alla famiglia dei **rompicapi a tasselli mobili**, usati spesso per il test dei nuovi algoritmi di ricerca. Questa classe generale di problemi è nota per essere NP-completa, per cui non ci si aspetta di trovare metodi molto migliori degli algoritmi di ricerca descritti in questo capitolo e nel prossimo. Il rompicapo a 8 tasselli ha $9!/2 = 181.440$ stati raggiungibili ed è facilmente risolvibile. Quello a 15 tasselli (che usa una plancia 4×4) ha circa 1,3 trilioni di stati, e può essere risolto dai migliori algoritmi di ricerca in pochi millisecondi. Il rompicapo a 24 tasselli (su una plancia 5×5) ha circa 10^{25} stati, e una sua istanza casuale è ancora piuttosto difficile da risolvere in modo ottimo con gli odierni computer e algoritmi.

Lo scopo del **problema delle 8 regine** è di piazzare otto regine su una scacchiera in modo tale che nessuna ne attacchi un'altra (negli scacchi una regina attacca i pezzi sulla sua stessa riga, più propriamente chiamata "traversa", sulla stessa colonna e sulle diagonali). La Figura 3.5 mostra un tentativo fallito di soluzione: la regina nella colonna più a destra è attaccata da quella nell'angolo in alto a sinistra.

Benché esistano algoritmi specializzati abbastanza efficienti per questo problema e per l'intera famiglia a n regine, esso rimane un interessante caso di test per gli algoritmi di ricerca. Ci sono due principali categorie di formulazione. Una **formulazione incrementale** utilizza operatori che estendono progressivamente la descrizione di stato, cominciando dallo stato vuoto; per il problema delle 8 regine, questo significa che ogni azione aggiunge una regina alla scacchiera. La **formulazione a stato completo** comincia con le otto regine già sulla scacchiera e le sposta. In ognuno dei due casi il costo di cammino non è rilevante, perché interessa solo lo stato finale. Una prima formulazione incrementale potrebbe essere la seguente.

Figura 3.5
Una "quasi soluzione" del problema delle 8 regine (la soluzione è lasciata al lettore come esercizio).



- ◆ **Stati:** ogni piazzamento sulla scacchiera di un numero da 0 a 8 regine è uno stato.
- ◆ **Stato iniziale:** la scacchiera è vuota.
- ◆ **Funzione successore:** aggiunge una regina in una casa vuota.
- ◆ **Test obiettivo:** sulla scacchiera ci sono 8 regine e nessuna è attaccata.

In questa formulazione abbiamo $64 \cdot 63 \cdots 57 \approx 3 \times 10^{14}$ possibili sequenze da investigare. Una formulazione migliore proibirebbe il posizionamento di una regina in una casella già sottoposta ad attacco.

- ◆ **Stati:** configurazioni di n regine ($0 \leq n \leq 8$), una per colonna a partire da sinistra, tali che nessuna regina ne attacchi un'altra.
- ◆ **Funzione successore:** aggiunge una regina in una qualsiasi casella della colonna vuota più a sinistra, in modo tale che essa non sia attaccata da nessun'altra regina.

Questa formulazione riduce lo spazio degli stati del problema da 3×10^{14} a soli 2.057, ed è facile trovare soluzioni. Nel caso di 100 regine, comunque, la formulazione iniziale ha circa 10^{400} stati mentre quella migliorata ne ha circa 10^{52} (Esercizio 3.5): il miglioramento è notevole, ma lo spazio degli stati ridotto è ancora troppo grande per essere gestito dagli algoritmi di questo capitolo. Nel Capitolo 4 descriviamo la formulazione a stato completo e nel Capitolo 5 presenteremo un semplice algoritmo che rende facile risolvere il problema anche con un milione di regine.

Problemi reali

Abbiamo già visto come un problema di ricerca dell'itinerario sia definito specificando una serie di locazioni e di transizioni lungo gli archi che le collegano. Gli algoritmi che trovano itinerari sono usati in una varietà di applicazioni: dall'istradamento nelle reti di computer, alla pianificazione delle operazioni militari, ai sistemi di supporto per la generazione di pacchetti di viaggi aerei. Questi problemi tipicamente sono difficili da specificare. Consideriamo una versione semplificata del problema dei viaggi in aereo.

problema di ricerca dell'itinerario

- ◆ **Stati:** ognuno è rappresentato da una locazione (ovvero un aeroporto) e dalla ora corrente.
- ◆ **Stato iniziale:** specificato dal problema.
- ◆ **Funzione successore:** restituisce gli stati che risultano dall'azione di prendere un volo (eventualmente con indicazioni specifiche riguardanti la classe e il posto a sedere) che parte in orario successivo a quello corrente (tenendo in considerazione il tempo di check-in) dallo stesso aeroporto in cui si trova l'agente.
- ◆ **Test obiettivo:** siamo arrivati alla destinazione desiderata entro un tempo massimo prestabilito?

- ♦ **Costo di cammino:** dipende dal costo monetario, dai tempi di attesa, dalla durata dei voli, dalle procedure di dogana, dalla qualità dei posti, dall'ora, dal tipo di aereo, dai premi dovuti ai viaggiatori frequenti e così via.

I sistemi commerciali di supporto alla generazione di pacchetti di viaggio aereo usano una formulazione del problema simile a questa, molto complicata dalla necessità di gestire le complesse politiche di prezzo imposte dalle linee aeree. Ogni viaggiatore sa, tuttavia, che non tutti i voli vanno come previsto. Un sistema davvero buono prevederebbe anche dei piani di riserva, come prenotazioni aggiuntive su voli alternativi, nella misura in cui queste sono giustificate dal loro costo e dalla probabilità di fallimento del piano originale.

I **problemi di viaggio** sono molto simili a quelli di ricerca di un itinerario, ma c'è un'importante differenza. Considerate ad esempio il problema "visita ogni città nella Figura 3.2 almeno una volta, partendo e arrivando a Bucarest". Come nella ricerca di itinerari, le azioni corrispondono a spostamenti tra città adiacenti. Lo spazio degli stati, tuttavia, è molto diverso. Ogni stato deve includere non solo la posizione corrente ma anche l'insieme delle città che l'agente ha già visitato. Così, lo stato iniziale sarebbe "In Bucarest; visitate {Bucarest}", uno stato intermedio tipico potrebbe essere "In Vaslui; visitate {Bucarest, Urziceni, Vaslui}", e il test obiettivo dovrebbe controllare che l'agente si trovi a Bucarest e che tutte le venti città della mappa siano state visitate.

Il **problema del commesso viaggiatore** (spesso indicato con l'acronimo TSP, dall'inglese *traveling salesperson problem*) è un problema di viaggio in cui bisogna visitare ogni città esattamente una volta: lo scopo è trovare il giro più breve. È noto che questo problema è NP-difficile, ma ciononostante sono stati compiuti grandi sforzi per migliorare gli algoritmi per la sua risoluzione. Oltre ad aiutare gli spostamenti dei commessi viaggiatori, questi algoritmi sono stati applicati per pianificare i movimenti dei macchinari che costruiscono le schede elettroniche o i bracci meccanici che spostano materiale nei magazzini.

Un problema di **configurazione VLSI** richiede che vengano posizionati al meglio i milioni di componenti e di connessioni che formano un chip in modo da minimizzare l'area del chip stesso, i ritardi dei circuiti e i problemi elettrici, massimizzando così la resa della produzione. Il **problema di configurazione** viene subito dopo la fase di progettazione logica, e solitamente è diviso in due parti: **configurazione delle celle e instradamento dei canali**. Nella configurazione delle celle, i componenti di basso livello sono raggruppati in celle, ognuna delle quali svolge una funzione ben definita. Ogni cella ha una forma e dimensione prefissata e richiede un certo numero di connessioni con le altre. L'obiettivo è dislocare le celle sul chip in modo che non si sovrappongano e che vi sia spazio per le piste di collegamento necessarie. L'instradamento dei canali serve a definire il cammino preciso di ogni pista negli spazi tra le celle. Questi problemi di ricerca sono davvero complessi, ma meritano decisamente di essere risolti: nel Capitolo 4 esamineremo alcuni algoritmi in grado di farlo.

problemi di viaggio

problema del
commesso viaggiatore

configurazione VLSI

La navigazione dei robot è una generalizzazione del problema di ricerca di itinerario che abbiamo descritto qui sopra. Invece di un insieme discreto di itinerari, un robot si può muovere in uno spazio continuo e avere, almeno in via di principio, un insieme infinito di possibili stati e azioni. Per un robot di forma circolare che si muove su una superficie piana, lo spazio è essenzialmente bidimensionale. Quando il robot è dotato di braccia o gambe o ruote che devono essere controllate, lo spazio di ricerca diventa multidimensionale, e sono richieste tecniche avanzate solo per renderlo finito. Esamineremo alcuni di questi metodi nel Capitolo 25, nel 2° volume. Oltre alla complessità intrinseca nel problema, i robot reali devono anche gestire gli errori nei loro sensori e nei controlli motorizzati.

navigazione dei robot

Sequenze di montaggio automatico di oggetti complessi da parte di un robot sono state eseguite per la prima volta da FREDDY (Michie, 1972). Da allora il progresso è stato lento ma continuo, fino al punto da rendere economicamente praticabile il montaggio automatico di oggetti complicati come un motore elettrico. In questi problemi lo scopo è trovare l'ordine in cui assemblare le varie parti dell'oggetto: se questo è sbagliato, a un certo punto non sarà possibile aggiungere un pezzo senza vanificare una parte del lavoro già svolto. Controllare che un determinato passo nella sequenza sia accettabile è un difficile problema di ricerca geometrica, molto simile alla navigazione dei robot. Di conseguenza, la generazione di successori legali è la parte più costosa della costruzione di una sequenza di montaggio: ogni algoritmo, se vuole essere applicabile nella pratica, deve evitare di esplorare lo spazio degli stati nella sua interezza e considerarne solo una frazione molto piccola. Un altro problema importante di montaggio è la progettazione di proteine, il cui obiettivo è trovare una sequenza di aminoacidi in grado di ripiegarsi in una proteina tridimensionale con le caratteristiche ideali per curare una carta malattia.

sequenze di montaggio automatico

In anni recenti c'è stata una crescente domanda di agenti software in grado di svolgere ricerche su Internet, cercando le risposte a domande poste dall'utente, informazioni su un argomento dato, o anche occasioni di shopping particolarmente ghiotte. Questa è una buona applicazione delle tecniche di ricerca, perché è facile rappresentare Internet come un grafo composto di nodi (pagine) collegate da link. Una descrizione completa della ricerca su Internet sarà presentata nel Capitolo 10.

progettazione di proteine

3.3 Cercare soluzioni |

Adesso che abbiamo formulato i problemi, dobbiamo risolverli: per far questo bisognerà svolgere una ricerca nello spazio degli stati. Questo capitolo descrive le tecniche che utilizzano un albero di ricerca esplicito, generato partendo dallo stato iniziale e dalla funzione successore che, insieme, definiscono appunto lo spazio de-

ricerche su Internet

albero di ricerca

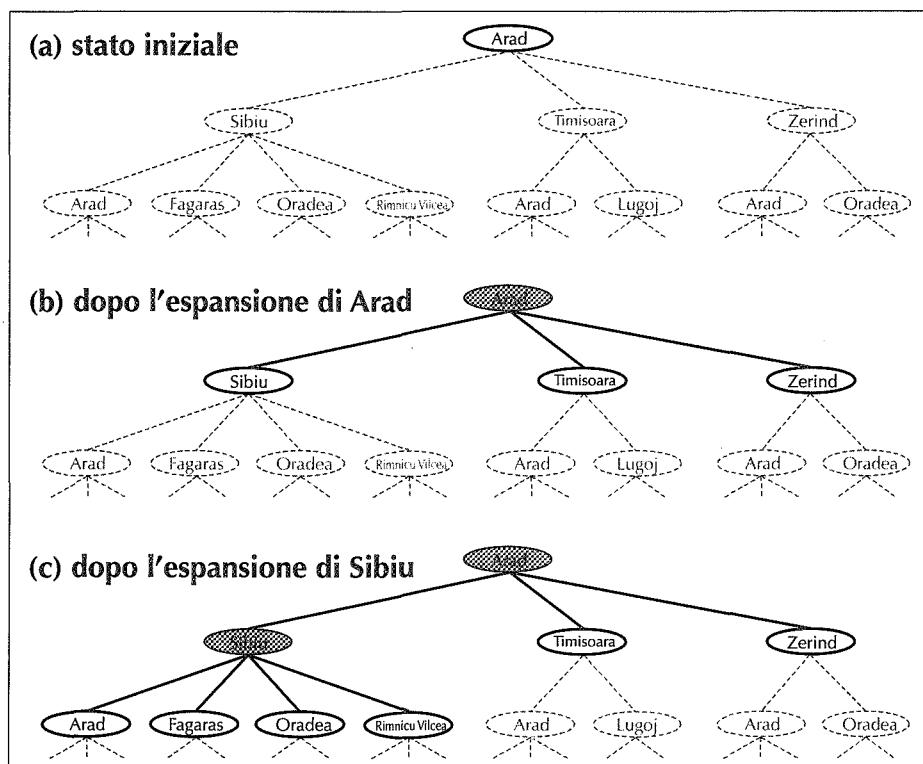


Figura 3.6 Alberi parziali per la ricerca dell'itinerario da Arad a Bucarest. I nodi già espansi sono ombreggiati in grigio; quelli generati ma non ancora espansi sono evidenziati in grassetto; quelli non ancora generati sono indicati con linee tratteggiate.

gli stati. In generale si può avere un grafo di ricerca anziché un albero, quando lo stesso stato può essere raggiunto seguendo cammini diversi. Considereremo quest'importante complicazione nel Paragrafo 3.5.

La Figura 3.6 mostra alcune espansioni dell'albero di ricerca dell'itinerario da Arad a Bucarest. La radice dell'albero è un nodo di ricerca corrispondente allo stato iniziale, In(Arad). Il primo passo è verificare che questo non sia uno stato obiettivo. In questo caso è chiaro che non è così, ma è importante eseguire ugualmente la verifica per evitare imprevisti nel caso di problemi particolari come “partendo da Arad, arrivare in Arad”. Dato che questo non è uno stato obiettivo, dobbiamo considerarne altri. Per far questo bisogna espandere lo stato corrente applicandovi la funzione successore e generare così un nuovo insieme di stati. In questo caso otteniamo tre nuovi stati: In(Sibiu), In(Timisoara) e In(Zerind). Ora dobbiamo scegliere quali di queste tre possibilità espandere ulteriormente.

nodo di ricerca

espansione di uno stato
generazione di stati

```

function RICERCA-ALBERO(problema, strategia) returns una soluzione, o il fallimento
    inizializza l'albero di ricerca usando lo stato iniziale di problema
    loop do
        if non ci sono più candidati per l'espansione then return fallimento
        scegli un nodo foglia per l'espansione in base alla strategia
        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        else espandi il nodo e aggiungi i nodi risultanti all'albero di ricerca

```

Figura 3.7 Una descrizione informale dell'algoritmo generale di ricerca su albero.

Questa è l'essenza della ricerca: approfondire un'opzione e mettere per il momento da parte le altre, pronti a riprenderle nel caso la prima scelta non porti a una soluzione. Supponiamo di scegliere per primo Sibiu. Controlliamo per vedere se questo è uno stato obiettivo (non è così) e quindi lo espandiamo per ottenere In(Arad), In(Fagaras), In(Oradea) e In(Rimnicu Vilcea). A questo punto dovremo scegliere uno di questi quattro, oppure tornare indietro e ripartire da Timisoara o Zerind. Continueremo a scegliere, controllare il test obiettivo ed espandere gli stati finché non avremo trovato una soluzione o non avremo considerato tutti gli stati. La scelta dello stato da espandere è dettata dalla strategia di ricerca adottata: l'algoritmo generale di ricerca su albero è descritto informalmente nella Figura 3.7.

È importante distinguere tra lo spazio degli stati e l'albero di ricerca. Nel caso del problema dell'itinerario, gli stati sono solamente 20, uno per ogni città; lo spazio degli stati invece contiene infiniti cammini, quindi anche l'albero avrà un numero infinito di nodi. Ad esempio, i tre cammini Arad–Sibiu, Arad–Sibiu–Arad, Arad–Sibiu–Arad–Sibiu sono i primi tre di una sequenza infinita (naturalmente un buon algoritmo di ricerca evita di seguire cammini ciclici; il Paragrafo 3.5 mostra come si fa).

Ci sono molti modi di rappresentare i nodi, ma noi useremo una struttura dati con cinque componenti.

- ◆ STATO: lo stato a cui il nodo corrisponde nello spazio degli stati.
- ◆ NODO-PADRE: il nodo nell'albero di ricerca che ha generato questo nodo.
- ◆ AZIONE: l'azione che è stata eseguita nel nodo padre per generare questo.
- ◆ COSTO-DEL-CAMMINO: tradizionalmente indicato con $g(n)$, è il costo del cammino dallo stato iniziale al nodo, come indicato dai puntatori dei nodi antenati.
- ◆ PROFONDITÀ: il numero di passi del cammino dallo stato iniziale.

strategia di ricerca

È importante ricordare la distinzione tra nodi e stati. Un nodo è una struttura dati che serve a costruire l'albero di ricerca; uno stato corrisponde a una particolare configurazione del mondo. Ne consegue che i nodi, a differenza degli stati, giacciono su cammini particolari definiti dai puntatori NODO-PADRE. Inoltre, due nodi differenti possono contenere lo stesso stato del mondo, generato seguendo due diversi cammini di ricerca. La struttura dati che rappresenta un nodo è illustrata nella Figura 3.8.

È anche necessario rappresentare la collezione di nodi che sono stati generati ma non ancora espansi: questa prende il nome di frontiera (fringe) della ricerca. Ogni elemento della frontiera è un nodo foglia dell'albero, ovvero un nodo che non ha alcun successore. Nella Figura 3.6, la frontiera di ogni albero consiste nei nodi evidenziati in grassetto. La rappresentazione più semplice della frontiera potrebbe consistere in un insieme di nodi: la strategia di ricerca allora consisterebbe in una funzione che sceglie il nodo successivo da espandere all'interno di tale insieme. Concettualmente tutto ciò è semplice, ma potrebbe risultare molto pesante dal punto di vista computazionale, perché la funzione che implementa la strategia dovrebbe considerare ogni volta tutti i nodi per scegliere il migliore. Noi adotteremo quindi una soluzione diversa, in cui la collezione di nodi è implementata come una coda. Le operazioni su una coda sono le seguenti:

- ◆ GENERA-CODA(*elemento*, ...) crea una coda con il dato *elemento*/i.
- ◆ VUOTA?(*coda*) restituisce true (vero) solo se non ci sono più elementi nella coda.
- ◆ PRIMO(*coda*) restituisce il primo elemento della coda.
- ◆ RIMUOVI-PRIMO(*coda*) restituisce PRIMO(*coda*) e lo rimuove dalla coda.
- ◆ INSERISCI(*elemento*, *coda*) inserisce un elemento nella coda e restituisce la coda risultante (come vedremo, tipi differenti di coda inseriscono gli elementi secondo ordini diversi).
- ◆ INSERISCI-TUTTI(*elementi*, *coda*) inserisce un insieme di elementi nella coda e restituisce la coda risultante.

Con queste definizioni possiamo scrivere la versione più formale dell'algoritmo generale di ricerca, mostrata nella Figura 3.9.

Misurare le prestazioni nella risoluzione di problemi

L'output di un algoritmo di risoluzione di problemi è o un fallimento o una soluzione (alcuni algoritmi potrebbero entrare in ciclo infinito e non restituire mai alcun output). Valuteremo le prestazioni di un algoritmo secondo quattro parametri.

- ◆ **Completezza:** l'algoritmo garantisce di trovare una soluzione, se questa esiste?
- ◆ **Ottimalità:** la strategia trova la soluzione ottima, com'è definita a pagina 86?

frontiera
nodo foglia

coda

completezza
ottimalità

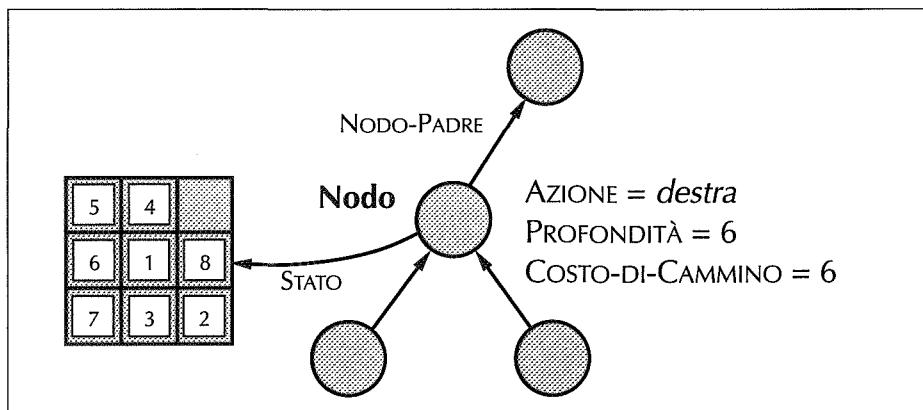


Figura 3.8 I nodi sono le strutture dati usate per costruire l'albero di ricerca. Ognuno di essi ha un nodo padre, uno stato e altri campi usati per memorizzare informazioni utili. Le frecce puntano dal nodo figlio al padre.

- ◆ **Complessità temporale:** quanto tempo ci mette a trovare una soluzione?
- ◆ **Complessità spaziale:** di quanta memoria necessita per effettuare la ricerca?

La complessità spaziale e quella temporale sono sempre considerate in rapporto a una misura della difficoltà del problema. Nell'informatica teorica la misura tipica è la dimensione del grafo degli stati, che viene considerato una struttura dati esplicita passata come input al programma di ricerca (la mappa della Romania ne è un esempio). Nell'IA, dove il grafo è rappresentato implicitamente dallo stato iniziale e dalla funzione successore ed è spesso infinito, la complessità si esprime usando tre quantità: b , il fattore di ramificazione (*branching factor*) o numero massimo di successori di un nodo; d , la profondità (*depth*) del nodo obiettivo più vicino allo stato iniziale; m , la lunghezza massima dei cammini nello spazio degli stati.

Il tempo si misura spesso con il numero di nodi generati⁵ durante la ricerca e lo spazio con il numero massimo di nodi mantenuti in memoria.

Per valutare l'efficienza di un algoritmo possiamo considerare solamente il costo di ricerca, che tipicamente dipende dalla complessità temporale ma può includere un termine che considera la memoria usata, oppure utilizzare il costo totale, che unisce il costo di ricerca con il costo di cammino della soluzione trova-

complessità temporale

complessità spaziale

fattore di ramificazione

costo di ricerca

costo totale

⁵ In alcuni testi il tempo viene misurato in termini di nodi espansi. Le due misure differiscono al più di un fattore b . A noi sembra che il tempo di esecuzione dell'espansione di un nodo padre aumenti col numero di nodi generati.

function RICERCA-ALBERO(*problema, frontiera*) **returns** una soluzione, o il fallimento

```

frontiera  $\leftarrow$  INSERISCI(CREA-NODO(STATO-INIZIALE[problema]), frontiera)
loop do
    if VUOTA?(frontiera) then return fallimento
    nodo  $\leftarrow$  RIMUOVI-PRIMO(frontiera)
    if TEST-OBIETTIVO[problema] applicato a STATO[nodo] ha successo
        then return SOLUZIONE(nodo)
    frontiera  $\leftarrow$  INSERISCI-TUTTI(ESPANDI(nodo, problema), frontiera)

```

function ESPANDI(*nodo, problema*) **returns** un insieme di nodi

```

successori  $\leftarrow$  l'insieme vuoto
for each <azione, risultato> in FUNZIONE-SUCCESSORE[problema](STATO[nodo]) do
    s  $\leftarrow$  un nuovo NODO
    STATO[s]  $\leftarrow$  risultato
    NODO-PADRE[s]  $\leftarrow$  nodo
    AZIONE[s]  $\leftarrow$  azione
    COSTO-DI-CAMMINO[s]  $\leftarrow$  COSTO-DI-CAMMINO[nodo] +
        COSTO-DI-PASSO(nodo, azione, s)
    PROFONDITÀ[s]  $\leftarrow$  PROFONDITÀ[nodo] + 1
    aggiungi s a successori
return successori

```

Figura 3.9 L'algoritmo generale di ricerca su albero. Notate che il parametro *frontiera* dev'essere una coda vuota, e il tipo di coda influenzera l'ordine della ricerca. La funzione SOLUZIONE restituisce la sequenza di azioni ottenuta seguendo i puntatori al nodo padre dallo stato finale fino alla radice dell'albero.

ta. Per il problema dell'itinerario da Arad a Bucarest, il costo di ricerca è la quantità di tempo impiegata nel calcolo mentre il costo della soluzione è la lunghezza totale dell'itinerario in chilometri. Per calcolare il costo totale, quindi, dovremmo sommare chilometri e millisecondi. Tra i due non esiste un "cambio ufficiale", ma in questo caso potrebbe essere ragionevole usare una stima della velocità media della macchina, dato che dopotutto all'agente interessa solamente il tempo. Questo permette all'agente di trovare un punto preciso in cui diventa controproducente spendere ulteriore tempo in calcoli per trovare un cammino più breve.

3.4 Strategie di ricerca non informata

Questo paragrafo presenta cinque strategie che ricadono nella categoria della **ricerca non informata** (chiamata anche *blind search* o **ricerca cieca**). Il termine si riferisce al fatto che non si conosce informazione aggiuntiva sugli stati oltre a quella fornita nella definizione del problema: tutto quello che si può fare è generare successori e distinguere gli stati obiettivo dagli altri. Le strategie che sanno distinguere se uno stato non obiettivo è “più promettente” di un altro fanno parte della **ricerca informata** o **ricerca euristica**; le vedremo nel Capitolo 4. Tutte le strategie di ricerca sono contraddistinte dall’*ordine* in cui vengono espansi i nodi.

ricerca non informata

ricerca informata
ricerca euristica

Ricerca in ampiezza

La **ricerca in ampiezza** è una semplice strategia nella quale si espande prima il nodo radice, quindi tutti i suoi successori, poi i *loro* successori, e così via. In generale, tutti i nodi a una determinata profondità dell’albero devono essere stati espansi prima che si possa espandere uno dei nodi al livello successivo.

La ricerca in ampiezza può essere implementata invocando RICERCA-ALBERO con una frontiera vuota che consiste in una coda “first-in-first-out” (FIFO), assicurandosi così che i nodi visitati per primi saranno anche espansi per primi. In altre parole, invocare RICERCA-ALBERO (*problema*, CODA-FIFO()) risulta in una ricerca in ampiezza. La coda FIFO mette tutti i successori appena generati in fondo alla coda, il che significa che i nodi più vicini alla radice saranno espansi prima di quelli più profondi. La Figura 3.10 illustra il progredire della ricerca su un semplice albero binario.

Valuteremo la ricerca in ampiezza secondo i quattro criteri presentati nel precedente paragrafo. Possiamo vedere facilmente che è *completa*: se il nodo obiettivo più vicino alla radice si trova a una profondità finita d , la ricerca in ampiezza lo troverà dopo aver espanso tutti i nodi che lo precedono (a patto che il fattore di ramificazione b sia finito). Il nodo obiettivo *meno profondo* non è necessariamente quello *ottimo*; tecnicamente la ricerca in ampiezza è ottima solamente se il costo di cammino è una funzione monotona crescente della profondità del nodo: ad esempio quando le azioni hanno tutte lo stesso costo.

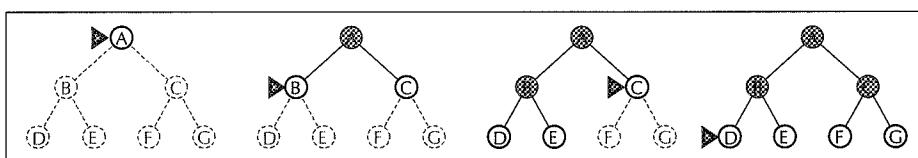


Figura 3.10 Ricerca in ampiezza su un semplice albero binario. A ogni iterazione viene espanso il nodo indicato.

Fin qui, la ricerca in ampiezza sembra una strategia piuttosto valida. Per vedere perché non è sempre la più indicata dobbiamo considerare quanto tempo e spazio occorrono per eseguire una ricerca. Per far questo, pensiamo a un ipotetico spazio degli stati in cui ogni stato ha b successori. La radice dell'albero di ricerca genera b nodi al primo livello, ognuno dei quali genera altri b nodi per un totale di b^2 al secondo livello. Ognuno di questi genera poi b ulteriori nodi, in modo che al terzo livello risultano b^3 nodi, e così via. Ora supponiamo che la soluzione si trovi a una profondità d . Nel caso peggiore dovremmo espandere tutti i nodi tranne l'ultimo al livello d (dato che l'obiettivo non viene espanso), generando $b^{d+1} - b$ nodi al livello $d + 1$. In definitiva il numero totale di nodi generati sarà:

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$

Ogni nodo generato deve rimanere in memoria, perché o fa parte della frontiera oppure è un antenato di un nodo nella frontiera. La complessità spaziale, quindi, è uguale a quella temporale (più un nodo per la radice).

Coloro che si occupano di analisi di complessità si preoccupano (o esaltano, se amano le sfide) quando incontrano una complessità esponenziale come $O(b^{d+1})$. La Figura 3.11 mostra perché, elencando il tempo e la memoria richiesti per una ricerca in ampiezza con fattore di ramificazione $b = 10$, per diversi valori della profondità della soluzione d . La tabella presume che si possano generare 10.000 nodi al secondo e che ognuno richieda circa 1000 byte di memoria. Molti problemi di ricerca, eseguiti su un personal computer moderno, concordano con queste ipotesi (più o meno un fattore 100).

La Figura 3.11 illustra due concetti importanti. Prima di tutto, *per le ricerche in ampiezza i requisiti di memoria sono un problema più grande del tempo di esecuzione.* 31 ore non sarebbe un tempo troppo lungo da attendere per avere la soluzione a un problema importante di profondità 8, ma ben pochi computer hanno il terabyte di memoria necessario. Fortunatamente esistono altre strategie di ricerca che hanno minori requisiti spaziali.

In secondo luogo, i requisiti temporali sono sempre un fattore importante. Se la soluzione del vostro problema si trova a profondità 12, date le nostre ipotesi la ricerca in ampiezza (o inverò qualsiasi ricerca non informata) prenderà 35 anni. In generale, *i problemi di ricerca di complessità esponenziale non possono essere risolti con metodi non informati tranne che nelle istanze più piccole.*

Ricerca a costo uniforme

La ricerca in ampiezza risulta ottima quanto tutti i costi di tutti i passi sono identici, perché espande sempre il nodo più vicino alla radice. Con una semplice estensione possiamo trovare un algoritmo che risulta ottimo per qualsiasi costo di passo. Invece di espandere il nodo meno profondo, la **ricerca a costo uniforme** espande il nodo n con il *minimo costo di cammino*. Notate che, se i costi di passo sono tutti uguali, questa strategia è identica a quella in ampiezza.

profondità	nodi	tempo	memoria
2	1100	0,11 secondi	1 megabyte
4	111.100	11 secondi	106 megabyte
6	10^7	19 minuti	10 gigabyte
8	10^9	31 ore	1 terabyte
10	10^{11}	129 giorni	101 terabyte
12	10^{13}	35 anni	10 petabyte
14	10^{15}	3523 anni	1 exabyte

Figura 3.11
Requisiti di tempo e memoria per la ricerca in ampiezza.
I numeri indicati presuppongono un fattore di ramificazione $b = 10$; 10.000 nodi/secondo; 1000 byte/nodo.

La ricerca a costo uniforme non si preoccupa del *numero* di passi che compongono un cammino, ma solo del suo costo totale. Ne consegue che, se dovesse capitare di espandere un nodo che ha un'azione a costo zero che porta allo stesso stato (ad esempio l'azione nulla *NoOp*), l'algoritmo rimarrebbe bloccato in un ciclo infinito. Possiamo garantire la completezza solo nel caso in cui il costo di ogni passo sia maggiore o uguale a una costante positiva piccola ε . Questa condizione è anche sufficiente a garantire l'*ottimalità*: infatti il costo di un cammino aumenta sempre man mano che lo percorriamo, ed è facile verificare che l'algoritmo espande i nodi in ordine di costo di cammino crescente. Di conseguenza, il primo nodo obiettivo prescelto per l'espansione dovrà corrispondere alla soluzione ottima (ricordate che RICERCA-ALBERO esegue il test obiettivo solo sui nodi che stanno per essere espansi). Vi esortiamo a provare quest'algoritmo per trovare l'itinerario più breve per arrivare a Bucarest.

La ricerca a costo uniforme è guidata dal costo dei cammini e non dalla loro profondità, per cui la sua complessità non può essere espressa facilmente in termini di fattore di ramificazione b e profondità d . Consideriamo invece C^* , definito come il costo della soluzione ottima, e diamo per ipotesi che ogni azione costi almeno ε . La complessità spaziale e temporale dell'algoritmo nel caso pessimo sarà $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, che può essere molto maggiore di b^d . Questo è dovuto al fatto che spesso la ricerca a costo uniforme esplora grandi alberi fatti di piccoli passi prima di considerare i cammini che prevedono passi molto grandi e forse più utili. Quando i costi di passo sono tutti uguali, naturalmente, $b^{1+\lfloor C^*/\varepsilon \rfloor}$ si riduce a b^d .

Ricerca in profondità

La ricerca in profondità espande sempre per primo il nodo più *profondo*, ovvero quello più lontano dalla radice, nella frontiera corrente dell'albero di ricerca. Il progredire della ricerca è illustrato nella Figura 3.12. La ricerca raggiunge immediatamente il livello più profondo dell'albero, dove i nodi non hanno successori. L'espansione di tali nodi li rimuove dalla frontiera, per cui la ricerca “torna indietro” e considera il nodo più profondo che ha ancora successori non espansi.

ricerca in profondità

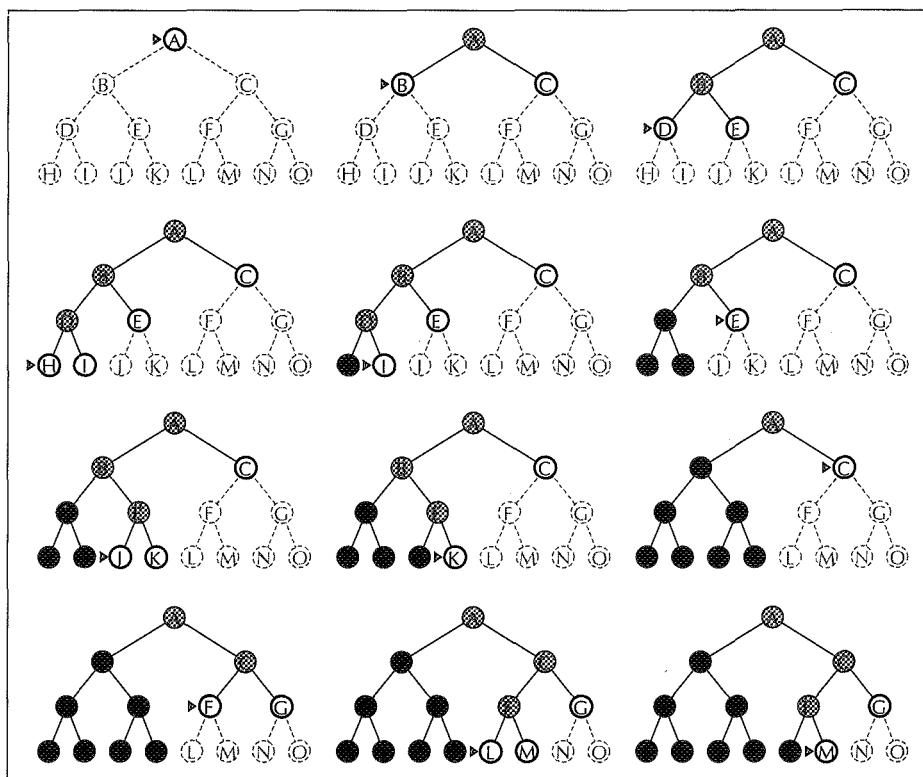


Figura 3.12 Ricerca in profondità su un albero binario. I nodi che sono stati espansi e non hanno discendenti nella frontiera possono essere rimossi dalla memoria; sono indicati in nero. Si presume che i nodi a profondità 3 non abbiano successori e che l'unico nodo obiettivo sia M .

Questa strategia può essere implementata da RICERCA-ALBERO usando una coda “last-in-first-out” (LIFO), più nota come *stack* o pila. Un’alternativa molto diffusa a tale implementazione consiste nell’utilizzo di una funzione ricorsiva che invoca se stessa su ognuno dei propri figli (un algoritmo simile, che incorpora un limite alla massima profondità raggiungibile, è mostrato nella Figura 3.13).

La ricerca in profondità ha requisiti di memoria molto modesti: in un dato momento deve memorizzare un solo cammino dalla radice a un nodo foglia, insieme a tutti i “fratelli” non ancora espansi di ogni nodo del cammino. Una volta che è stato espanso, e che tutti i suoi discendenti sono stati esplorati completamente, un nodo può essere rimosso dalla memoria (v. Figura 3.12). Per uno spazio degli stati con fattore di ramificazione b e profondità massima m , la ricerca in profondità richiede la memorizzazione di soli $bm + 1$ nodi. Con le stesse ipotesi della Figura 3.11, e presumendo che i nodi alla stessa profondità dell’obiettivo non abbiano successori, possiamo calcolare che una ricerca in profondità con $d = 12$ richiede solo 118 kilobyte invece di 10 petabyte, un fattore spaziale 10 miliardi di volte inferiore.

```

function RICERCA-PROFONDITÀ-LIMITATA(problema, limite) returns una soluzione, o il
fallimento/taglio
return RPL-RICORSIVA(CREA-NODO(STATO-INIZIALE[problema]), problema, limite)

function RPL-RICORSIVA(nodo, problema, limite) returns una soluzione, o il fallimento/taglio
avvenuto_taglio?  $\leftarrow$  false
if TEST-OBIETTIVO[problema](STATO[nodo]) then return SOLUZIONE(nodo)
else if PROFONDITÀ[nodo] = limite then return taglio
else for each successore in ESPANDI(nodo, problema) do
    risultato  $\leftarrow$  RPL-RICORSIVA(successore, problema, limite)
    if result = taglio then avvenuto_taglio?  $\leftarrow$  true
    else if risultato  $\neq$  fallimento then return risultato
if avvenuto_taglio? then return taglio else return fallimento

```

Figura 3.13 Un'implementazione ricorsiva della ricerca a profondità limitata.

Una variante della ricerca in profondità richiede ancora meno memoria: la **ricerca con backtracking** (il termine significa letteralmente “ritornare sui propri passi”). Con il backtracking quando si espande un nodo non vengono generati tutti i successori, bensì uno solo; ogni nodo parzialmente espanso ricorda quale successore deve generare in seguito. In questo modo è necessaria solo $O(m)$ memoria anziché $O(bm)$. La ricerca con backtracking permette di applicare un altro trucco che fa risparmiare memoria e tempo, che consiste nel generare un successore *modificando* la descrizione corrente dello stato anziché, per prima cosa, copiarla. Questo riduce i requisiti spaziali a una sola descrizione dello stato e $O(m)$ azioni. Affinché il trucco funzioni, dev'essere possibile annullare ogni modifica quando “torniamo sui nostri passi” per generare un altro successore. Nei problemi caratterizzati da descrizioni di stato molto grandi, come il montaggio robotizzato, queste tecniche risultano fondamentali per il successo.

ricerca con
backtracking

Lo svantaggio di questo tipo di ricerca è che può sbagliare e imboccare un cammino molto lungo (o addirittura infinito) quando invece una scelta differente potrebbe portare a una soluzione vicina alla radice dell'albero. Ad esempio, nella Figura 3.12, una ricerca in profondità esplorerebbe l'intero sottoalbero sinistro anche se *C* fosse un nodo obiettivo. Se anche il nodo *J* fosse un obiettivo, una ricerca in profondità lo restituirebbe come soluzione; ne consegue che non è una strategia ottima. Se il sottoalbero sinistro avesse profondità illimitata ma non contenesse nessuna soluzione, una ricerca in profondità non terminerebbe mai l'esecuzione; ne consegue che non è completa. Nel caso pessimo genererà tutti gli $O(b^m)$ nodi nell'albero, dove *m* è la profondità massima di un nodo. Notate che *m* può essere molto più grande di *d* (la profondità della soluzione più vicina alla radice), ed è infinito per alberi illimitati.

ricerca a profondità limitata

diametro

ricerca ad approfondimento iterativo

Ricerca a profondità limitata

Il problema degli alberi illimitati può essere alleviato impostando un limite predefinito ℓ alla profondità della ricerca. Questo significa che i nodi alla profondità ℓ saranno trattati come se non avessero alcun successore. Quest'approccio prende il nome di **ricerca a profondità limitata**. Il limite alla profondità risolve il problema dei cammini di lunghezza infinita ma, sfortunatamente, introduce una nuova fonte di incompletezza se $\ell < d$, ovvero se il nodo obiettivo più vicino alla radice si trova comunque a una profondità maggiore del limite (cosa non improbabile quando d è sconosciuta). La ricerca a profondità limitata risulterà anche non ottima, qualora si dovesse scegliere $\ell > d$. La sua complessità temporale è $O(b^\ell)$ e quella spaziale è $O(b\ell)$. La ricerca in profondità può essere vista come un caso speciale di quella a profondità limitata con $\ell = \infty$.

Talvolta i limiti di profondità possono essere basati sulla conoscenza del problema. Ad esempio, la nostra mappa della Romania riporta 20 città: quindi sappiamo che, se esiste una soluzione, dev'essere lunga al massimo 19 passi, e una possibile scelta è $\ell = 19$. In effetti, studiando la mappa con attenzione scopriremmo che ogni città può essere raggiunta da ogni altra al massimo in 9 passi. Questo numero, noto come **diametro** dello spazio degli stati, rappresenta un limite di profondità preferibile al precedente e porta a una ricerca più efficiente. Nella maggior parte dei casi, comunque, non si può stabilire quale sia un buon limite di profondità finché non si è risolto il problema.

La ricerca a profondità limitata può essere implementata come una semplice modifica all'algoritmo generale di ricerca o a quello ricorsivo di ricerca in profondità. Nella Figura 3.13 ne abbiamo riportato lo pseudocodice. Notate che questo tipo di ricerca può terminare con due tipi di fallimento: il valore standard (*fallimento*) indica che non esiste soluzione; il cosiddetto valore di taglio (*taglio*) indica che non è stata trovata alcuna soluzione all'interno del limite di profondità specificato.

Ricerca ad approfondimento iterativo

La **ricerca ad approfondimento iterativo** è una strategia generale usata spesso in combinazione con la ricerca in profondità per trovare il limite ideale per quest'ultima. Per far questo il limite viene incrementato progressivamente: prima 0, poi 1, quindi 2 e così via, finché non viene trovato un nodo obiettivo. Questo avverrà non appena il limite raggiunge d , la profondità del nodo obiettivo più vicino alla radice. L'algoritmo è illustrato nella Figura 3.14. L'approfondimento iterativo prende gli aspetti migliori delle ricerche in profondità e in ampiezza: come la prima, ha dei requisiti di memoria molto modesti, e precisamente $O(bd)$; come la seconda, è completa quando il fattore di ramificazione è finito e ottima quando il costo di cammino è una funzione non decrescente della profondità del nodo. La Figura 3.15 mostra quattro iterazioni di RICERCA-APPROFONDIMENTO-ITERATIVO su un albero binario: la soluzione viene trovata alla quarta iterazione.

```

function RICERCA-APPROFONDIMENTO-ITERATIVO(problema) returns una soluzione, o il fallimento
  inputs: problema, un problema

  for profondità  $\leftarrow 0$  to  $\infty$  do
    risultato  $\leftarrow$  RICERCA-PROFOUNDITÀ-LIMITATA(problema, profondità)
    if risultato  $\neq$  taglio then return risultato

```

Figura 3.14 L'algoritmo di ricerca ad approfondimento iterativo, che esegue ripetutamente una ricerca a profondità limitata aumentando via via il limite. L'algoritmo termina quando trova una soluzione o quando la ricerca restituisce *fallimento*, indicando così che non ne esiste alcuna.

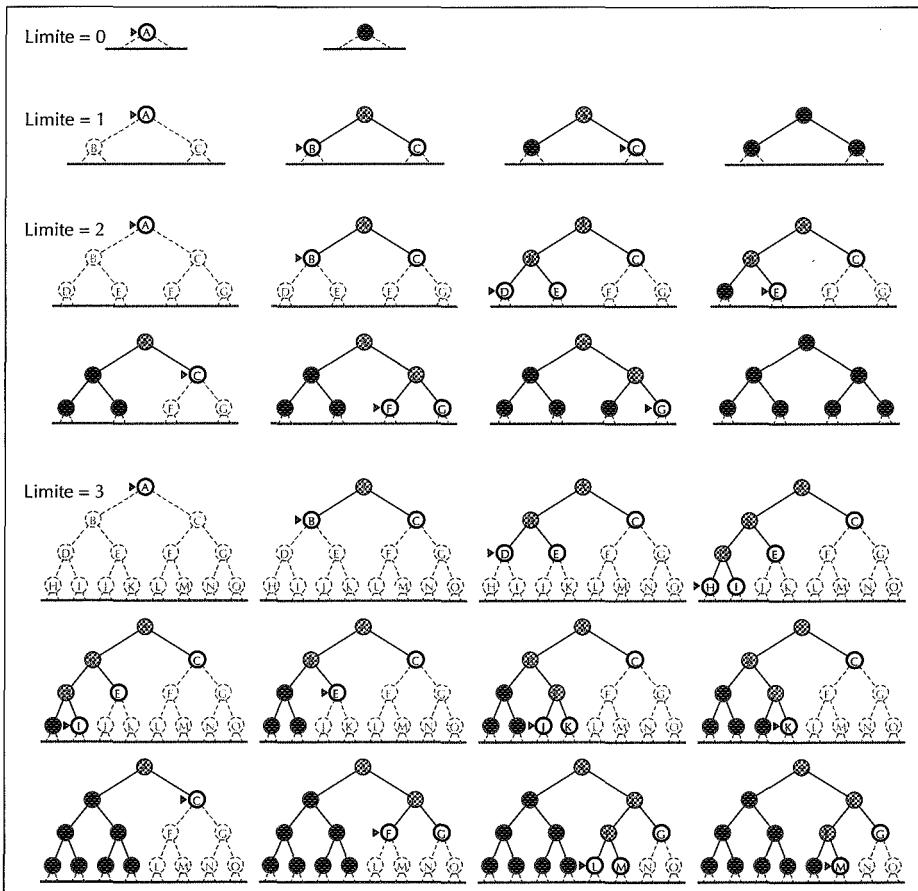


Figura 3.15
Quattro iterazioni di ricerca ad approfondimento iterativo su un albero binario.

La ricerca ad approfondimento iterativo potrebbe sembrare uno spreco, dato che gli stessi stati vengono generati più volte. In effetti, il suo costo non risulta così alto: il fatto è che in un albero di ricerca con lo stesso fattore di ramificazione a ogni livello (o uno molto simile), la maggior parte dei nodi si trova comunque al livello più basso, e quindi la generazione multipla dei livelli superiori non ha un grande impatto. Nella ricerca ad approfondimento iterativo i nodi al livello più basso (a profondità d) sono generati una sola volta, quelli al livello immediatamente superiore due volte, e così via, fino ai figli diretti del nodo radice, che sono generati d volte. Così, il numero totale dei nodi generati è

$$N(\text{RAI}) = (d)b + (d-1)b^2 + \dots + (1)b^d,$$

il che ci dà una complessità temporale $O(b^d)$. Possiamo confrontare questo risultato con il numero di nodi generati in una ricerca in ampiezza:

$$N(\text{RIA}) = b + b^2 + \dots + b^d + (b^{d+1} - b).$$

Notate che la ricerca in ampiezza genera dei nodi alla profondità $d+1$, cosa che la ricerca ad approfondimento iterativo non fa. Il risultato è che quest'ultima strategia è in realtà *più veloce* di quella in ampiezza, nonostante la generazione multipla degli stati. Ad esempio, per $b = 10$ e $d = 5$, i risultati sono

$$N(\text{RAI}) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$

$$N(\text{RIA}) = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100.$$

In generale, la ricerca ad approfondimento iterativo è il metodo preferito di ricerca non informata quando lo spazio di ricerca è grande e la profondità della soluzione non è nota.

Questa ricerca è analoga a quella in ampiezza, in quanto a ogni iterazione esplora completamente un livello di nodi prima di prendere in considerazione il successivo. Sembra una buona idea sviluppare un'analogia versione iterativa della ricerca a costo uniforme, ereditando così la sua ottimalità e ovviando ai suoi pesanti requisiti di memoria. L'idea è stabilire un limite man mano crescente non più per la profondità, ma per il costo dei cammini. L'algoritmo risultante, chiamato **ricerca a lunghezza iterativa**, è analizzato nell'Esercizio 3.11. Sfortunatamente, dall'analisi risulta che la ricerca a lunghezza iterativa non è affatto vantaggiosa rispetto a quella a costo uniforme.

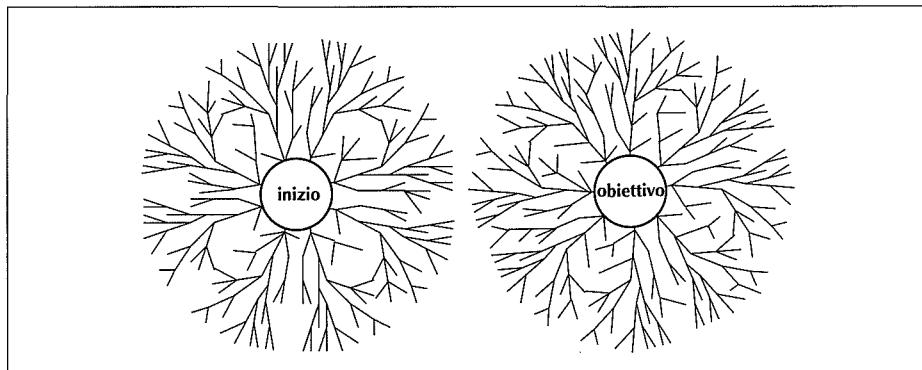


Figura 3.16
Una rappresentazione schematica di una ricerca bidirezionale che si sta per concludere con successo, non appena un ramo partito dal nodo iniziale ne incontrerà uno partito dal nodo obiettivo.

Ricerca bidirezionale

L'idea alla base della ricerca bidirezionale è eseguire due ricerche in parallelo, una in avanti dallo stato iniziale e l'altra all'indietro partendo dall'obiettivo, fermandosi quando si incontrano a metà strada (Figura 3.16). La ragione sta nel fatto che $b^{d/2} + b^{d/2}$ è molto più piccolo di b^d : questo equivale a dire che, nella figura, l'area totale dei due cerchi è inferiore all'area di un ipotetico cerchio centrato sul nodo iniziale e abbastanza grande da includere l'obiettivo.

Per implementare la ricerca bidirezionale occorre che una o entrambe le ricerche controllino, prima di espandere un nodo, che esso non faccia parte della frontiera dell'altro albero: in tal caso è stata trovata una soluzione. Ad esempio, se un problema ha una soluzione alla profondità $d = 6$, e se in ogni direzione si esegue una ricerca in ampiezza un nodo per volta, nel caso pessimo le due ricerche si incontreranno quando ognuna ha espanso tutti i nodi a profondità 3 tranne uno. Con $b = 10$, questo significa un totale di 22.200 generazioni di nodi, contro gli 11.111.100 di una ricerca standard in ampiezza. Controllare se un nodo fa parte della frontiera dell'altro albero di ricerca può essere fatto in tempo costante con una tabella di hash, per cui la complessità temporale di una ricerca bidirezionale è $O(b^{d/2})$. Almeno uno dei due alberi dev'essere tenuto in memoria per fare il controllo, per cui la complessità spaziale è parimenti $O(b^{d/2})$. Questo requisito spaziale rappresenta la debolezza maggiore della ricerca bidirezionale. L'algoritmo è completo e ottimo (quando i costi dei passi sono uniformi) se entrambe le ricerche sono in ampiezza; altre combinazioni possono sacrificare la completezza, l'ottimalità o entrambe.

La riduzione di complessità temporale rende interessante la ricerca bidirezionale, ma come si fa a cercare all'indietro? Non è facile come sembra. Definiamo predecessori di un nodo n , e indichiamo con $Pred(n)$, tutti quei nodi che hanno n come successore. La ricerca bidirezionale richiede che $Pred(n)$ sia calcolabile in

predecessori

modo efficiente. Il caso più semplice si ha quando tutte le azioni dello spazio degli stati sono reversibili, dimodoché $\text{Pred}(n) = \text{Succ}(n)$. In altri casi calcolare i predecessori può richiedere un certo ingegno.

Considerate anche la questione di cosa si intende per “obiettivo” nella “ricerca a ritroso dall’obiettivo”. Nel rompicapo a 8 tasselli e nella ricerca dell’itinerario in Romania, lo stato obiettivo è uno solo, per cui la ricerca all’indietro è molto simile a quella in avanti. Se ci sono più stati obiettivo *elencati esplicitamente* – come i due stati privi di sporco nella Figura 3.3 – allora possiamo costruire un nuovo obiettivo di comodo, i cui predecessori immediati sono tutti gli stati obiettivo reali. Alternativamente si può evitare qualche generazione ridondante di nodi considerando un insieme di stati obiettivo come fossero un singolo stato, e ognuno dei loro predecessori ancora come un insieme di stati, e precisamente quelli che hanno un corrispondente successore nell’insieme degli stati obiettivo (v. anche Paragrafo 3.6).

Il caso più difficile, per la ricerca bidirezionale, è quando viene fornita solo la descrizione implicita di un insieme di stati obiettivo potenzialmente molto grande come, negli scacchi, la condizione di “scacco matto”. Una ricerca all’indietro dovrebbe costruire una descrizione compatta di “tutti gli stati che portano al matto eseguendo la mossa m_1 ” e così via; tali descrizioni dovranno poi essere confrontate con gli stati generati dalla ricerca in avanti. Non esiste un modo generale per fare ciò in modo efficiente.

Confronto tra le strategie di ricerca non informata

La Figura 3.17 confronta le strategie di ricerca secondo i quattro criteri di valutazione definiti nel Paragrafo 3.4.

3.5 Evitare ripetizioni negli stati

Finora abbiamo ignorato totalmente una delle complicazioni più gravi del processo di ricerca: la possibilità di perdere tempo espandendo stati che sono già stati incontrati ed espansi. Per alcuni problemi, questa possibilità non si verifica mai; lo stesso spazio degli stati è un albero e c’è un solo cammino per ogni stato. La formulazione del problema delle 8 regine in cui ogni nuova regina è piazzata nella colonna vuota più a sinistra deve gran parte della sua efficienza proprio a questo: ogni stato può essere aggiunto seguendo un solo cammino. Se formuliamo il problema in modo che una regina possa essere piazzata su una colonna qualsiasi, ogni stato con n regine può essere raggiunto seguendo $n!$ cammini diversi.

Ci sono problemi in cui la ripetizione degli stati è inevitabile: ad esempio, tutti i problemi in cui le azioni sono reversibili, come le ricerche di itinerari o rompicapi a tasselli mobili. In questi casi gli alberi di ricerca sono infiniti, ma se

criterio	in ampiezza	a costo uniforme	in profondità	a profondità limitata	ad approfondimento iterativo	bidirezionale (se applicabile)
completa?	Si ^a	Si ^{a,b}	No	No	Si ^a	Si ^{a,d}
tempo	$O(b^{d+1})$	$O(b^{1+\lfloor C/\epsilon \rfloor})$	$O(b^m)$	$O(b^d)$	$O(b^d)$	$O(b^{d/2})$
spazio	$O(b^{d+1})$	$O(b^{1+\lfloor C/\epsilon \rfloor})$	$O(bm)$	$O(b\lambda)$	$O(bd)$	$O(b^{d/2})$
ottima?	Si ^c	Si	No	No	Si ^c	Si ^{c,d}

Figura 3.17 Valutazione delle strategie di ricerca. b è il fattore di ramificazione; d la profondità della soluzione più vicina alla radice; m la profondità massima dell'albero di ricerca; λ il limite alla profondità. Gli apicì sono da interpretare come segue: ^a completa se b è finito; ^b completa se i costi dei passi sono \geq di un ϵ positivo; ^c ottima se i costi dei passi sono tutti identici; ^d se in entrambe le direzioni si usa una ricerca in ampiezza.

“potiamo” alcuni stati ripetuti possiamo tagliare l’albero fino a renderlo finito, generando solo la porzione che copre il grafo dello spazio degli stati. Se consideriamo l’albero di ricerca solo fino a una certa profondità, è facile trovare casi in cui l’eliminazione degli stati ripetuti porta a una riduzione esponenziale del costo di ricerca. Nel caso estremo, uno spazio degli stati di dimensione $d+1$ (Figura 3.18(a)) diventa un albero con 2^d foglie (Figura 3.18(b)). Un esempio più realistico è la griglia rettangolare illustrata nella Figura 3.18(c). Su una griglia, ogni stato ha quattro successori, per cui l’albero di ricerca ha 4^d foglie se includiamo gli stati ripetuti; ma ci sono solo circa $2d^2$ stati distinti entro d passi da ogni stato. Per $d=20$, questo significa un trilione circa di nodi ma solamente 800 stati distinti.

griglia rettangolare

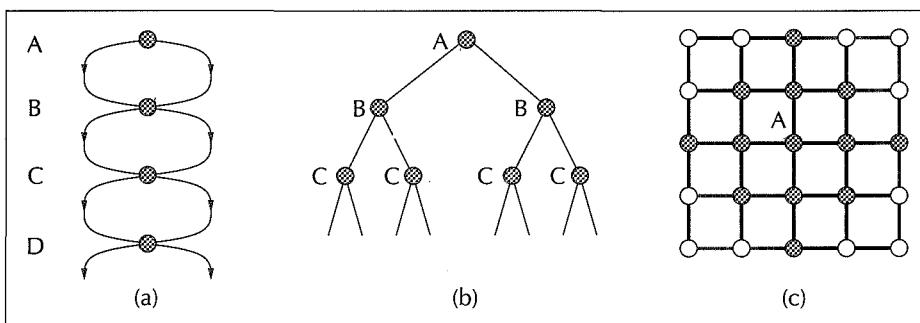


Figura 3.18 Spazi degli stati che generano un albero di ricerca di dimensioni esponenzialmente più grandi. (a) Uno spazio degli stati in cui ci sono due possibili azioni che portano da A a B, due da B a C e così via. Lo spazio degli stati contiene $d+1$ stati, ove d è la profondità massima. (b) l’albero di ricerca corrispondente, che ha 2^d rami corrispondenti ai 2^d cammini che attraversano lo spazio. (c) Uno spazio a griglia rettangolare: gli stati che si trovano entro due passi da quello iniziale (A) sono evidenziati in grigio.

Gli stati ripetuti non rilevati dall'algoritmo, insomma, possono far diventare irrisolvibile un problema che non lo è. Rilevare le ripetizioni solitamente significa confrontare il nodo che si sta espandendo con quelli che sono già stati espansi; se si trova una corrispondenza, l'algoritmo ha scoperto due diversi cammini che portano allo stesso stato e può scartarne uno.

Per la ricerca in profondità, i soli nodi in memoria sono quelli sul cammino che va dalla radice dell'albero al nodo corrente. Confrontare questi nodi con quello corrente permette all'algoritmo di rilevare cammini ciclici che possono essere scartati immediatamente. Questo va bene per assicurarsi che spazi degli stati che sono finiti non diventino infiniti per colpa dei cicli; sfortunatamente però non evita l'esplosione esponenziale dei cammini non ciclici nei problemi come quello nella Figura 3.18. L'unico modo di evitarlo è tenere più nodi in memoria: sostanzialmente si deve operare un compromesso tra spazio e tempo. *Gli algoritmi che dimenticano la loro storia sono condannati a ripeterla.*

Se un algoritmo ricorda ogni stato visitato, si può pensare che esplori direttamente il grafo dello spazio degli stati. Possiamo modificare l'algoritmo generale RICERCA-ALBERO aggiungendo una struttura dati chiamata **lista chiusa**, che memorizza ogni nodo espanso (talvolta la frontiera di nodi non espansi viene chiamata **lista aperta**). Se il nodo corrente corrisponde a un nodo nella lista chiusa, invece di essere espanso viene scartato. Il nuovo algoritmo prende il nome di RICERCA-GRAFO (Figura 3.19): nei problemi con molti stati ripetuti, è molto più efficiente di RICERCA-ALBERO. I suoi requisiti di spazio e tempo, nel caso pessimo, sono proporzionali alla dimensione dello spazio degli stati, che può essere molto più piccola di $O(b^d)$.

L'ottimalità è un aspetto spinoso della ricerca su grafo. Abbiamo già detto che quando si rileva uno stato ripetuto significa che l'algoritmo ha scoperto un altro cammino che porta allo stesso nodo. L'algoritmo RICERCA-GRAFO nella Figura 3.19 scarta sempre il cammino *appena scoperto*; naturalmente, se quest'ultimo è più breve di quello originale, RICERCA-GRAFO potrebbe perdere così una soluzione ottima. Fortunatamente, possiamo dimostrare (Esercizio 3.12) che ciò non può accadere quando si usa la ricerca a costo uniforme o quella in ampiezza con costo di passo costante; quindi queste due strategie ottime di ricerca su albero sono ottime anche su grafo. La ricerca ad approfondimento iterativo, d'altro canto, si espande prima in profondità e può facilmente seguire un cammino subottimo verso un nodo prima di trovare quello ottimo. Ne consegue che questa strategia deve controllare se un nuovo cammino appena scoperto verso un nodo è migliore di quello precedente, e in tal caso aggiornare le profondità e i costi di cammino dei discendenti di tale nodo.

Notate che l'uso della lista chiusa significa che la ricerca in profondità e quella ad approfondimento iterativo non hanno più requisiti spaziali lineari. Dato che l'algoritmo RICERCA-GRAFO tiene in memoria ogni nodo, alcune ricerche risulteranno impossibili a causa dei requisiti spaziali.



lista chiusa



lista aperta

function RICERCA-GRAFO(*problema*, *frontiera*) **returns** una soluzione, o il fallimento

```

chiuso  $\leftarrow$  un insieme vuoto
frontiera  $\leftarrow$  INSERISCI(CREA-NODO(STATO-INIZIALE[problema]), frontiera)
loop do
    if VUOTO?(frontiera) then return fallimento
    nodo  $\leftarrow$  RIMUOVI-PRIMO(frontiera)
    if TEST-OBIETTIVO[problema](STATO[nodo]) then return SOLUZIONE(nodo)
    if STATO[nodo] non è in chiuso then
        aggiungi STATO[nodo] a chiuso
        frontiera  $\leftarrow$  INSERISCI-TUTTI(ESPANDI(nodo, problema), frontiera)

```

Figura 3.19 L'algoritmo generale di ricerca su grafo. L'insieme *chiuso* può essere implementato con una tabella di hash per consentire una verifica efficiente della presenza di stati ripetuti. Quest'algoritmo presume che il primo cammino che raggiunge uno stato *s* sia sempre il più conveniente.

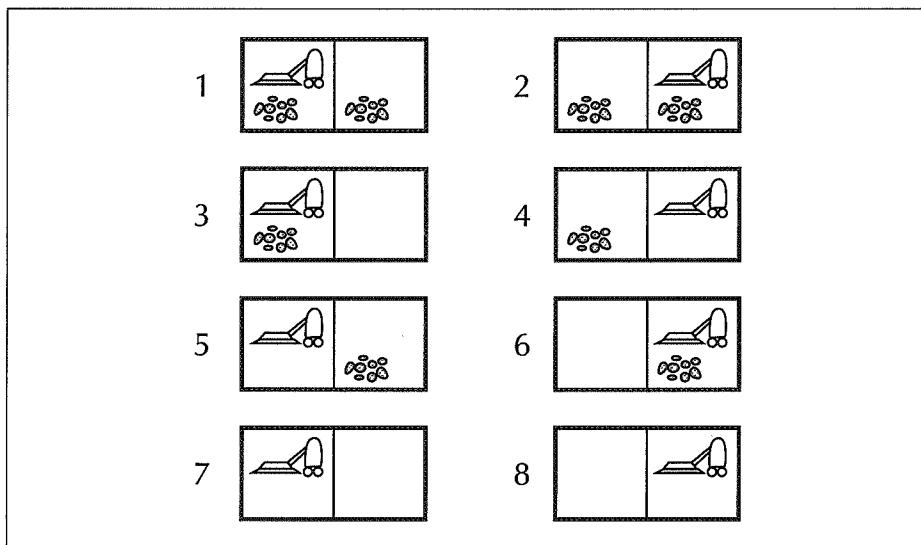
3.6 Ricerca con informazione parziale

Nel Paragrafo 3.3 abbiamo adottato l'ipotesi che l'ambiente sia completamente osservabile e deterministico, e che l'agente conosca l'effetto di ogni sua azione. In questa situazione l'agente può calcolare esattamente lo stato risultante da qualsiasi sequenza di azioni e conoscere sempre esattamente in quale stato si trova. Le sue percezioni dopo ogni azione, in effetti, non forniscono alcuna informazione aggiuntiva utile. Che cosa succede quando la conoscenza degli stati o delle azioni è incompleta? Tipi diversi di incompletezza portano a tre categorie distinte di problemi.

- 1. Problemi senza sensori** (chiamati anche **problemni conformanti**): se l'agente è del tutto privo di sensori, per quanto ne sa potrebbe trovarsi in uno tra più possibili stati iniziali, e ogni azione potrebbe quindi portare a uno tra molti possibili stati successori.
- 2. Problemi di contingenza:** se l'ambiente è parzialmente osservabile o le azioni sono incerte, le percezioni dell'agente forniscono effettivamente *nuove* informazioni dopo ogni azione. Ogni possibile percezione definisce una contingenza che dev'essere pianificata. Un problema è chiamato **con avversari** se l'incertezza è causata dalle azioni di un altro agente.
- 3. Problemi di esplorazione:** quando gli stati e le azioni dell'ambiente sono sconosciuti, l'agente deve fare in modo di scoprirlle. I problemi di esplorazione possono essere considerati casi estremi di problemi di contingenza.

Figura 3.20

Gli otto possibili stati del mondo dell'aspirapolvere.



Come esempio, torneremo al mondo dell'aspirapolvere. Ricorderete che sono possibili otto stati, come si vede nella Figura 3.20. Ci sono tre possibili azioni – *Sinistra*, *Destra* e *Aspira* – e l'obiettivo è raccogliere tutto lo sporco (stati 7 e 8). Se l'ambiente è osservabile, deterministico e completamente conosciuto, il problema si può risolvere banalmente applicando uno qualsiasi degli algoritmi che abbiamo descritto. Ad esempio, se lo stato iniziale è 5, la sequenza di azioni [*Destra*, *Aspira*] raggiungerà lo stato obiettivo 8. Il resto di questo paragrafo considera le versioni senza sensori e con contingenza di questo problema. I problemi di esplorazione sono trattati nel Paragrafo 4.5, quelli con avversari nel Capitolo 6.

Problemi senza sensori

Supponiamo che l'agente aspirapolvere conosca gli effetti di tutte le sue azioni, ma non abbia sensori. Inoltre lo stato iniziale può essere uno qualsiasi degli stati nell'insieme {1, 2, 3, 4, 5, 6, 7, 8}. Si potrebbe supporre che la sua situazione sia disperata, ma in effetti l'agente se la può cavare bene: dato che conosce l'effetto delle sue azioni può calcolare, ad esempio, che l'azione *Destra* lo porterà in uno degli stati {2, 4, 6, 8} e che la sequenza di azioni [*Destra*, *Aspira*] terminerà sempre in uno dei sue stati {4, 8}. Infine, la sequenza [*Destra*, *Aspira*, *Sinistra*, *Aspira*] garantisce il raggiungimento dello stato obiettivo 7 indipendentemente dallo stato iniziale. Diciamo che l'agente può **forzare** il mondo nello stato 7, anche quando non sa da che stato è partito. Riassumendo, quando il mondo non è completamente osservabile l'agente deve ragionare in termini di *insiemi* di stati raggiungibili, e non di stati singoli. Ogni insieme viene chiamato **stato-credenza** (*belief state*), perché

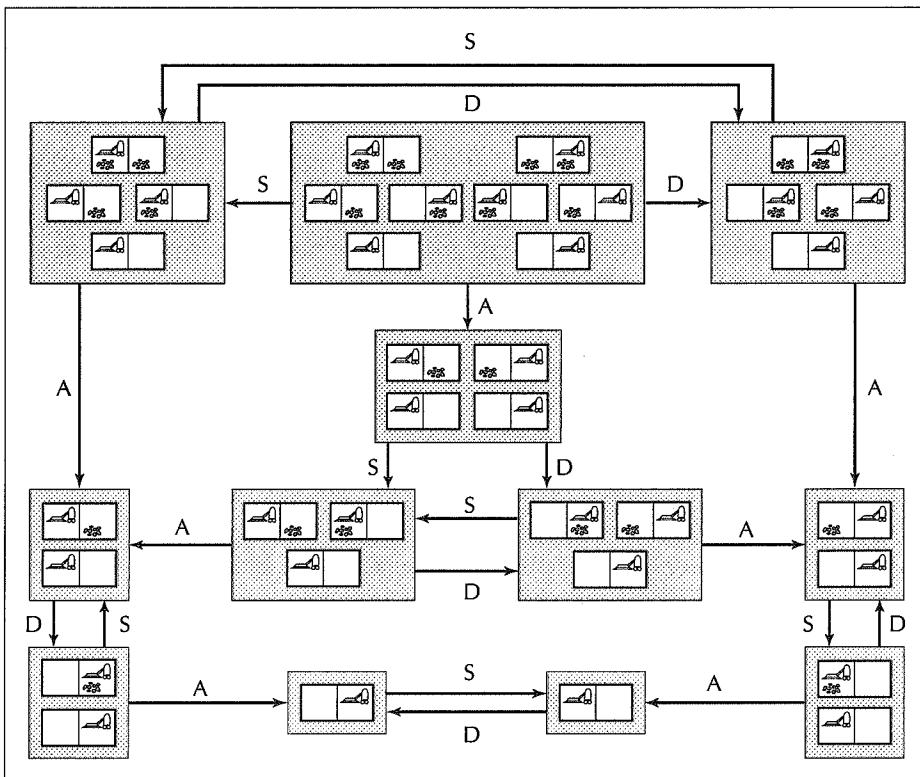


Figura 3.21 La porzione raggiungibile dello spazio degli stati-credenza per un mondo dell’aspirapolvere deterministico e privo di sensori. Ogni rettangolo ombreggiato corrisponde a un singolo stato-credenza. In ogni momento l’agente si trova in un particolare stato-credenza ma non conosce lo stato fisico corrispondente. Lo stato-credenza iniziale (totale ignoranza) corrisponde al rettangolo centrale in alto nella figura. Le azioni sono rappresentate da archi etichettati: gli auto-anelli sono omessi per chiarezza.

rappresenta la credenza corrente dell’agente circa i possibili stati in cui potrebbe trovarsi. In un ambiente completamente osservabile, ogni stato-credenza contiene esattamente uno stato fisico.

Per risolvere i problemi senza sensori occorre eseguire la ricerca nello spazio degli stati-credenza invece che negli stati fisici. Lo stato iniziale è uno stato-credenza, e ogni azione porta da uno stato-credenza a un altro. Per applicare un’azione a uno stato-credenza si deve calcolare l’unione dei risultati dell’applicazione di tale azione a tutti gli stati fisici che lo compongono. Un cammino ora collega tra loro stati-credenza, e una soluzione è un cammino che porta a uno stato-credenza *i cui membri sono tutti* stati obiettivo. La Figura 3.21 mostra lo spazio degli stati-credenza raggiungibili in un mondo aspirapolvere deterministico e privo di sensori. Ci sono solamente 12 stati-credenza raggiungibili, ma l’intero spazio degli stati-

credenza contiene ogni possibile combinazione di stati fisici, ovvero $2^8 = 256$ stati-credenza. In generale, se lo spazio degli stati fisici ha S stati, il corrispondente spazio degli stati-credenza ha cardinalità 2^S .

La nostra discussione dei problemi senza sensori, fin qui, ha dato per ipotesi che le azioni fossero deterministiche (ovvero che ogni azione potesse avere un solo risultato), ma l'analisi non cambia se l'ambiente è non deterministico. La ragione sta nel fatto che, privo di sensori, l'agente non può sapere qual è stato il risultato effettivo delle sue azioni, per cui i diversi possibili esiti non rappresentano altro che stati fisici aggiuntivi nello stato-credenza successore. Ad esempio, supponiamo che l'ambiente obbedisca alla Legge di Murphy: *talvolta* l'azione *Aspira* deposita sporco sul tappeto anziché pulirlo, *ma solo se non è già presente dello sporco*.⁶ A questo punto, se *Aspira* viene eseguita nello stato fisico 4 (v. Figura 3.20), i risultati possibili sono due, rappresentati dagli stati 2 e 4. Applicata allo stato-credenza iniziale, $\{1, 2, 3, 4, 5, 6, 7, 8\}$, *Aspira* ora porta allo stato-credenza corrispondente all'unione degli insiemi risultato degli otto stati fisici. Se lo calcoliamo, vediamo che quest'insieme è ancora $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Così, per un agente senza sensori nel mondo della Legge di Murphy, l'azione *Aspira* lascia lo stato-credenza immutato! In effetti, il problema non è risolvibile (v. Esercizio 3.18). Intuitivamente, la ragione sta nel fatto che l'agente non può sapere se il riquadro corrente è sporco, quindi non può sapere neppure se l'azione *Aspira* lo pulirà o aggiungerà altro sporco.

Problemi di contingenza

problema di contingenza

Quando l'ambiente è tale che l'agente può ottenere nuove informazioni dai suoi sensori dopo aver agito, l'agente si trova a fronteggiare un **problema di contingenza**. La soluzione a un tale problema prende spesso la forma di un *albero*, ogni ramo del quale può essere selezionato a seconda delle percezioni ricevute fino al nodo corrispondente. Ad esempio, supponiamo che l'agente si trovi nel mondo della Legge di Murphy e che abbia un sensore di posizione e uno per lo sporco locale, ma non possa rilevare lo sporco negli altri riquadri. Così, la percezione $[Sin, Sporco]$ significa che l'agente si trova in uno dei due stati $\{1, 3\}$. L'agente potrebbe formulare la sequenza di azioni $[Aspira, Destra, Aspira]$. La prima aspirazione cambierebbe lo stato in uno di $\{5, 7\}$, muoversi verso destra porterebbe poi in uno dei due stati $\{6, 8\}$. Eseguire l'azione finale *Aspira* nello stato 6 ci porterebbe allo stato obiettivo 8, ma se fossimo già in quello stato potremmo ritrovarci nello stato 6 (grazie alla Legge di Murphy), e in tal caso il piano fallirebbe.

⁶ Diamo per scontato che la maggior parte dei lettori si trovino a fronteggiare problemi simili e possano così simpatizzare con il nostro agente. Chiediamo scusa ai possessori di elettrodomestici moderni ed efficienti, che non potranno trarre vantaggio da questo espediente pedagogico.

Esaminando lo spazio degli stati-credenza di questa versione del problema, si può vedere facilmente che non esiste alcuna sequenza prefissata di azioni capace di garantire la soluzione del problema. La soluzione esiste, però, se non cerchiamo una sequenza di azioni *fissata a priori*:

[Aspira, Destra, if [Dx, Sporco] then Aspira].

Così lo spazio delle soluzioni è esteso in modo da includere la possibilità di scegliere azioni in base alle contingenze verificatesi durante l'esecuzione. Molti problemi nel mondo fisico reale sono contingenti, dato che è impossibile formulare predizioni esatte. Per questa ragione molte persone tengono gli occhi aperti mentre camminano o guidano.

I problemi di contingenza *talvolta* permettono soluzioni puramente sequenziali. Ad esempio, considerate un mondo della Legge di Murphy *completamente osservabile*. Le contingenze insorgono se l'agente esegue l'azione *Aspira* in un riquadro pulito, perché in seguito a ciò nuovo sporco potrebbe (ma non è detto) essere depositato nel riquadro. Finché l'agente non compie mai quell'azione, non sorgerà nessuna contingenza e ci sarà sempre una soluzione sequenziale per ogni stato iniziale (Esercizio 3.18).

Gli algoritmi che trattano i problemi di contingenza sono più complessi di quelli classici per la ricerca che abbiamo visto in questo capitolo; li analizzeremo nel Capitolo 12. Questi problemi portano anche alla progettazione di agenti diversi da quelli considerati fin qui, dato che possono agire *prima* di aver formulato un piano definitivo. Questo è molto comodo perché, invece di considerare in anticipo ogni contingenza che *potrebbe* verificarsi durante l'esecuzione, è spesso preferibile cominciare ad agire e vedere quali contingenze *sorgono effettivamente*. L'agente a questo punto può continuare a risolvere il problema prendendo in considerazione l'informazione aggiuntiva. Questo tipo di alternanza tra ricerca ed esecuzione, detto **interleaving**, è utile anche nei problemi di ricerca (v. Paragrafo 4.5) e nei giochi (v. Capitolo 6).

interleaving

3.7 Riepilogo

Questo capitolo ha presentato i metodi utilizzabili da un agente per scegliere le proprie azioni all'interno di ambienti deterministici, osservabili, statici e completamente noti. In tali casi, l'agente può costruire sequenze di azioni che raggiungono il suo obiettivo; questo processo si chiama **ricerca**.

- ♦ Prima di cominciare a cercare le soluzioni, un agente deve formulare un **obiettivo** e sulla base di quello specificare un **problema**.

- ◆ Un problema è composto da quattro parti: lo **stato iniziale**, un insieme di **azioni**, una funzione **test obiettivo** e una funzione **costo di cammino**. L'ambiente del problema è rappresentato dallo **spazio degli stati**. Un **cammino** attraverso lo spazio degli stati da quello iniziale a uno stato obiettivo è una **soluzione**.
- ◆ Per risolvere qualsiasi problema si può usare un singolo algoritmo generale, RICERCA-ALBERO; varianti specifiche di tale algoritmo implementano le diverse strategie.
- ◆ Gli algoritmi di ricerca sono valutati sulla base della loro **completezza**, **ottimalità**, **complessità temporale** e **complessità spaziale**. La complessità dipende da b , il fattore di ramificazione (*branching factor*) dello spazio degli stati, e d , la profondità (*depth*) della soluzione più vicina alla radice dell'albero.
- ◆ La **ricerca in ampiezza** espande sempre il nodo non ancora espanso meno profondo, ovvero più vicino alla radice dell'albero. È una strategia completa, ottima quando i passi hanno costo unitario, e ha una complessità spaziale e temporale $O(b^d)$. La complessità spaziale la rende inapplicabile alla maggior parte dei casi. La **ricerca a costo uniforme** è simile a quella in ampiezza ma espande il nodo che ha minor costo di cammino, $g(n)$. È completa e ottima se il costo di ogni passo è maggiore di un valore positivo ε .
- ◆ La **ricerca in profondità** espande sempre il nodo più profondo non ancora espanso. Non è completa né ottima, ha complessità temporale $O(b^m)$ e complessità spaziale $O(bm)$, dove m è la profondità massima dei cammini nello spazio di stato.
- ◆ La **ricerca a profondità limitata** è simile alla precedente, ma impone un limite prefissato alla profondità dell'albero.
- ◆ La **ricerca ad approfondimento iterativo** esegue una serie di ricerche a profondità limitata, estendendo progressivamente il limite finché non trova una soluzione. È completa, ottima quando i passi hanno costo unitario e ha complessità temporale $O(b^d)$ e spaziale $O(bd)$.
- ◆ La **ricerca bidirezionale** può ridurre enormemente la complessità temporale, ma non è sempre applicabile e può richiedere troppa memoria.
- ◆ Quando lo spazio degli stati è un grafo invece di un albero, può essere conveniente controllare se uno stato è replicato più volte nell'albero di ricerca. L'algoritmo RICERCA-GRAFO elimina tutti gli stati ripetuti.
- ◆ Quando l'ambiente è solo parzialmente osservabile l'agente può applicare gli algoritmi di ricerca allo spazio degli **stati-credenza**, definiti come insiemi di possibili stati fisici in cui l'agente potrebbe trovarsi. Alcune volte potrebbe essere possibile costruire singole sequenze di azioni risolutive; in altri casi l'agente necessiterà di un **piano di contingenza** per gestire le circostanze sconosciute che potranno verificarsi.

Note storiche e bibliografiche

La maggior parte dei problemi di ricerca nello spazio degli stati analizzati in questo capitolo hanno una lunga storia nella letteratura e sono meno banali di quanto possono sembrare. Il problema dei missionari e dei cannibali, proposto nell'Esercizio 3.9, è stato analizzato in dettaglio da Amarel (1968). In precedenza era già stato considerato nell'ambito dell'IA da Simon e Newell (1961), e nella ricerca operativa da Bellman e Dreyfus (1962). Studi come questi, e il lavoro di Newell e Simon su Logic Theorist (1957) e GPS (1961), portarono negli anni '60 all'istituzione degli algoritmi di ricerca come strumento principale nell'arsenale dei ricercatori di intelligenza artificiale, e della risoluzione di problemi come sua attività precipua. Sfortunatamente, ben poco lavoro fu dedicato all'automazione della formulazione dei problemi. Una discussione più recente della rappresentazione dei problemi e della loro astrazione, che include programmi che automatizzano in parte queste attività, si trova in Knoblock (1990).

Il rompicapo a 8 tasselli è il fratello minore di quello a 15, ideato dal famoso inventore di giochi americano Sam Loyd (1959) negli anni '70 del 1800. Negli Stati Uniti il rompicapo a 15 tasselli ottenne subito un'immensa popolarità, paragonabile al successo recente del Cubo di Rubik: non mancò di attirare anche l'attenzione dei matematici (Johnson e Story, 1879; Tait, 1880). I curatori dell'*American Journal of Mathematics* scrissero: "Il rompicapo a 15 tasselli, nelle ultime settimane, ha goduto di una posizione preminente nell'attenzione del pubblico americano, e si può affermare con certezza che ha interessato nove persone su dieci della comunità, di entrambi i sessi e di tutte le età. Ma tutto questo non sarebbe bastato a includere articoli dedicati a un siffatto argomento nell'*American Journal of Mathematics*, se non fosse per il fatto che..." (seguiva un riassunto degli aspetti interessanti del rompicapo da un punto di vista matematico). Un'analisi esaustiva del rompicapo a 8 tasselli fu svolta, con l'aiuto del computer, da Schofield (1967). Ratner e Warmuth (1986) mostrarono che la versione generale $n \times n$ del rompicapo a tasselli mobili appartiene alla classe di problemi NP-completi.

Il problema delle 8 regine fu pubblicato per la prima volta nel 1848 in forma anonima, nella rivista tedesca di scacchi *Schach*; più tardi fu attribuito a un certo Max Bezzel. Fu ripubblicato nel 1850 e attirò l'attenzione del famoso matematico Carl Friedrich Gauss, che cercò di enumerare tutte le possibili soluzioni, ma ne trovò solo 72. Nauck pubblicò tutte le 92 soluzioni più tardi quello stesso anno. Netto (1901) generalizzò il problema delle n regine, e infine Abramson e Yung (1989) trovarono un algoritmo di complessità $O(n)$.

Ognuno dei problemi di ricerca nel mondo reale presentati in questo capitolo sono stati oggetto di un grande sforzo di ricerca. I metodi per selezionare i voli di linea in modo ottimo rimangono in gran parte proprietari, ma Carl de Marcken (in una comunicazione personale) ha dimostrato che le tariffe delle compagnie aeree e le restrizioni imposte sono diventate così complesse e involute che scegliere un volo ottimo è oggi formalmente *indecidibile*. Il problema del commesso viaggiatore è un classico problema combinatorio dell'informatica teorica (Lawler,

1985; Lawler et al., 1992). Karp (1972) ha dimostrato che TSP è un problema NP-difficile, ma per trattarlo sono stati sviluppati diversi metodi euristici approssimati (Lin e Kernighan, 1973). Arora (1998) ha definito uno schema di approssimazione completamente polinomiale per i problemi TSP euclidei. Shahookar e Mazumder (1991) hanno scritto una presentazione generale dei metodi di configurazione VLSI, e le riviste specializzate includono molti articoli sull'argomento.

Gli algoritmi di ricerca non informata per la risoluzione di problemi sono un argomento centrale dell'informatica classica (Horowitz e Sahni, 1978) e della ricerca operativa (Dreyfus, 1969); Deo e Pang (1984) e Gallo e Pallottino (1988) offrono panoramiche più recenti. La ricerca in ampiezza fu formulata da Moore (1959) per risolvere labirinti. Il metodo della **programmazione dinamica** (Bellman e Dreyfus, 1962), che registra sistematicamente le soluzioni di tutti i sottoproblemi di lunghezza crescente, può essere considerato una forma di ricerca in ampiezza su grafo. L'algoritmo di Dijkstra (1959) che trova il cammino minimo tra due punti è all'origine della ricerca a costo uniforme.

Una versione dell'approfondimento iterativo progettata per utilizzare al meglio un orologio da scacchi fu usata per la prima volta da Slate e Atkin (1977) nel programma scacchistico CHESS 4.5, ma l'applicazione alla ricerca di cammini minimi su grafo è dovuta a Korf (1985a). Anche la ricerca bidirezionale, introdotta da Pohl (1969, 1971), in alcuni casi può essere molto efficiente.

Nell'ambito della risoluzione di problemi gli ambienti parzialmente osservabili e non deterministici non sono stati studiati molto a fondo. Alcuni problemi di efficienza della ricerca nello spazio degli stati-credenza sono stati investigati da Genesereth e Nourbakhsh (1993). Koenig e Simmons (1998) hanno studiato la navigazione robotica con posizione iniziale sconosciuta, mentre Erdmann e Mason (1988) hanno esaminato il problema della manipolazione robotica senza sensori, usando una forma continua della ricerca negli stati-credenza. La ricerca con contingenza è stata studiata all'interno della pianificazione (v. Capitolo 12). Nella maggior parte dei casi, la pianificazione e l'azione con informazione incerta sono state gestite usando gli strumenti della probabilità e della teoria delle decisioni (v. Capitolo 17, 2^o vol.).

I libri di Nilsson (1971, 1980) rappresentano buone fonti generali di informazione sugli algoritmi di ricerca classici. Una panoramica esaustiva e più aggiornata può essere trovata in Korf (1988). Articoli dedicati ai nuovi algoritmi di ricerca – che, in modo sorprendente, continuano a essere scoperti – sono pubblicati su riviste come *Artificial Intelligence*.

Esercizi

- 3.1 Definite con parole vostre i seguenti termini: stato, spazio degli stati, albero di ricerca, nodo di ricerca, obiettivo, azione, funzione successore e fattore di ramificazione.
- 3.2 Spiegate perché la formulazione dei problemi deve sempre seguire quella dell'obiettivo.
- 3.3 Supponiamo che $AZIONI\text{-LEGALI}(s)$ denoti l'insieme di azioni legali nello stato s e $RISULTATO(a, s)$ denoti lo stato che risulta dall'esecuzione dell'azione legale a nello stato s . Definite **FUNZIONE-SUCCESSORE** in termini di **AZIONI-LEGALI** e **RISULTATO**, e *viceversa*.
- 3.4 Mostrate che gli stati del rompicapo a 8 tasselli sono suddivisi in due insiemi disgiunti, in modo tale che nessuno stato di un insieme può essere trasformato in uno stato dell'altro insieme con un numero qualsiasi di mosse (*suggerimento*: v. Berlekamp et al. (1982)). Scrivete una procedura che dica a quale classe appartiene un determinato stato, e spiegate perché questa è una buona cosa da avere quando si generano stati casuali.
- 3.5 Considerate il problema a n regine usando la formulazione incrementale “efficiente” data a pag. 90. Spiegate perché la dimensione dello spazio degli stati è almeno $\sqrt[3]{n!}$ e stimate il più grande valore di n per cui si può eseguire un'esplorazione esaustiva (*suggerimento*: calcolate un limite inferiore del fattore di ramificazione considerando il numero massimo di caselle che una regina può attaccare in una qualsiasi colonna).
- 3.6 Uno spazio degli stati finito porta sempre a un albero di ricerca finito? E se lo spazio degli stati finito è un albero? Potete essere più precisi riguardo ai tipi di spazi degli stati che portano sempre ad alberi di ricerca finiti? (Adattato da Bender, 1996).
- 3.7 Scrivete lo stato iniziale, il test obiettivo, la funzione successore e quella di costo per ognuno dei seguenti problemi. Scegliete una formulazione abbastanza precisa da poter essere implementata.
 - a. Dovete colorare una mappa usando solo quattro colori, in modo tale che non ci siano regioni adiacenti con lo stesso colore.
 - b. Una scimmia alta 90 cm. si trova in una stanza con delle banane appese al soffitto, che è alto 3 metri. La scimmia vorrebbe raggiungere le banane. La stanza contiene due casse alte ognuna 1 metro; le casse si possono muovere, scalare, impilare.
 - c. Avete un programma che stampa il messaggio “dato illegale in input” quando gli si passa un certo file di dati. Voi sapete che l'elaborazione di ogni dato è indipendente dagli altri. Volete scoprire quale dato è illegale.

d. Avete tre brocche rispettivamente da 12, 8 e 3 litri e un rubinetto. Potete riempire le brocche fino all'orlo e svuotarle una dentro l'altra o nel lavandino. Dovete misurare esattamente un litro d'acqua.

3.8 Considerate uno spazio degli stati in cui lo stato iniziale abbia il numero 1 e la funzione successore dello stato n restituiscia due stati, numerati $2n$ e $2n + 1$.

a. Disegnate la porzione di spazio degli stati per gli stati che vanno da 1 a 15.
b. Supponete che lo stato obiettivo sia 11. Scrivete l'ordine di visita dei nodi per la ricerca in ampiezza, la ricerca a profondità limitata con limite 3 e la ricerca ad approfondimento iterativo.

c. Per questo problema sarebbe appropriata la ricerca bidirezionale? Se la risposta è sì, descrivete in dettaglio come funzionerebbe.

d. Qual è il fattore di ramificazione in ogni verso della ricerca bidirezionale?

e. La risposta al punto (c) vi suggerisce una riformulazione che vi permetterebbe di risolvere il problema di andare dallo stato 1 a un qualsiasi stato obiettivo senza quasi svolgere ricerca?

3.9 Il problema dei **missionari e cannibali** è solitamente descritto come segue. Tre missionari e tre cannibali si trovano sulla riva di un fiume, e hanno una barca che può portare una o due persone. Trovate il modo di portare tutti dall'altra parte, senza mai permettere che su una delle due rive il numero dei missionari sia inferiore a quello dei cannibali. Questo problema è famoso nel campo dell'IA perché era l'argomento del primo articolo che presentò la formulazione dei problemi da un punto di vista analitico (Amarel, 1968).

a. Formulate il problema precisamente, facendo solo le distinzioni necessarie ad assicurare una soluzione valida. Disegnate un diagramma completo dello spazio degli stati.

b. Implementate e risolvete il problema in modo ottimo usando un appropriato algoritmo di ricerca. È una buona idea controllare se ci sono stati ripetuti?

c. Per quale ragione pensate che la gente trovi difficile risolvere questo rompicapo, quando lo spazio degli stati è così semplice?

3.10 Implementate due versioni della funzione successore per il rompicapo a 8 tasselli: una deve generare tutti i successori in una volta, copiando e modificando la struttura dati del rompicapo; l'altra deve generare un solo nuovo successore a ogni invocazione modificando direttamente lo stato padre (e annullando le modifiche quando necessario). Usando queste due funzioni scrivete due versioni della ricerca ad approfondimento iterativo e confrontate le loro prestazioni.

3.11 A pag. 106 abbiamo menzionato la **ricerca a lunghezza iterativa**, versione iterativa della ricerca a costo uniforme. L'idea è di incrementare progressivamente un limite al costo dei cammini: se viene generato un nodo il cui co-

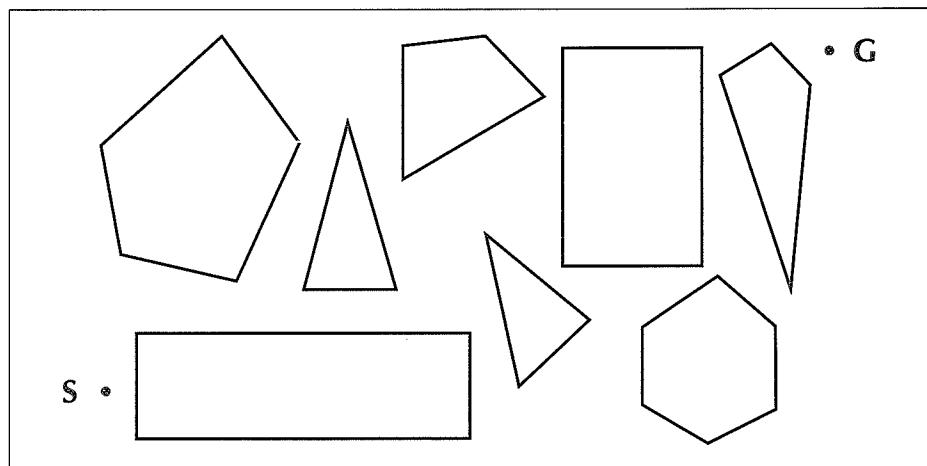
sto di cammino supera il limite corrente, esso viene immediatamente scartato. A ogni nuova iterazione, il limite viene impostato al costo minimo tra tutti i costi dei nodi scartati nell'iterazione precedente.

- a. Dimostrate che quest'algoritmo è ottimo.
 - b. Considerate un albero uniforme con fattore di ramificazione b , profondità della soluzione d e costi di passo unitari. Quante iterazioni richiederà la ricerca a lunghezza iterativa?
 - c. Ora considerate che ogni passo abbia un costo compreso nell'intervallo continuo $[0, 1]$, con un costo minimo positivo ε . Quante iterazioni saranno necessarie nel caso pessimo?
 - d. Implementate l'algoritmo e applicatelo a istanze del rompicapo a 8 tasselli e del problema del commesso viaggiatore. Confrontate le prestazioni dell'algoritmo a quelle di una ricerca a costo uniforme e commentate i risultati.
- 3.12 Dimostrate che la ricerca a costo uniforme e quella in ampiezza con costo di passo costante sono ottime quando si usa l'algoritmo RICERCA-GRAFO. Mostrate uno spazio degli stati con costo di passi costante in cui RICERCA-GRAFO, usando la ricerca ad approfondimento iterativo, trova una soluzione subottima.
- 3.13 Descrivete uno spazio degli stati in cui la ricerca ad approfondimento iterativo ha prestazioni molto inferiori a quelle della ricerca in profondità (ad esempio, $O(n^2)$ contro $O(n)$).
- 3.14 Scrivete un programma che prenda come input due URL (indirizzi di pagine web) e trovi un cammino fatto di collegamenti che vada dall'una all'altra. Qual è la strategia di ricerca più appropriata? Usare la ricerca bidirezionale è una buona idea? Si potrebbe usare un motore di ricerca per implementare una funzione predecessore?
- 3.15 Considerate il problema di trovare il cammino più breve tra due punti su un piano che ha ostacoli poligonali convessi, come si vede nella Figura 3.22. Questa è un'astrazione del problema che si trova ad affrontare un robot che deve navigare in un ambiente affollato.
- a. Supponiamo che lo spazio degli stati consista in tutte le posizioni (x, y) sul piano. Quanti stati ci sono? Quanti cammini verso l'obiettivo?
 - b. Spiegate brevemente perché il cammino minimo dal vertice di un poligono a uno qualsiasi degli altri deve consistere in una serie di segmenti di retta che uniscono alcuni dei vertici dei poligoni sulla scena. Ora definite un nuovo, migliore spazio degli stati. Quanto è grande quest'ultimo?
 - c. Definite le funzioni necessarie per implementare il problema di ricerca, includendo una funzione successore che prende un vertice come input e restituisce l'insieme di vertici che possono essere raggiunti partendo da esso e camminando in linea retta (non dimenticate i vertici adiacenti sul-



Figura 3.22

Uno scenario con ostacoli poligonali.



lo stesso poligono). Usate la distanza in linea retta come funzione euristica (v. cap. successivo).

- d. Applicate uno o più algoritmi descritti nel capitolo per risolvere vari problemi in questo dominio, e commentate le loro prestazioni.
- 3.16** Possiamo trasformare in un ambiente il problema di navigazione dell'Esercizio 3.15 nel modo seguente:
- ◆ le percezioni saranno una serie di posizioni, *relative all'agente*, di vertici visibili. Le percezioni *non* includono la posizione del robot! Il robot deve dedurre la propria posizione dalla mappa; per adesso potete supporre che ogni posizione abbia una "vista" differente;
 - ◆ ogni azione consisterà in un vettore che descrive la direzione da seguire in linea retta. Se il cammino non incontra ostacoli, l'azione avrà successo; altrimenti il robot si fermerà nel primo punto in cui il suo cammino incontra un ostacolo. Se l'agente restituisce un vettore di movimento nullo ed è sull'obiettivo (la cui posizione è fissa e conosciuta), l'ambiente dovrebbe teletrasportare il robot in una *posizione casuale* (che non si trovi dentro a un ostacolo);
 - ◆ la misura di prestazione toglie all'agente 1 punto per ogni unità spaziale attraversata e assegna 1000 punti ogni volta che raggiunge l'obiettivo.
- a. Implementate questo ambiente e un agente risolutore di problemi adatto. L'agente dovrà formulare un nuovo problema ogni volta che viene teletrasportato; tale problema dovrà includere la scoperta della sua nuova posizione.
 - b. Documentate le prestazioni del vostro agente (facendo sì che questo generi appropriati commenti durante i suoi movimenti) e scrivete un rapporto della sua performance su 100 episodi.

- c. Modificate l'ambiente in modo tale che, il 30% delle volte, l'agente finisce in una destinazione inattesa (scelta casualmente tra gli altri vertici visibili, se ce ne sono, altrimenti l'agente non si muoverà affatto). Questo è un modello molto rozzo degli errori di movimento dei robot reali. Modificate l'agente in modo che, quando viene rilevato un simile errore, determini nuovamente la sua posizione e costruisca un piano per tornare dov'era in modo da riprendere il vecchio piano. Ricordate che talvolta potrà fallire anche il tentativo di tornare indietro! Mostrate un esempio di un agente che supera con successo due errori consecutivi di movimento e riesce lo stesso a raggiungere il suo obiettivo.
- d. Ora provate due schemi diversi di recupero dopo un errore di movimento: (1) puntare verso il vertice più vicino appartenente al cammino originale; (2) pianificare un itinerario completamente nuovo dalla nuova posizione all'obiettivo. Confrontate le prestazioni dei tre schemi di recupero. Se includeste i costi di ricerca il confronto cambierebbe?
- e. Ora supponete che ci siano diverse posizioni con la stessa "vista" dei poligoni (ad esempio, se la scena è una griglia con ostacoli quadrati). Quale tipo di nuovi problemi deve affrontare ora l'agente? Che aspetto hanno le soluzioni?
- 3.17 A pag. 86 abbiamo detto che non avremmo considerato problemi con passi di costo negativo. In questo esercizio approfondiremo quest'aspetto.
- Supponete che le azioni possano avere costi negativi arbitrariamente grandi; spiegate perché questa possibilità obbligherebbe ogni algoritmo ottimo a esplorare l'intero spazio degli stati.
 - È un aiuto supporre che i costi dei passi debbano essere maggiori o uguali a una qualche costante negativa c ? Considerate sia il caso degli alberi che dei grafi.
 - Supponete che un insieme di azioni formi un ciclo in modo tale che eseguendole tutte in un dato ordine si ritorni allo stato di partenza. Se tutti i passi hanno costo negativo, che effetto si avrà sul comportamento ottimo di un agente in quell'ambiente?
 - Si possono facilmente immaginare azioni con un costo negativo molto alto, anche in domini come la ricerca di itinerari. Ad esempio, alcuni tratti di strada potrebbero offrire un paesaggio così stupendo da controbilanciare ampiamente i costi in termini di tempo e carburante. Spiegate precisamente, nel contesto della ricerca nello spazio degli stati, perché gli esseri umani non continuano a guidare in cerchio intorno a panorami bellissimi, e spieghate come definire lo spazio degli stati e le azioni in modo tale che anche gli agenti artificiali possano evitarlo.
 - Potete pensare a un dominio reale in cui i costi dei passi siano tali da causare dei cicli?

- 
- 3.18 Considerate un mondo dell'aspirapolvere a due locazioni, senza sensori e soggetto alla Legge di Murphy. Disegnate lo spazio degli stati-credenza raggiungibile dallo stato-credenza iniziale $\{1, 2, 3, 4, 5, 6, 7, 8\}$ e spiegate perché il problema è insolubile. Dimostrate anche che in un mondo completamente osservabile c'è una sequenza di azioni risolutiva per ogni possibile stato iniziale.
- 3.19 Considerate il mondo dell'aspirapolvere definito nella Figura 2.2.
- Quale degli algoritmi definiti in questo capitolo sarebbe più appropriato per questo problema? L'algoritmo dovrebbe controllare la presenza di stati ripetuti?
 - Applicate l'algoritmo prescelto per calcolare una sequenza ottima di azioni in un mondo 3×3 il cui stato iniziale ha i tre riquadri più in alto sporchi e l'agente nel centro.
 - Costruite un agente per il mondo aspirapolvere e valutate le sue prestazioni in un insieme di mondi 3×3 con probabilità 0.2 che ci sia sporco in ogni riquadro. Nella misura delle prestazioni includete il costo di ricerca oltre a quello del cammino, applicando un ragionevole fattore di conversione.
 - Confrontate il vostro migliore agente con ricerca a un agente reattivo semplice randomizzato, che aspira se rileva sporco nel riquadro oppure si muove casualmente.
 - Considerate cosa accadrebbe se il mondo fosse ingrandito alle dimensioni $n \times n$. Al variare di n , come variano le prestazioni dell'agente con ricerca e di quello reattivo?

Capitolo 4

Ricerca informata ed esplorazione

Nel quale vediamo come l'informazione sullo spazio degli stati può aiutare gli algoritmi a non brancolare nel buio.

Il Capitolo 3 ha mostrato che le strategie di ricerca non informata possono trovare la soluzione dei problemi generando sistematicamente nuovi stati e verificando se corrispondono all'obiettivo. Sfortunatamente queste strategie sono, nella maggior parte dei casi, incredibilmente inefficienti. Questo capitolo mostra che una strategia di ricerca informata, che utilizza conoscenze specifiche del problema, può trovare soluzioni in modo più efficiente. Il Paragrafo 4.1 descrive le versioni informate degli algoritmi del Capitolo 3, il Paragrafo 4.2 spiega come si può ottenere la necessaria informazione specifica del problema. I Paragrafi 4.3 e 4.4 presentano algoritmi che eseguono puramente una ricerca locale nello spazio degli stati, valutando e modificando uno o più degli stati correnti invece di esplorare sistematicamente i cammini che partono da uno stato iniziale. Questi algoritmi sono adatti per i problemi in cui il costo di cammino è irrilevante e quello che conta è solo la soluzione. La famiglia di algoritmi di ricerca locale include metodi ispirati alla fisica statistica (simulated annealing) e alla biologia dell'evoluzione (algoritmi genetici). Infine, il Paragrafo 4.5 investiga la ricerca online, in cui l'agente si trova a fronteggiare uno spazio degli stati completamente sconosciuto.

4.1 Strategie di ricerca informata o euristica

ricerca informata

Questo paragrafo mostra come una strategia di ricerca informata, che sfrutta conoscenza specifica del problema al di là della semplice definizione, può trovare soluzioni in modo più efficiente di una strategia non informata.

ricerca best-first

funzione di valutazione

L'approccio generale che prenderemo in considerazione è chiamato **ricerca best-first** (letteralmente, "prima il migliore"). La ricerca best-first è un'istanza dell'algoritmo generale RICERCA-ALBERO o RICERCA-GRAFO in cui il nodo da espandere viene scelto in base a una **funzione di valutazione $f(n)$** . Solitamente viene selezionato il nodo con la valutazione **più bassa**, perché la funzione misura la distanza dall'obiettivo. La ricerca best-first può essere implementata all'interno della nostra infrastruttura generale di ricerca utilizzando una coda con priorità, una struttura dati che memorizza la frontiera in ordine di f -valori.

→ Il nome "ricerca best-first" è classico, ma impreciso. Dopotutto, se potessimo veramente espandere sempre per primo il **nodo migliore**, non ci sarebbe affatto ricerca ma solo una marcia diretta verso l'obiettivo. Tutto quello che possiamo fare è scegliere il **nodo che sembra più promettente** di tutti gli altri secondo la funzione di **valutazione**. Se quest'ultima è assolutamente accurata, tale nodo sarà effettivamente **quello migliore**; nella realtà la funzione di valutazione potrà fare errori e condurre la ricerca su cammini divergenti. Nonostante ciò continueremo a usare il nome "ricerca best-first", anche perché "ricerca-del-nodo-che-sembra-quello-migliore" è una locuzione piuttosto involuta.

funzione euristica

 $h(n)$

C'è un'intera famiglia di algoritmi RICERCA-BEST-FIRST con diverse funzioni di valutazione.¹ Un componente chiave di questi algoritmi è la **funzione euristica²** denominata $h(n)$:

$h(n)$ = costo stimato del cammino più conveniente dal nodo n a un nodo obiettivo.

Ad esempio, in Romania, si potrebbe stimare il costo del cammino più conveniente da Arad a Bucarest usando la loro distanza in linea d'aria.

Le funzioni euristiche sono la forma più comune per inserire conoscenza aggiuntiva nell'algoritmo di ricerca: le studieremo più approfonditamente nel Paragrafo 4.2. Per adesso ci limiteremo a considerarle funzioni arbitrarie specifiche del

¹ L'Esercizio 4.3 vi chiede di mostrare che questa famiglia include diversi algoritmi di ricerca non informata a noi familiari.

² Una funzione euristica $h(n)$ prende un *nodo* come input, ma dipende solamente dallo *stato* in quel nodo.

problema, con un solo vincolo: se n è un nodo obiettivo, allora $h(n) = 0$. Il resto di questo paragrafo tratterà due modi di usare l'informazione euristica per guidare la ricerca.

Ricerca best-first greedy o "golosa"

La ricerca best-first greedy³ (il termine viene talvolta tradotto in italiano come "golosa" o "avida") cerca sempre di espandere il nodo più vicino all'obiettivo, sulla base del fatto che è probabile che questo porti rapidamente a una soluzione. Di conseguenza la valutazione dei nodi viene fatta applicando direttamente la funzione euristica: $f(n) = h(n)$.

Vediamo come funziona con il nostro problema dell'itinerario in Romania usando l'euristica della distanza in linea d'aria, che indicheremo con h_{DLA} . Se l'obiettivo è Bucarest, dovremo conoscere tutte le distanze in linea d'aria verso quella città, che sono elencate nella Figura 4.1. Per esempio, $h_{DLA}(In(Arad)) = 366$. Notate che i valori di h_{DLA} non possono essere calcolati direttamente dalla descrizione del problema. Inoltre, serve una certa esperienza per sapere che h_{DLA} è effettivamente correlata alle distanze su strada e rappresenta quindi un'euristica utile.

La Figura 4.2 mostra i progressi di una ricerca best-first greedy che usa h_{DLA} per trovare un cammino da Arad a Bucarest. Il primo nodo espanso da Arad sarà Sibiu, perché è più vicino a Bucarest di Zerind e Timisoara. Il nodo successivo sarà Fagaras, ancora una volta perché è il più vicino in linea d'aria. Fagaras a sua volta

ricerca best-first
greedy

Arad	366	Mehadia	241
Bucarest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figura 4.1
Valori di h_{DLA} –
distanze in linea
d'aria verso
Bucarest.

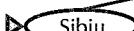
³ La prima edizione di questo libro la chiamava semplicemente ricerca greedy o golosa; altri autori l'hanno chiamata ricerca best-first. Noi usiamo quest'ultimo termine in modo più generale, seguendo Pearl (1984).

(a) Stato iniziale



366

(b) Dopo l'espansione di Arad



253



329



374

(c) Dopo l'espansione di Sibiu



329



374

366

176

380

193

(d) Dopo l'espansione di Fagaras



329



374

366

380

193

253

0

Figura 4.2 Fasi di una ricerca best-first greedy verso Bucarest che usa l'euristica della distanza in linea d'aria h_{DLA} . I nodi sono etichettati con i loro h -valori.

genererà Bucarest, che è l'obiettivo. In questo particolare problema, la ricerca best-first greedy che usa h_{DLA} trova una soluzione senza mai neppure espandere un nodo che non sia sul cammino della soluzione; ne consegue che il costo della ricerca è minimo. Tuttavia, la soluzione non è ottima: il cammino che passa da Sibiu e Fagaras è 32 chilometri più lungo di quello che attraversa Rimnicu Vilcea e Pitesti. Questo dimostra perché l'algoritmo è chiamato “goloso”: a ogni passo cerca sempre di arrivare il più possibile vicino all'obiettivo.

Minimizzare $h(n)$ può portare a false partenze. Considerate il problema di andare da Iasi a Fagaras. L'euristica suggerisce di espandere per primo Neamt, perché è il nodo più vicino a Fagaras; questo però si rivela un vicolo cieco. La soluzione richiede di andare per prima cosa a Vaslui, un passo che secondo l'euristica ci porta addirittura più lontano dall'obiettivo; da lì potremo continuare per Urziceni, Bucarest e infine Fagaras. In questo caso, quindi, l'euristica ha causato l'espansione di nodi non necessari. Inoltre se non rileviamo la presenza di stati ripetuti la soluzione non verrà mai trovata, perché la ricerca continuerà a oscillare tra Neamt e Iasi.

La ricerca best-first greedy assomiglia a quella in profondità perché preferisce seguire un singolo cammino fino all'obiettivo, tornando indietro quando incontra un vicolo cieco. I suoi difetti sono gli stessi della ricerca in profondità: non è ottima e inoltre è incompleta, perché può imboccare un cammino infinito e non tornare mai indietro per verificare le altre possibilità. Nel caso pessimo, la complessità temporale e spaziale è $O(b^m)$, dove m è la profondità massima dello spazio di ricerca. Con una buona funzione euristica, comunque, la complessità può essere ridotta considerevolmente: l'entità precisa della riduzione dipende dalla natura del problema e dalla qualità dell'euristica.

Ricerca A*: minimizzare il costo totale stimato della soluzione

La forma più diffusa di ricerca best-first è la ricerca A* (si dice "ricerca A-stella"). La valutazione dei nodi viene eseguita combinando $g(n)$, il costo per raggiungere il nodo, e $h(n)$, il costo per andare da lì all'obiettivo:

$$f(n) = g(n) + h(n).$$

Dal momento che $g(n)$ fornisce il costo di cammino dal nodo iniziale al nodo n , e $h(n)$ rappresenta il costo stimato del cammino più conveniente da n all'obiettivo, risulta

$$f(n) = \text{costo stimato della soluzione più conveniente che passa per } n.$$

Se stiamo cercando di trovare la soluzione meno costosa, quindi, una cosa ragionevole è provare per primo il nodo col valore più basso di $g(n) + h(n)$. In effetti risulta che questa strategia è molto più che ragionevole: a patto che la funzione euristica $h(n)$ soddisfi certe condizioni, la ricerca A* è sia completa che ottima.

L'ottimalità di A* si vede subito se applicata a RICERCA-ALBERO. In questo caso, A* è ottima se $h(n)$ è una euristica ammissibile, ovvero se $h(n)$ non sbaglia mai per eccesso la stima del costo per arrivare all'obiettivo. Le euristiche ammissibili sono ottimiste per natura, perché pensano che il costo della soluzione del problema sia inferiore a quello reale. Dal momento che $g(n)$ è il costo esatto del cammino che arriva a n , ne consegue direttamente che $f(n)$ non sopravvaluta mai il costo reale di una soluzione che passa per il nodo n .

ricerca A*

euristica ammissibile

Un esempio palese di euristica ammissibile è la distanza il linea d'aria h_{DLA} che abbiamo usato per arrivare a Bucarest. Questa euristica è ammissibile perché la distanza minima tra due punti è proprio il segmento di retta che li congiunge, e quindi la distanza in linea d'aria non può mai peccare per eccesso. Nella Figura 4.3 è illustrato il progresso di un albero di ricerca A* per Bucarest. I valori di g sono calcolati sulla base dei costi dei passi riportati nella mappa della Figura 3.2, mentre i valori di h_{DLA} sono elencati nella Figura 4.1. Notate in particolare che Bucarest compare per la prima volta sulla frontiera del passo (e) ma non viene scelto per l'espansione perché il suo f -costo (450) è maggiore di quello di Pitesti (417). Un altro modo di esprimere questo fatto è dire che potrebbe esserci una soluzione che passa da Pitesti il cui costo potrebbe arrivare a costare 417, per cui l'algoritmo non si accontenterà di una soluzione che costa 450. Da quest'esempio possiamo estrarre la prova generale che usando RICERCA-ALBERO A* risulta ottima se $h(n)$ è ammissibile. Supponiamo che sulla frontiera appaia un nodo obiettivo subottimo G_2 , e sia C^* il costo della soluzione ottima. Allora, dato che G_2 è subottimo e che $h(G_2) = 0$ (vero per ogni nodo obiettivo), sappiamo che

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*.$$

Ora considerate un nodo n della frontiera che si trovi sul cammino di una soluzione ottima: ad esempio Pitesti nel caso appena presentato (se una soluzione esiste, un nodo simile deve sempre esistere). Se $h(n)$ non sopravvaluta il costo del completamento del cammino della soluzione, allora sappiamo che

$$f(n) = g(n) + h(n) \leq C^*.$$

Ora abbiamo mostrato che $f(n) \leq C^* < f(G_2)$, cosicché G_2 non sarà espanso e A* dovrà restituire una soluzione ottima.

Se invece di RICERCA-ALBERO usiamo l'algoritmo RICERCA-GRAFO della Figura 3.19 questa dimostrazione non vale più: potranno essere restituite soluzioni subottime perché, se il cammino ottimo verso uno stato ripetuto non è il primo generato, RICERCA-GRAFO potrà scartarlo (v. Esercizio 4.4). Ci sono due modi per risolvere questo problema. La prima soluzione è estendere RICERCA-GRAFO in modo che, ogni volta che vengono trovati due cammini che portano allo stesso nodo, venga scartato sempre il più costoso dei due (v. la discussione nel Paragrafo 3.5). Per far ciò sono richiesti calcoli aggiuntivi complicati, che tuttavia garantiscono l'ottimalità della strategia. La seconda soluzione è assicurarsi che il primo cammino che arriva in uno stato ripetuto sia sempre quello ottimo, come nel caso della ricerca a costo uniforme. Questa proprietà vale se imponiamo un requisito supplementare su $h(n)$, che prende il nome di **consistenza** (chiamata anche monotonicità). Un'euristica $h(n)$ è consistente se, per ogni nodo n e ogni successore n' di n generato da un'azione a , il costo stimato per raggiungere l'obiettivo partendo da n non è superiore al costo di passo per arrivare a n' sommato al costo stimato per andare da lì all'obiettivo:

$$h(n) \leq c(n, a, n') + h(n').$$



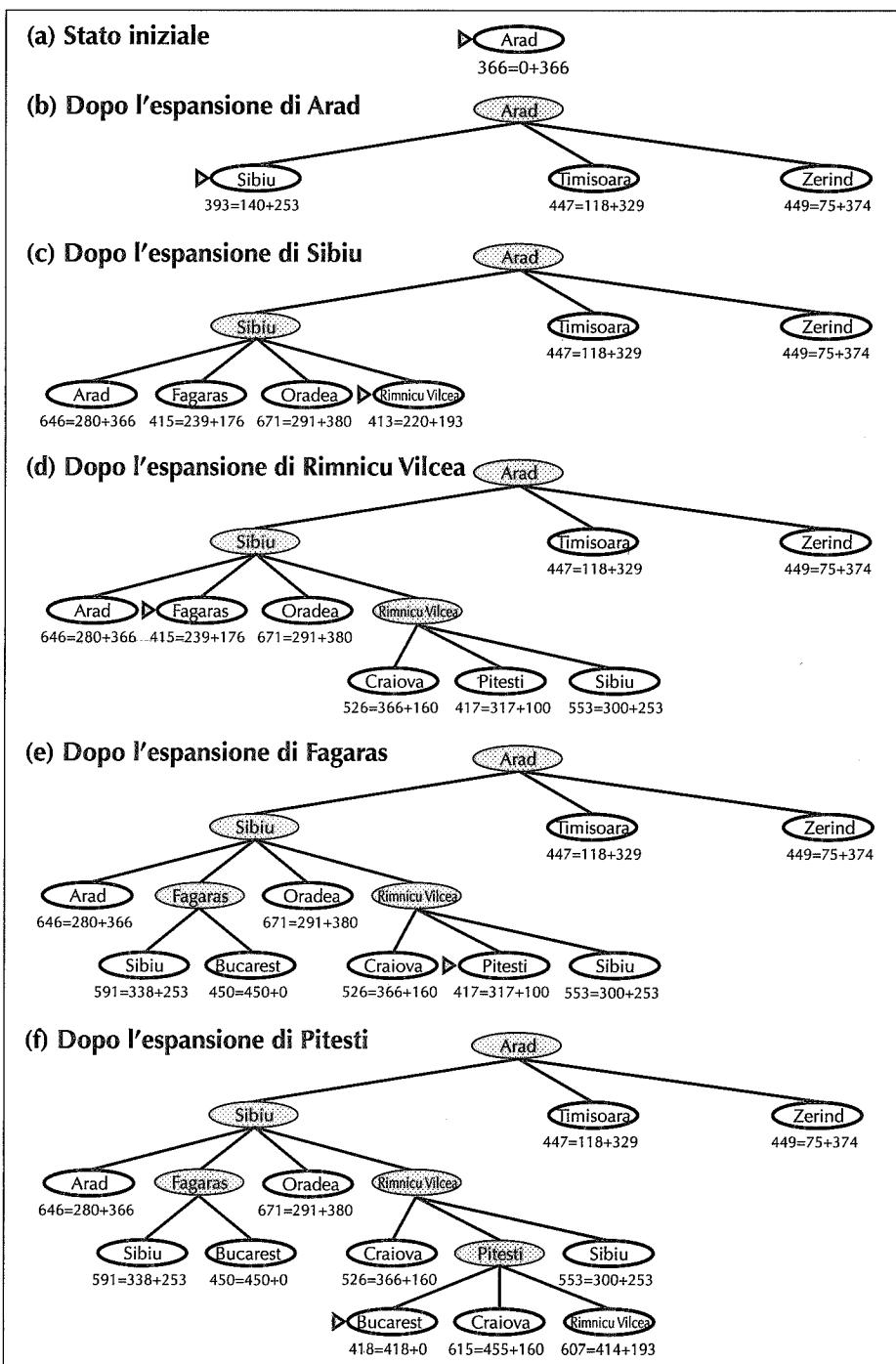


Figura 4.3
Fasi di una ricerca A* verso Bucarest.
I nodi sono etichettati con i valori $f = g + h$.
I valori h sono distanze in linea d'aria verso Bucarest, prese dalla Figura 4.1.

disuguaglianza triangolare



Questa è una forma della **disuguaglianza triangolare**, che afferma che ogni lato di un triangolo non può mai essere più lungo della somma degli altri due. In questo caso il triangolo è formato da n , n' e l'obiettivo più vicino a n . È abbastanza facile mostrare (v. Esercizio 4.7) che ogni euristica consistente è anche ammisible. La conseguenza più importante della consistenza è la seguente: *la strategia A* che usa l'algoritmo RICERCA-GRAFO è ottima se $h(n)$ è consistente*.

Benché la consistenza sia un requisito più stringente dell'ammisibilità, è molto difficile escogitare euristiche che siano ammissibili ma non consistenti. Tutte le euristiche ammissibili che abbiamo discusso in questo capitolo sono anche consistenti. Considerate ad esempio h_{DLA} : sappiamo che la disuguaglianza triangolare è soddisfatta quando ogni lato è misurato con la distanza in linea d'aria, e la distanza tra n e n' non è mai superiore a $c(n, a, n')$. ne consegue che h_{DLA} è un'euristica consistente.

Un'altra conseguenza importante della consistenza è questa: *se $h(n)$ è consistente, i valori di $f(n)$ lungo ogni cammino sono non decrescenti*. La dimostrazione è una diretta conseguenza della definizione: supponiamo che n' sia un successore di n ; allora $g(n') = g(n) + c(n, a, n')$ per qualche a , e risulta

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n).$$

Ne consegue che la sequenza di nodi espansi da A* usando RICERCA-GRAFO è in ordine non decrescente di $f(n)$; il primo nodo obiettivo selezionato per l'espansione dev'essere quindi una soluzione ottima, dato che tutti quelli successivi avranno un costo almeno uguale.

confini

Il fatto che gli f -costi siano non decrescenti su tutti i cammini significa anche che possiamo disegnare **confini** nello spazio degli stati, proprio come quelli su una mappa topografica (il termine inglese è *contour*, talvolta tradotto *frontiera*; abbiamo deciso di riservare questo termine per la frontiera dei nodi sull'albero di ricerca). La Figura 4.4 mostra un esempio: all'interno del confine etichettato 400, i nodi hanno tutti $f(n)$ minore o uguale a 400 e così via. Ne consegue che, dato che A* espande sempre il nodo di f -costo minore, la ricerca A* si allargherà a ventaglio dal nodo iniziale, aggiungendo nodi in bande concentriche di f -costo crescente.

Con la ricerca a costo uniforme (una ricerca A* che usa $h(n) = 0$), le bande avranno sempre una forma circolare centrata intorno allo stato iniziale. Avvalendosi di euristiche più accurate, le bande avranno una forma allungata verso lo stato obiettivo e si stringeranno intorno al cammino ottimo. Se C^* è il costo di cammino della soluzione ottima, possiamo dire quanto segue:

- ◆ A* espande tutti i nodi con $f(n) < C^*$.
- ◆ A* potrebbe espandere alcuni dei nodi che stanno proprio sul "confine dell'obiettivo" (dove $f(n) = C^*$) prima di scegliere un nodo obiettivo.

Intuitivamente è chiaro che la prima soluzione trovata dev'essere quella ottima, dato che i nodi obiettivo in tutti i confini successivi avranno un f -costo maggiore, e quindi anche un g -costo più alto (perché i nodi obiettivo hanno tutti $h(n) = 0$).

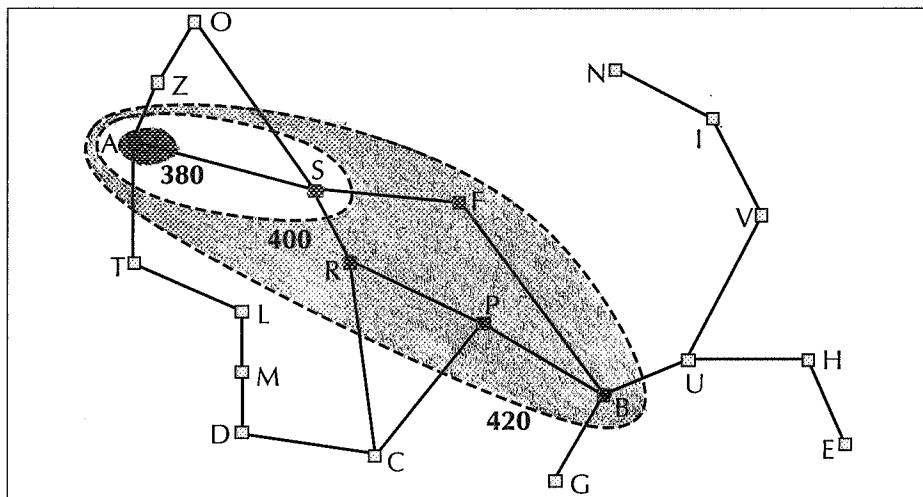


Figura 4.4 Una mappa della Romania che indica i confini per $f = 380$, $f = 400$ e $f = 420$ con Arad come stato iniziale. I nodi all'interno di un dato confine hanno tutti un f -costo minore o uguale al valore del confine.

È altrettanto ovvio che la ricerca A^* è completa: aggiungendo bande di f crescente, dovremo prima o poi raggiungere la banda in cui f è uguale al costo di cammino di una soluzione.⁴

Notate che A^* non espande alcun nodo con $f(n) > C^*$: ad esempio, nella Figura 4.3 Timisoara non è espansa, benché il nodo sia direttamente figlio della radice dell'albero. Si dice che il sottoalbero che ha come radice Timisoara è stato **potato** (*pruned*); poiché h_{DLA} è ammissibile, l'algoritmo può ignorare questo sottoalbero e continuare a garantire l'ottimalità. Il concetto di potatura, che permette di eliminare alcune possibilità senza bisogno di considerarle, è importante in molte aree dell'IA.

Come osservazione finale, notate che tra gli algoritmi ottimi del suo genere (quelli che costruiscono cammini di ricerca partendo dalla radice) A^* è **ottimamente efficiente** per qualsiasi funzione euristica. Questo significa che non c'è un altro algoritmo ottimo che garantisca di espandere meno nodi di A^* (eccetto forse per quanto riguarda i nodi che hanno $f(n) = C^*$). Questo perché qualsiasi algoritmo che *non* espande tutti i nodi con $f(n) < C^*$ corre il rischio di trascurare la soluzione ottima. È abbastanza soddisfacente che la ricerca A^* , tra tutti quegli algoritmi, sia completa, ottima e ottimamente efficiente. Sfortunatamente questo

potatura

ottimamente efficiente

⁴ La completezza richiede che ci sia un numero finito di nodi di costo minore o uguale a C^* , una condizione vera se tutti i costi di passo superano un ε finito e b è finito.

non significa che A* rappresenti la risposta a tutti i nostri problemi di ricerca. L'intoppo sta nel fatto che, nella maggior parte dei problemi, il numero di nodi all'interno del confine dello spazio di ricerca dell'obiettivo cresce ancora esponenzialmente con la lunghezza della soluzione. Benché la dimostrazione di questo risultato vada oltre gli scopi di questo libro, è stato provato che la crescita esponenziale si verificherà sempre, a meno che l'errore nella funzione euristica cresca al più con il logaritmo del costo effettivo del cammino. In notazione matematica, la condizione per una crescita meno che esponenziale è

$$| h(n) - h^*(n) | \leq O(\log h^*(n))$$

dove $h^*(n)$ è il costo *reale* del cammino da n all'obiettivo. Per quasi tutte le euristiche di uso pratico, l'errore è proporzionale almeno al costo di cammino, e la crescita esponenziale che ne deriva è in grado di mettere in ginocchio rapidamente qualsiasi computer. Per questa ragione spesso non conviene insistere sulla ricerca di una soluzione ottima. Si possono usare varianti di A* che trovano rapidamente soluzioni subottime, e talvolta si possono progettare euristiche più accurate, anche se non strettamente ammissibili. In ogni caso, l'uso di una buona euristica rappresenta ancora un vantaggio enorme rispetto alla ricerca non informata. Nel Paragrafo 4.2 considereremo il problema della progettazione di buone euristiche.

Il tempo di computazione, tuttavia, non è il difetto principale della ricerca A*. Poiché (come tutti gli algoritmi RICERCA-GRAFO) tiene in memoria tutti i nodi generati, A* di solito finisce lo spazio molto prima del tempo. Per questa ragione, in pratica, non è applicabile a problemi di larga scala. Algoritmi sviluppati di recente hanno superato il problema dello spazio senza sacrificare l'ottimalità né la completezza, con un piccolo costo nel tempo di esecuzione: li discuteremo ora.

Ricerca euristica con memoria limitata

Il modo più semplice di ridurre i requisiti di memoria di A* è adattare l'idea dell'approfondimento iterativo al contesto della ricerca euristica, dando così origine all'algoritmo IDA* (dall'inglese *Iterative-Deepening A**). La differenza principale tra IDA* e l'approfondimento iterativo standard sta nel valore di taglio, che non è più basato sulla profondità ma sull' f -costo ($g + h$); a ogni iterazione il nuovo valore di taglio è l' f -costo minimo tra quelli di tutti i nodi che hanno superato il valore di taglio nell'iterazione precedente. Nella pratica IDA* è applicabile a molti problemi che hanno costi di passo unitari e non deve svolgere i pesanti calcoli necessari per mantenere una coda ordinata di nodi. Sfortunatamente soffre delle stesse difficoltà che abbiamo descritto nell'Esercizio 3.11 per la versione iterativa della ricerca a costo uniforme, quando i costi sono espressi da valori reali. Questo paragrafo presenta brevemente due algoritmi più moderni a memoria limitata, chiamati RBFS e MA*.

La ricerca best-first ricorsiva (RBFS, da *Recursive Best-First Search*) è un semplice algoritmo ricorsivo che cerca di imitare il funzionamento di una ricerca best-first standard usando solamente uno spazio lineare. L'algoritmo è illustrato nella

```

function RICERCA-BEST-FIRST-RICORSIVA(problema) returns una soluzione, o il fallimento
  RBFS(problema, CREA-NODO(STATO-INIZIALE[problema]),  $\infty$ )

function RBFS(problema, nodo, f-limite) returns una soluzione, o il fallimento
  e un nuovo limite all'f-costo
  if TEST-OBIETTIVO[problema](STATO[nodo]) then return nodo
  successori  $\leftarrow$  ESPANDI(nodo, problema)
  if successori è vuoto then return fallimento,  $\infty$ 
  for each s in successori do
    f[s]  $\leftarrow$  max(g(s) + h(s), f[nodo])
  repeat
    best  $\leftarrow$  il nodo con f-valore minimo in successori
    if f[best] > f-limite then return fallimento, f[best]
    alternativa  $\leftarrow$  il secondo f-valore più basso in tutti i successori
    risultato, f[best]  $\leftarrow$  RBFS(problema, best, min(f-limite, alternativa))
    if risultato  $\neq$  fallimento then return risultato

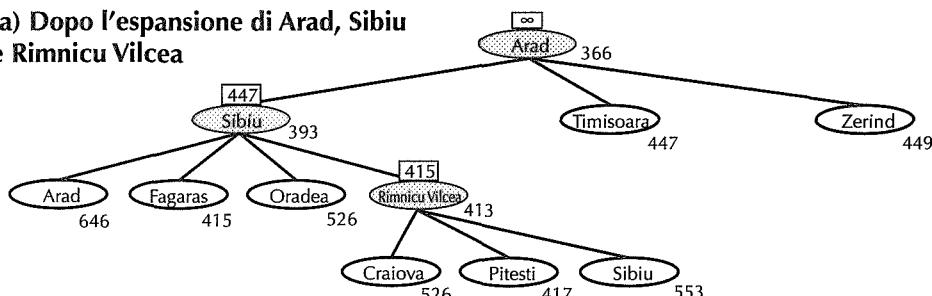
```

Figura 4.5 L'algoritmo per la ricerca best-first ricorsiva.

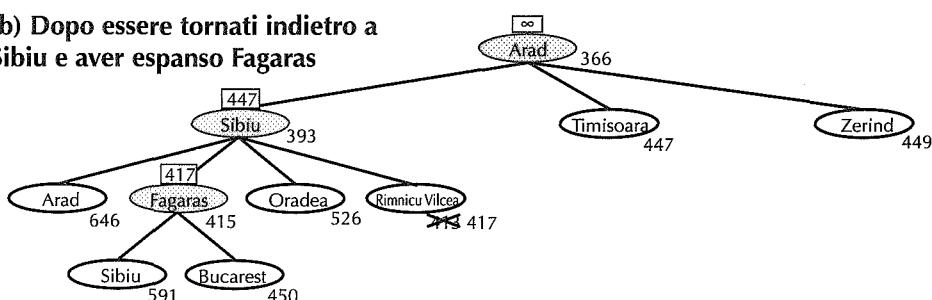
Figura 4.5: la sua struttura è simile a quella della ricerca ricorsiva in profondità, ma invece di continuare a seguire indefinitamente il cammino corrente, si tiene traccia dell'*f*-valore del miglior cammino alternativo che parte da uno qualsiasi degli antenati del nodo corrente. Se il nodo corrente supera questo limite, la ricorsione torna indietro al cammino alternativo. Durante il ritorno, RBFS sostituisce l'*f*-valore di ogni nodo lungo il cammino con il miglior *f*-valore dei suoi nodi figli. In questo modo RBFS ricorda l'*f*-valore della foglia migliore nel sottoalbero abbandonato e può quindi decidere in seguito di ri-espanderlo. La Figura 4.6 mostra come RBFS raggiunge Bucarest.

RBFS è più efficiente di IDA*, ma soffre ancora per un'eccessiva ri-generazione di nodi. Nell'esempio della Figura 4.6 RBFS segue prima il cammino che passa per Rimnicu Vilcea, poi "ci ripensa" e prova Fagaras, poi cambia ancora idea. Questo si verifica perché, ogni volta che viene esteso il cammino migliore, c'è una buona probabilità che il suo *f*-valore aumenti: infatti di solito *h* è meno ottimista quando i nodi sono più vicini all'obiettivo. Quando ciò accade, e in particolar modo quando gli spazi di ricerca sono grandi, il secondo cammino migliore può diventare quello migliore in assoluto, e così la ricerca deve tornare indietro per seguirlo. Ogni "ripensamento" corrisponde a un'iterazione di IDA* e può richiedere molte ri-expansioni di nodi dimenticati per ricreare il cammino migliore ed estenderlo di un nodo.

(a) Dopo l'espansione di Arad, Sibiu e Rimnicu Vilcea



(b) Dopo essere tornati indietro a Sibiu e aver espanso Fagaras



(c) Dopo essere ritornati ancora a Rimnicu Vilcea e aver espanso Pitesti

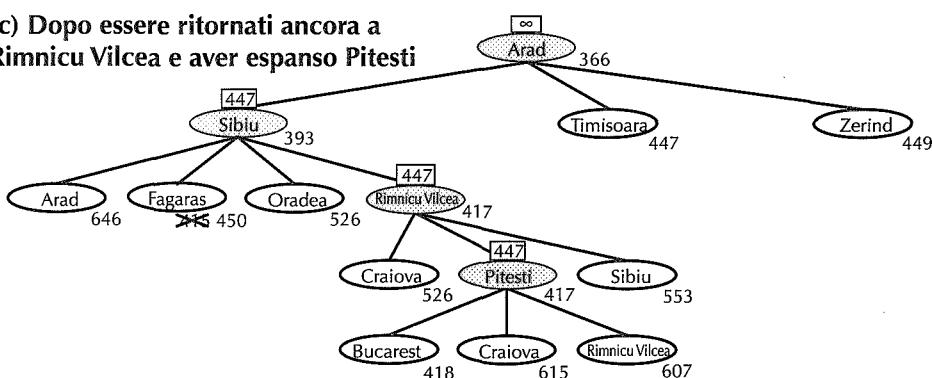


Figura 4.6 Fasi di una ricerca RBFS per trovare la strada più breve verso Bucarest. Il limite agli f -valori di ogni chiamata ricorsiva è indicato in un box sopra al nodo corrente. (a) Si segue il cammino che passa da Rimnicu Vilcea finché la foglia migliore corrente (Pitesti) non ha un valore peggiore del miglior cammino alternativo (Fagaras). (b) La ricorsione "torna su" e il valore della foglia migliore del sottoalbero dimenticato (417) è salvato in Rimnicu Vilcea; quindi viene espanso Fagaras, rivelando che il valore della foglia migliore è 450. (c) La ricorsione torna ancora indietro e il valore della foglia migliore del sottoalbero dimenticato (450) è salvato in Fagaras; quindi si espanderà Rimnicu Vilcea. Stavolta, dato che il miglior cammino alternativo (quello che passa per Timisoara) costa almeno 447, l'espansione continua fino a raggiungere Bucarest.

Come A*, RBFS è un algoritmo ottimo se la funzione euristica $h(n)$ è ammissibile. La sua complessità spaziale è $O(bd)$, ma quella temporale è abbastanza difficile da definire, perché dipende sia dall'accuratezza della funzione euristica sia dalla frequenza dei cambiamenti del cammino ottimo durante l'espansione dei nodi. Sia IDA* che RBFS sono soggetti all'incremento di complessità, potenzialmente esponenziale, associato alla ricerca su grafo (v. Paragrafo 3.5), perché non possono verificare l'esistenza di stati ripetuti se non sul cammino corrente. Perciò, potrebbero esplorare più volte lo stesso stato.

Il problema di IDA* e RBFS è che usano *troppa poca* memoria. Tra un'iterazione e l'altra, IDA* ricorda solo un numero: il limite corrente all' f -costo. RBFS mantiene in memoria più informazione, ma ne usa solo $O(bd)$: anche se ne fosse disponibile molta di più, non avrebbe alcun modo di usarla.

In effetti, sfruttare esattamente tutta la memoria disponibile sembra la cosa più sensata. Due algoritmi che lo fanno sono MA* (*memory-bounded A**) e SMA* (*simplified MA**). Tra i due descriveremo SMA*, che come dice il nome è una versione semplificata del primo. SMA* procede proprio come A*, espandendo la foglia migliore finché la memoria è piena. A questo punto non può aggiungere un nuovo nodo all'albero di ricerca senza cancellarne uno vecchio. SMA* scarta sempre il nodo foglia *peggiore*, quello con f -valore più alto. Come RBFS, memorizza nel nodo padre il valore del nodo dimenticato. In questo modo la radice di un sottoalbero dimenticato conosce la qualità del cammino migliore in quel sottoalbero. Con quest'informazione SMA* ri-genera il sottoalbero dimenticato solo quando *tutti gli altri cammini* promettono di comportarsi peggio di quello. Potremmo anche dire che, se tutti i discendenti di un nodo n sono dimenticati, allora non sappiamo da che parte andare partendo da n , ma abbiamo ancora un'idea chiara di quanto sia conveniente passare per n .

MA*
SMA*

L'algoritmo completo è troppo complicato per riprodurlo qui,⁵ ma vale la pena di menzionare una sottigliezza. Abbiamo detto che SMA* espande la foglia migliore e cancella quella peggiore. Ma cosa succede se *tutte* le foglie hanno lo stesso f -valore? In questo caso l'algoritmo potrebbe scegliere lo stesso nodo sia per la cancellazione che per l'espansione. SMA* risolve questo problema espandendo la foglia migliore *più recente* e cancellando la foglia peggiore *più vecchia*. Questi due nodi possono coincidere solamente se c'è una sola foglia; in tal caso l'albero di ricerca deve consistere in un singolo cammino, che riempie tutta la memoria, dalla radice alla foglia in questione. Se la foglia non è un nodo obiettivo, allora *anche se si trovasse sul cammino di una soluzione ottima* tale soluzione non sarebbe raggiungibile con la memoria disponibile. Di conseguenza quel nodo può essere scartato esattamente come se non avesse successori.

⁵ Uno schema semplificato dell'algoritmo compariva nella prima edizione di questo libro.

SMA* è completa se c'è una soluzione raggiungibile, ovvero se d , la profondità del nodo obiettivo più vicino alla radice, è inferiore alla dimensione della memoria espressa in nodi. La strategia è ottima se c'è una soluzione ottima raggiungibile; altrimenti viene restituita la soluzione migliore tra quelle raggiungibili. In pratica, SMA* è probabilmente il miglior algoritmo di uso generale per trovare soluzioni ottime, in particolare quando lo spazio degli stati è un grafo, i costi dei passi non sono uniformi e la generazione dei nodi è costosa rispetto all'elaborazione aggiuntiva richiesta per il mantenimento della lista aperta e di quella chiusa.

Nel caso di problemi molto difficili, comunque, una ricerca SMA* sarà spesso obbligata a passare continuamente dall'uno all'altro tra molti cammini candidati, di cui sarà possibile tenere in memoria solo un piccolo sottoinsieme (questo problema ricorda quello delle pagine di memoria nei sistemi operativi). A questo punto il tempo aggiuntivo richiesto per la generazione ripetuta degli stessi nodi potrebbe far sì che problemi risolvibili in pratica da A*, a patto di avere memoria illimitata, diventino intrattabili per SMA*. Questo significa che *le limitazioni di memoria possono rendere un problema intrattabile sotto l'aspetto temporale*. Sebbene non ci sia una teoria che formalizzi il problema del compromesso tra tempo e memoria, non sembra che esista una soluzione efficace: l'unica possibilità è rinunciare al requisito dell'ottimalità.

Imparare a cercare meglio

Abbiamo presentato diverse strategie prefissate progettate dagli informatici: ricerca in ampiezza, best-first greedy, e così via. Ma è possibile che un agente possa *imparare* a cercare meglio? La risposta è sì, e il metodo è basato su un concetto importante chiamato **spazio degli stati di metalivello**. Ogni stato in questo spazio rappresenta lo stato interno (computazionale) di un programma che sta eseguendo una ricerca in uno **spazio degli stati a livello degli oggetti**, come la Romania. Ad esempio, lo stato interno dell'algoritmo A* consiste nell'albero di ricerca corrente. Ogni azione nello spazio degli stati di metalivello è un passo computazionale che altera lo stato interno; per esempio, ogni passo in A* espande un nodo foglia e aggiunge all'albero i suoi successori. Quindi la Figura 4.3, che mostra una sequenza di alberi della ricerca sempre più grandi, può essere vista come un cammino nello spazio degli stati di metalivello in cui ogni stato, al livello degli oggetti, corrisponde a un albero di ricerca.

Il cammino nella Figura 4.3 ha cinque passi tra cui uno, l'espansione di Fagaras, non molto utile. Nei problemi più difficili si verificheranno molti analoghi passi falsi, e un algoritmo di **apprendimento di metalivello** può imparare da tali esperienze ed evitare di esplorare sottoalberi poco promettenti (le tecniche usate per questo tipo di apprendimento sono descritte nel Capitolo 21, nel 2º vol.). Lo scopo dell'apprendimento è minimizzare il **costo totale** della soluzione del problema, trovando il giusto equilibrio tra costo computazionale e costo del cammino.



spazio degli stati di metalivello

spazio degli stati a livello degli oggetti

apprendimento di metalivello

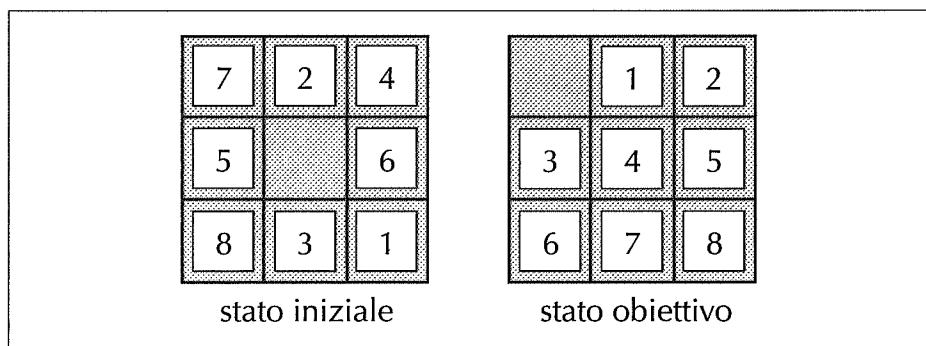


Figura 4.7
Un'istanza tipica
del rompicapo a 8
tasselli.
La soluzione è
lunga 26 passi.

4.2 Funzioni euristiche

In questo paragrafo esamineremo le euristiche per il rompicapo a 8 tasselli, con lo scopo di gettare luce sulla natura delle euristiche in generale.

Questo puzzle è stato uno dei primi problemi di ricerca euristica. Come abbiamo detto nel Paragrafo 3.2, lo scopo è di far scivolare i tasselli orizzontalmente o verticalmente nello spazio vuoto finché la loro configurazione non corrisponde all’obiettivo (Figura 4.7).

Il costo medio della soluzione per un'istanza generata casualmente è di circa 22 passi. Il fattore di ramificazione è di circa 3: infatti quando lo spazio vuoto è nel centro ci sono quattro possibili mosse; quando è in un angolo ce ne sono due; quando si trova su un lato ce ne sono tre. Questo significa che una ricerca esaustiva a una profondità 22 dovrebbe esaminare circa $3^{22} \approx 3,1 \times 10^{10}$ stati. Tenendo traccia degli stati ripetuti potremmo ridurre questo numero di un fattore di circa 170.000, perché ci sono solamente $9!/2 = 181,440$ stati distinti raggiungibili (v. Esercizio 3.4). Questo è gestibile, ma il numero corrispondente per il rompicapo a 15 tasselli è di circa 10^{13} , per cui ci conviene trovare una buona funzione euristica. Se vogliamo trovare le soluzioni più rapide usando A*, ci serve una funzione euristica che non sopravvaluti mai il numero di passi che ci separano dall'obiettivo. C'è un lunga storia di euristiche simili per il rompicapo a 15 tasselli; ecco due delle candidate più comuni:

- ◆ b_1 = il numero di tasselli fuori posto. Nella Figura 4.7 gli otto tasselli sono tutti fuori posto, per cui lo stato iniziale avrebbe $b_1 = 8$. b_1 è un'euristica ammissibile, perché è palese che ogni tassello fuori posto dovrà essere mosso almeno una volta;
 - ◆ b_2 = la somma delle distanze di tutti i tasselli dalla loro posizione corrente a quella nella configurazione obiettivo. Dato che i tasselli non si possono muovere in diagonale, considereremo la somma delle distanze in orizzontale e ver-

distanza Manhattan

ticale, che viene chiamata **distanza tra isolati** o **distanza Manhattan**. Anche h_2 è ammmissible, perché una mossa può al massimo spostare un tassello un passo più vicino al suo obiettivo. Considerando i tasselli nell'ordine da 1 a 8, nello stato iniziale abbiamo una distanza Manhattan di

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

Come speravamo, nessuna di queste due sovrastima il vero costo della soluzione, che è 26.

fattore effettivo di ramificazione

Effetto dell'accuratezza dell'euristica sulle prestazioni

Un modo di caratterizzare la qualità di un'euristica è il **fattore effettivo di ramificazione**, indicato con b^* . Se il numero totale di nodi generati da A^* per un particolare problema è N e la profondità è d , allora b^* è il fattore di ramificazione che un albero uniforme di profondità d dovrebbe avere per contenere $N + 1$ nodi. Quindi,

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d.$$

Ad esempio, se A^* trova una soluzione a profondità 5 usando 52 nodi, il fattore effettivo di ramificazione è 1,92. Questo fattore può cambiare da un'istanza del problema all'altra, ma solitamente ha un valore abbastanza costante quando il problema è sufficientemente difficile. Di conseguenza, basta misurare b^* con un numero abbastanza piccolo di esperimenti per avere una buona idea dell'utilità generale dell'euristica. Un'euristica ben progettata dovrebbe avere un valore di b^* vicino a 1, permettendo così la soluzione di problemi abbastanza grandi.

Per il test delle funzioni euristiche h_1 e h_2 abbiamo generato casualmente 1200 problemi, la lunghezza delle cui soluzioni va da 2 a 24 (ce ne sono 100 per ogni numero pari) e li abbiamo risolti con una ricerca ad approfondimento iterativo nonché con una ricerca A^* usando sia h_1 che h_2 . La Figura 4.8 fornisce il numero medio di nodi espanso da ogni strategia e il fattore effettivo di ramificazione. I risultati suggeriscono che h_2 è meglio di h_1 , e molto meglio della ricerca ad approfondimento iterativo. Nelle soluzioni di lunghezza 14, A^* con h_2 è 30.000 volte più efficiente della ricerca non informata.

domina

Ci si potrebbe chiedere se h_2 è *sempre* meglio di h_1 : la risposta è sì. Dalle definizioni delle due euristiche è facile vedere che, per ogni nodo n , $h_2(n) \geq h_1(n)$. Diciamo quindi che h_2 **domina** h_1 . La dominazione si traduce direttamente in efficienza: la strategia A^* che usa h_2 non espanderà mai più nodi di quella che usa h_1 (eccetto forse alcuni per cui $f(n) = C^*$). La dimostrazione è semplice: ricordate l'osservazione a pag. 132, che afferma che ogni nodo con $f(n) < C^*$ sarà sicuramente espanso. Questo è equivalente a dire che sarà espanso ogni nodo con $h(n) < C^* - g(n)$. Ma, dato che h_2 è grande almeno quanto h_1 per tutti i nodi, ogni nodo espanso dalla ricerca A^* che usa h_2 lo sarà anche con h_1 , mentre h_1 po-

d	costo di ricerca			fattore di ramificazione effettivo		
	RAI	$A^*(h_1)$	$A^*(h_2)$	RAI	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	3644035	227	173	2,78	1,42	1,24
14	-	539	113	-	1,44	1,23
16	-	1301	211	-	1,45	1,25
18	-	3056	363	-	1,46	1,26
20	-	7276	676	-	1,47	1,27
22	-	18094	1219	-	1,48	1,28
24	-	39135	1641	-	1,48	1,26

Figura 4.8 Confronto dei costi di ricerca e dei fattori di ramificazione effettivi per gli algoritmi RICERCA-APPROFONDIMENTO-ITERATIVO e A^* con le euristiche h_1 e h_2 . I dati rappresentano una media su 100 istanze del rompicapo a 8 tasselli, con soluzioni di lunghezza variabile.

trà causare anche l'espansione di altri nodi. Se ne deduce che è sempre meglio usare una funzione euristica con valori più alti, a patto che la sua stima non sia sbagliata per eccesso e che il tempo per calcolarla non sia troppo grande.

Inventare funzioni euristiche ammissibili

Abbiamo visto che sia h_1 (tasselli fuori posto) che h_2 (distanza Manhattan) sono euristiche abbastanza buone per il rompicapo a 8 tasselli, e che tra le due è preferibile h_2 . Ma com'è possibile inventare un'euristica come h_2 ? Un computer può farlo automaticamente?

h_1 e h_2 sono stime della lunghezza rimanente del cammino che conduce alla soluzione, ma sono anche lunghezze di cammino perfettamente accurate per versioni *semplificate* del rompicapo. Se le regole del problema fossero cambiate in modo che un tassello potesse muoversi ovunque, invece che solo nello spazio vuoto adiacente, h_1 rappresenterebbe esattamente il numero di passi della soluzione più breve. In modo analogo, se un tassello potesse muoversi di uno spazio in ogni direzione, anche se la posizione fosse già occupata da un altro tassello, h_2 indicherebbe l'esatto numero di passi della soluzione ottima. Un problema con meno restrizioni sulle azioni possibili è chiamato **problema rilassato**. Il costo di una soluzione ottima di un problema rilassato è un'euroistica ammissibile per il problema originale. L'euristica è ammissibile perché la soluzione ottima del problema originale è per definizione anche una soluzione del problema rilassato, quindi deve avere un costo almeno pari alla soluzione ottima di quest'ultimo. Dato che l'euristica derivata è

problema rilassato



un costo esatto del problema rilassato, deve rispettare la disuguaglianza triangolare ed è quindi consistente (v. pag. 130).

Se la definizione di un problema è scritta in un linguaggio formale, è possibile costruire automaticamente i suoi rilassamenti.⁶ Per esempio, se le azioni del rompicapo sono descritte come segue:

un tassello si può muovere dalla posizione A a quella B se

A è adiacente orizzontalmente o verticalmente a B and B è vuota

possiamo generare tre problemi rilassati rimuovendo una o entrambe le condizioni:

- (a) un tassello può muoversi da A a B se A e B sono adiacenti
- (b) un tassello può muoversi da A a B se B è vuota
- (c) un tassello può muoversi da A a B.

Da (a) possiamo derivare h_2 (distanza Manhattan). L'idea è che h_2 sarebbe la distanza esatta dalla soluzione se potessimo semplicemente muovere i tasselli uno dopo l'altro fino alla loro destinazione. L'euristica derivata da (b) è discussa nell'Esercizio 4.9. Da (c) possiamo derivare h_1 (tasselli fuori posto), perché tale euristica rappresenterebbe la distanza esatta se i tasselli potessero essere spostati nella loro destinazione in un solo passo. Un aspetto cruciale è che i problemi rilassati generati con questa tecnica possono essere risolti praticamente *senza ricerca*, perché il rilassamento delle regole permette di scomporre il problema in otto sottoproblemi indipendenti. Se il problema rilassato è ancora difficile da risolvere i valori della corrispondente euristica saranno costosi da ottenere.⁷

Esiste un programma chiamato ABSOLVER che può generare automaticamente euristiche partendo dalla definizione dei problemi, usando il metodo dei problemi rilassati e varie altre tecniche (Prieditis, 1993). ABSOLVER ha generato una nuova euristica per il rompicapo a 8 tasselli migliore di tutte quelle preesistenti e ha trovato la prima euristica utile per il famoso cubo di Rubik.

Una difficoltà della generazione di nuove funzioni euristiche è che spesso non si riesce a capire quale sia quella "chiaramente" preferibile alle altre. Se per un problema abbiamo una collezione di euristiche ammissibili h_1, \dots, h_m e nessuna domina le altre, quale dovremmo scegliere? In effetti, non siamo obbligati a fare una scelta. Possiamo prendere il meglio di tutte definendo

$$h(n) = \max \{h_1(n), \dots, h_m(n)\}.$$

⁶ Nei Capitoli 8 e 11 descriveremo linguaggi formali adatti a questo compito; se le descrizioni formali possono essere manipolate, la costruzione di problemi rilassati può essere automatizzata. Per adesso useremo il linguaggio naturale.

⁷ Notate che si può sempre ottenere un'euristica perfetta semplicemente permettendo a h di eseguire "furtivamente" una completa ricerca in ampiezza. Per le funzioni euristiche, quindi, c'è sempre un compromesso tra accuratezza e tempo di computazione.

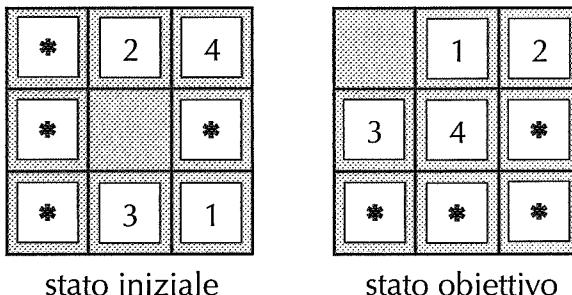


Figura 4.9
Un sottoproblema dell'istanza del rompicapo a 8 tasselli della Figura 4.7. Lo scopo è mettere i tasselli 1, 2, 3 e 4 nella posizione corretta, senza preoccuparsi degli altri.

Quest'euristica composita usa la funzione più accurata per ogni nodo. Dato che tutte le euristiche componenti sono ammissibili, h è ammissibile; è anche facile dimostrare che è consistente. In definitiva, h domina tutte le sue euristiche componenti.

È anche possibile derivare euristiche ammissibili dal costo della soluzione di un sottoproblema del problema dato. Per esempio, la Figura 4.9 mostra un sottoproblema dell'istanza del rompicapo della Figura 4.7, che considera solo i tasselli 1, 2, 3 e 4. Palesemente, il costo della soluzione ottima di questo sottoproblema è un limite inferiore del costo del problema completo. Quest'euristica, in alcuni casi, si è rivelata molto più accurata della distanza Manhattan.

L'idea alla base dei database di pattern è di memorizzare i costi esatti delle soluzioni di ogni possibile istanza di sottoproblema: nel nostro esempio, ogni possibile configurazione dei primi quattro tasselli e dello spazio vuoto (notate che le posizioni degli altri quattro tasselli sono irrilevanti per risolvere il sottoproblema, ma le loro mosse continuano a contare ai fini del calcolo del costo). Fatto questo possiamo ottenere un'euristica ammissibile h_{DB} per ogni stato completo incontrato durante una ricerca, semplicemente estraendo dal database la corrispondente configurazione del sottoproblema. Il database stesso è costruito eseguendo una ricerca all'indietro dallo stato obiettivo e memorizzando il costo di ogni nuova configurazione; la spesa di questa ricerca è ammortizzata dal fatto che il database può essere usato per molte istanze del problema.

La scelta di usare i primi quattro tasselli è arbitraria; avremmo potuto costruire database per i tasselli 5-6-7-8, oppure 2-4-6-8, e così via. Ognuno di essi fornisce un'euristica ammissibile, e queste possono essere tutte combinate, come abbiamo visto poco fa, prendendo sempre il loro valore massimo. Un'euristica combinata di questo tipo è molto più accurata della distanza Manhattan; il numero di nodi generati per risolvere un rompicapo a 15 tasselli può essere ridotto anche di un fattore 1000.

Ci si potrebbe chiedere se è possibile sommare l'euristica ottenuta dal database 1-2-3-4 e quella del database 5-6-7-8, dato che i due sottoproblemi non sembrano sovrapporsi. L'euristica risultante sarebbe ancora ammissibile? La risposta è

sottoproblema

database di pattern

database di pattern
disgiunti

no, perché le soluzioni del sottoproblema 1-2-3-4 e quelle del sottoproblema 5-6-7-8 in un dato stato condivideranno certamente alcune mosse: in altre parole, è molto improbabile che i tasselli 1-2-3-4 possano essere spostati al loro posto senza toccare i tasselli 5-6-7-8 e viceversa. E se non contassimo quelle mosse? Possiamo considerare non già il costo totale della soluzione del sottoproblema 1-2-3-4, ma solo il numero di mosse che coinvolgono i primi quattro tasselli. In questo caso è facile verificare che la somma dei due costi è ancora un limite inferiore del costo della soluzione dell'intero problema. Questa è l'idea alla base dei database di pattern disgiunti. Usando questi ultimi, è possibile risolvere istanze casuali del rompicapo a 15 tasselli in pochi millisecondi – il numero di nodi generati è ridotto di un fattore 10.000 rispetto alla strategia che utilizza la distanza Manhattan. Nel caso del rompicapo a 24 tasselli, il processo può essere velocizzato circa un milione di volte.

In questo caso i database a pattern disgiunti funzionano bene, perché dato che si sposta comunque un solo tassello per volta il problema può essere suddiviso in modo tale che ogni mossa influisce su un solo sottoproblema. Per un problema come il cubo di Rubik questo tipo di suddivisione non può essere fatto, perché ogni mossa coinvolge 8 o 9 dei 26 cubetti: a tutt'oggi, non è chiaro come si possano definire database disgiunti per questo tipo di problemi.

Apprendere euristiche dall'esperienza

Una funzione euristica $h(n)$ deve stimare il costo di una soluzione a partire dallo stato corrispondente a un nodo n . In che modo un agente potrebbe costruire una funzione siffatta? Una soluzione è inventare problemi rilassati per cui è facile trovare una soluzione ottima. Un'altra possibilità è apprendere dall'esperienza: in questo particolare caso significa risolvere un sacco di rompicapi a 8 tasselli. Ogni soluzione ottima per un'istanza del rompicapo rappresenta un esempio da cui si può estrapolare un'euristica $h(n)$. Ogni esempio consiste in uno stato sul cammino della soluzione unito al costo effettivo della soluzione a partire da quel punto. Un algoritmo di apprendimento induttivo può sfruttare questi esempi per costruire una funzione $h(n)$ capace, con un po' di fortuna, di predire i costi delle soluzioni per altri stati che dovessero verificarsi durante la ricerca. Tecniche per far questo usando reti neurali, alberi di decisioni e altri metodi sono presentate nel 2^o volume, nel Capitolo 18 (si può applicare anche l'apprendimento per rinforzo, descritto nel Capitolo 21).

I metodi di apprendimento induttivo funzionano meglio quando si forniscano loro, invece dalla semplice descrizione di uno stato, le caratteristiche più rilevanti per la sua valutazione. Ad esempio, la caratteristica "numero di tasselli fuori posto" potrebbe essere utile nella predizione dell'effettiva distanza di uno stato dall'obiettivo. Chiamiamo $x_1(n)$ questa caratteristica. Potremmo prendere 100 configurazioni generate casualmente di rompicapi a 8 tasselli e raccogliere statistiche sui costi effettivi delle loro soluzioni. Potremmo trovare che quando $x_1(n)$ vale 5, il costo medio della soluzione è intorno a 14, e così via. Partendo da questi dati, po-

caratteristiche

tremmo usare il valore di x_1 per predire $h(n)$. Naturalmente, possiamo sfruttare più di una caratteristica: una seconda caratteristica $x_2(n)$ potrebbe essere “il numero di coppie di tasselli adiacenti che sono adiacenti anche nello stato obiettivo”. Come dovrebbero essere combinate $x_1(n)$ e $x_2(n)$ nella predizione di $h(n)$? Un approccio comune è usare una combinazione lineare:

$$h(n) = c_1x_1(n) + c_2x_2(n).$$

Le costanti c_1 e c_2 sono calcolate per adattarsi al meglio ai dati effettivi relativi ai costi delle soluzioni. In questo caso, presumibilmente, c_1 dovrebbe essere positiva e c_2 negativa.

4.3 Algoritmi di ricerca locale e problemi di ottimizzazione

Gli algoritmi che abbiamo visto fin qui sono progettati per esplorare sistematicamente gli spazi di ricerca. Questa sistematicità si ottiene tenendo in memoria uno o più cammini e registrando quali alternative sono state esplorate in ogni punto del cammino e quali no. Quando viene raggiunto uno stato obiettivo, il cammino verso quello stato costituisce una soluzione del problema.

Per molti problemi, comunque, il cammino verso l'obiettivo è irrilevante: ad esempio, nel problema delle 8 regine (v. pag. 88), ciò che conta è la configurazione finale delle regine e non l'ordine con cui sono state aggiunte. Questa classe di problemi include molte applicazioni importanti come la progettazione di circuiti integrati, la configurazione degli spazi nelle fabbriche, il cosiddetto “job-shop scheduling”, la programmazione automatica, l'ottimizzazione di reti di telecomunicazioni, l'instradamento di veicoli e la gestione di portafogli di azioni.

Se il cammino verso l'obiettivo non ha importanza possiamo considerare una diversa classe di algoritmi, che dei cammini non si curano proprio. Gli algoritmi di ricerca locale operano su un singolo stato corrente invece che su cammini multipli e in generale si spostano solo negli stati immediatamente adiacenti. I cammini seguiti dalla ricerca, tipicamente, non vengono memorizzati. Benché gli algoritmi di ricerca locale non siano sistematici, hanno due vantaggi importanti: (1) usano molta poca memoria, solitamente una quantità costante; (2) possono spesso trovare soluzioni ragionevoli in spazi degli stati grandi o infiniti (continui) in cui gli algoritmi sistematici non sono applicabili.

Oltre a trovare soluzioni, gli algoritmi di ricerca locale sono utili per risolvere problemi di ottimizzazione puri, in cui lo scopo è trovare lo stato migliore secondo una funzione obiettivo (*objective function*). Molti problemi di ottimizzazione non si adattano al modello di ricerca “standard” presentato nel Capitolo 3. Ad esempio, la natura fornisce una funzione obiettivo – la capacità di riprodursi – che l'evoluzione delle specie sembra cercare di ottimizzare, ma non c'è “test obiettivo” o “costo di cammino” per questo problema.

ricerca locale
stato corrente

problem di
ottimizzazione
funzione obiettivo

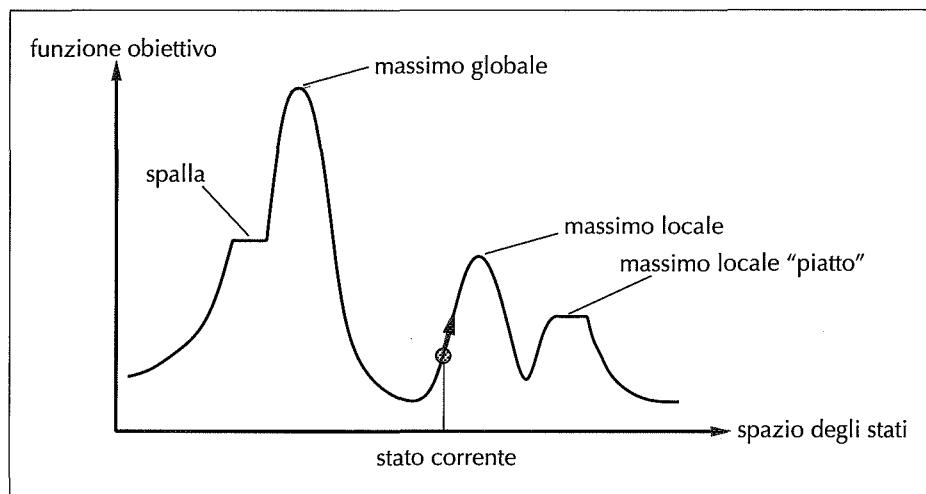


Figura 4.10 Un panorama dello spazio degli stati monodimensionale, in cui l'altezza corrisponde alla funzione obiettivo. Lo scopo è quindi trovare un massimo globale. La ricerca hill-climbing modifica lo stato corrente cercando di migliorarlo, come mostrato dalla freccia. Le varie caratteristiche topografiche sono definite nel testo.

panorama dello spazio
degli stati

minimo globale

massimo globale

hill-climbing

Per comprendere la ricerca locale ci sarà molto utile considerare il panorama dello spazio degli stati (*state space landscape*), come si vede nella Figura 4.10. Un panorama ha sia una “posizione” (definita dallo stato) sia una “altezza” (definita dal valore della funzione euristica di costo o funzione obiettivo). Se l'altezza corrisponde al costo, lo scopo è trovare l'avvallamento più basso, ovvero un minimo globale; se l'altezza corrisponde invece alla funzione obiettivo lo scopo è trovare il picco più alto, ovvero un massimo globale (naturalmente, per passare dall'uno all'altro basta inserire un segno meno). Gli algoritmi di ricerca locale esplorano questo panorama. Un algoritmo di ricerca locale completo trova sempre un obiettivo, se questo esiste; un algoritmo ottimo trova sempre un minimo/massimo globale.

Ricerca hill-climbing

L'algoritmo di ricerca **hill-climbing** è riportato nella Figura 4.11: il termine, che significa letteralmente “scalatore di colline”, deriva dal fatto che segue sempre le salite più ripide. Si tratta di un semplice ciclo che si muove continuamente verso l'alto, cioè nella direzione dei valori crescenti, e termina quando raggiunge un picco che non ha vicini di valore più alto. L'algoritmo non mantiene un albero di ricerca, per cui la struttura dati del nodo corrente deve solo memorizzare lo stato e il valore della sua funzione obiettivo. L'algoritmo hill-climbing non guarda al di là degli stati immediatamente vicini a quello corrente: è un po' come cercare la cima del monte Everest immersi in un nebbione, e soffrendo di amnesia.

```

function HILL-CLIMBING(problema) returns uno stato che è un massimo locale
  inputs: problema, un problema
  variabili locali: nodo_corrente, un nodo
    vicino, un nodo

  nodo_corrente  $\leftarrow$  CREA-NODO(STATO-INIZIALE[problema])
  loop do
    vicino  $\leftarrow$  il successore di nodo_corrente di valore più alto
    if VALORE[vicino]  $\leq$  VALORE[nodo_corrente] then return STATO[nodo_corrente]
    nodo_corrente  $\leftarrow$  vicino

```

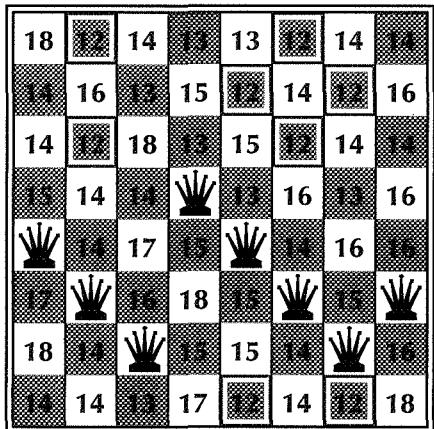
Figure 4.11 L'algoritmo di ricerca hill-climbing (nella versione denominata **ascesa più ripida**), che rappresenta la tecnica più semplice di ricerca locale. A ogni passo il nodo corrente viene rimpiazzato dal vicino migliore: in questa versione ciò significa il vicino con il **VALORE** più alto, ma se usassimo una stima euristica h del costo il vicino prescelto sarebbe quello con h minimo.

Per illustrare l'hill-climbing torneremo al **problema delle 8 regine** presentato a pag. 88. Gli algoritmi di ricerca locale tipicamente usano una **formulazione a stato completo**, in cui ogni stato ha 8 regine sulla scacchiera, una per colonna. La funzione successore restituisce tutti gli stati possibili generati muovendo una singola regina in un'altra casella della stessa colonna (così, ogni stato ha $8 \times 7 = 56$ successori). La **funzione euristica di costo h** è il numero di coppie di regine che si stanno attaccando a vicenda, direttamente o indirettamente. Il minimo globale di questa funzione è zero, che si verifica solo nel caso delle soluzioni. La Figura 4.12(a) mostra uno stato con $h = 17$. La figura mostra anche i valori di tutti i successori, i migliori dei quali hanno $h = 12$. Tipicamente gli algoritmi hill-climbing, quando ce n'è più d'uno, scelgono a caso nell'insieme dei migliori successori.

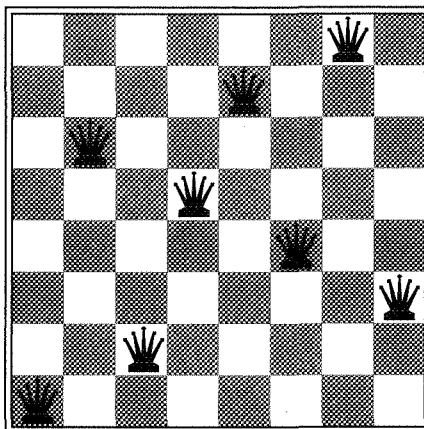
L'algoritmo hill-climbing viene talvolta chiamato **ricerca locale greedy**, perché sceglie uno stato vicino "buono" senza pensare a come andrà avanti. Benché l'avarsia sia considerata uno dei sette **vizi capitali**, gli algoritmi "avari" (**greedy**) si comportano spesso molto bene. L'hill-climbing spesso procede molto rapidamente verso la soluzione, perché di solito è abbastanza facile migliorare uno stato sfavorevole. Ad esempio, dallo stato nella Figura 4.12(a) ci vogliono solo cinque passi per raggiungere quello della Figura 4.12(b), che ha $h = 1$ ed è vicinissimo a una soluzione. Sfortunatamente, spesso l'hill-climbing rimane bloccato per le seguenti ragioni:

- ♦ **massimi locali:** un massimo locale è un picco più alto degli stati vicini, ma inferiore al **massimo globale**. Gli algoritmi hill-climbing che raggiungono la vicinanza di un massimo locale saranno **attratti verso il picco**, ma rimarranno bloccati lì senza poter andare altrove. La Figura 4.10 illustra schematicamen-

ricerca locale greedy



(a)



(b)

Figura 4.12 (a) Uno stato del problema delle 8 regine con una stima euristica di costo $h = 17$. Sono indicati i valori di h di ogni possibile successore ottenuto muovendo una regina nella sua colonna, e le mosse migliori sono evidenziate. (b) Un minimo locale nello spazio degli stati delle 8 regine; lo stato ha $h = 1$ ma ogni successore ha un costo più alto.

te il problema. Più concretamente, lo stato nella Figura 4.12(b) è in effetti un massimo locale (cioè, un minimo locale della funzione di costo h); ogni singola mossa di una regina peggiora la situazione;

- ♦ **crest:** una cresta (*ridge*) è mostrata nella Figura 4.13. Le creste danno origine a una sequenza di massimi locali molto difficili da esplorare da parte degli algoritmi greedy;
- ♦ **plateau:** un plateau è un'area del panorama dello spazio degli stati in cui la funzione di valutazione è piatta. Può essere un massimo locale piatto, da cui non è possibile fare ulteriori progressi, oppure una spalla (*shoulder*), da cui si potrà salire ulteriormente (v. Figura 4.10). Una ricerca hill-climbing potrebbe non essere capace di uscire da un plateau.

In ognuno di questi casi, l'algoritmo raggiunge un punto dal quale non riesce a compiere ulteriori progressi. Partendo da uno stato generato casualmente del problema delle 8 regine, l'algoritmo hill-climbing si blocca l'86% delle volte, risolvendo solo il 14% delle istanze. Funziona molto velocemente, richiedendo in media solo 4 passi per trovare una soluzione e 3 quando si blocca: non male, per uno spazio degli stati con $8^8 \approx 17$ milioni di stati.

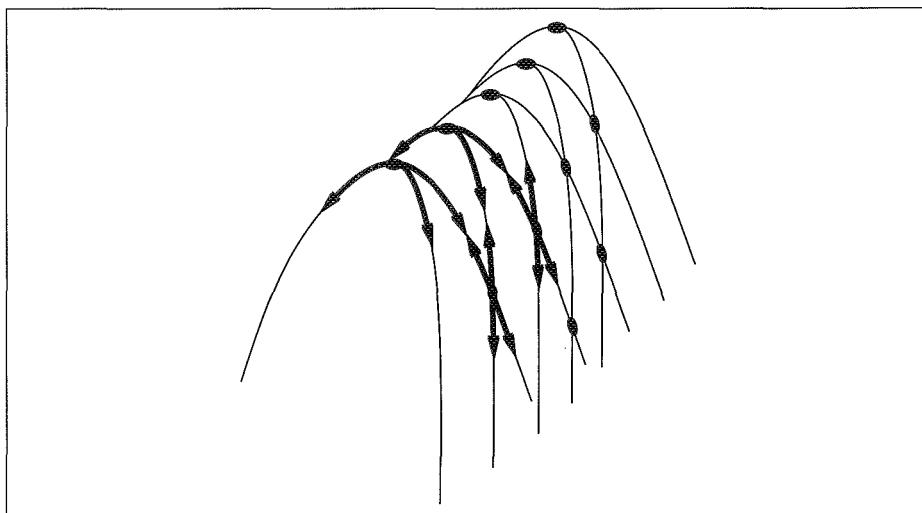


Figura 4.13 Il grafico mostra perché le creste causano difficoltà agli algoritmi di hill-climbing. La griglia di stati (piccoli ovali scuri) è sovrapposta a una cresta che sale da sinistra a destra, creando una sequenza di massimi locali che non sono direttamente collegati l'uno all'altro. Da ogni massimo locale, tutte le azioni possibili puntano "verso valle".

quando il miglior aveva lo stesso valore = com.

L'algoritmo nella Figura 4.11 si blocca se raggiunge un plateau dove il miglior successore ha lo stesso valore dello stato corrente. Potrebbe valere la pena di continuare la ricerca con una mossa laterale, nella speranza che il plateau sia effettivamente una spalla, come si vede nella Figura 4.10? La risposta solitamente è sì, ma bisogna stare attenti. Se permettiamo sempre di eseguire un numero illimitato di mosse laterali, l'algoritmo potrebbe andare in ciclo infinito una volta raggiunto un massimo locale piatto che non è una spalla. Una soluzione diffusa è porre un limite al numero di mosse laterali consecutive: per esempio, potremmo dire che nel problema delle 8 regine sono ammesse al massimo 100 mosse laterali consecutive. Questo alza la percentuale delle istanze di problemi risolte dall'hill-climbing dal 14% al 94%. Il successo però costa: adesso l'algoritmo impiega una media di 21 passi per ogni soluzione e 64 per ogni fallimento.

mossa laterale

Sono state inventate molte varianti dell'hill-climbing. L'hill-climbing stocastico sceglie a caso tra tutte le mosse che vanno verso l'alto: la probabilità della scelta può essere influenzata dalla "pendenza" delle mosse. Normalmente questo algoritmo converge più lentamente di quello che sceglie sempre la mossa più conveniente, ma in alcuni panorami di stati è capace di trovare soluzioni migliori. L'hill-climbing con prima scelta implementa la precedente versione stocastica generando casualmente i successori fino a ottenerne uno preferibile allo stato corrente. Questa strategia è molto buona quando uno stato ha molti (cioè, migliaia di) successori. L'Esercizio 4.16 vi chiederà di investigarla.

hill-climbing stocastico

hill-climbing con prima scelta

hill-climbing con
riavvio casuale

Gli algoritmi hill-climbing descritti fin qui sono incompleti, e spesso non riescono a trovare un obiettivo anche quando esiste perché rimangono bloccati sui massimi locali. L'hill-climbing con riavvio casuale adotta il vecchio motto, “se all’inizio non ce la fai riprova, e poi riprova ancora”. L’algoritmo conduce una serie di ricerche hill-climbing partendo da stati iniziali generati casualmente,⁸ ferman-
dosi quando raggiunge un obiettivo. È completo con probabilità tendente a 1, per la banale ragione che prima o poi dovrà generare, come stato iniziale, proprio un obiettivo. Se ogni ricerca hill-climbing ha una probabilità p di successo, il numero atteso di riavvii richiesti è $1/p$. Per istanze del problema delle 8 regine senza mosse laterali, $p \approx 0,14$, quindi saranno necessarie circa 7 iterazioni per trovare uno stato obiettivo (6 fallimenti e 1 successo). Il numero atteso di passi è il costo dell’iterazione che ha successo più $(1 - p)/p$ volte il costo del fallimento, o circa 22 passi. Permettendo le mosse laterali, sono richieste in media $1/0,94 \approx 1,06$ iterazioni e $(1 \times 21) + (0,06/0,94) \times 64 \approx 25$ passi. Nel caso delle 8 regine, quindi, l’hill-climbing a riavvio casuale è davvero molto efficace. Anche con tre milioni di regine, quest’approccio può trovare soluzioni in meno di un minuto.⁹

Il successo dell’hill-climbing dipende molto dalla forma del panorama dello spazio degli stati: se ci sono pochi massimi locali e plateau, quello con riavvio casuale troverà molto velocemente una buona soluzione. D’altra parte, molti problemi reali hanno un panorama che assomiglia più a una famiglia di porcospini adagiata su un pavimento piatto, con piccoli porcospini appoggiati alla punta di ogni spina dei porcospini più grandi, e così via *ad infinitum*. I problemi NP-difficili tipicamente hanno un numero esponenziale di massimi locali in cui ci si può incastrare. Nonostante questo, spesso è possibile trovare un massimo locale ragionevolmente buono con un numero abbastanza piccolo di riavvii.

Simulated annealing

Un algoritmo hill-climbing che non scende mai “a valle” verso stati con valore più basso (o costo più alto) sarà certamente incompleto, perché può rimanere bloccato in corrispondenza di un massimo locale. Di contro un’esplorazione completamente randomizzata, che si muove verso uno stato scelto a caso nell’insieme dei successori, è completa ma estremamente inefficiente. Sembra ragionevole, quindi, cercare di combinare in qualche modo l’hill-climbing con un’esplorazione casuale in modo da ottenere sia l’efficienza che la completezza: un algoritmo che fa proprio questo è il simulated annealing (letteralmente, “tempra simulata”). In metallurgia, la tempra è il processo usato per indurire i metalli o il vetro riscaldandoli ad

simulated annealing

⁸ Generare uno stato casuale in uno spazio degli stati definito implicitamente può essere di per sé un problema difficile.

⁹ Luby et al. (1993) hanno dimostrato che, in certi casi, conviene far ripartire un algoritmo di ricerca randomizzato dopo una quantità prefissata di tempo e che questo approccio può essere molto più efficiente che permettere a ogni ricerca di continuare indefinitamente. Impedire o limitare il numero di mosse laterali è un esempio di quest’approccio.

altissime temperature e raffreddandoli gradualmente, permettendo così al materiale di cristallizzare in uno stato a bassa energia. Per capire il simulated annealing, cambiamo il punto di vista dalla salita di colline alla discesa di gradiente (cioè la minimizzazione del costo) e immaginiamoci alle prese con il problema di far entrare una pallina da ping-pong nella fessura più profonda di una superficie corrugata. Se lasciamo semplicemente rotolare la pallina, si fermerà in corrispondenza di un minimo locale. Se scuotiamo la superficie, possiamo far uscire la pallina da questi avvallamenti poco profondi: il trucco è di scuotere abbastanza da spostare la pallina fuori dai minimi locali, ma non così tanto da farla uscire anche dal minimo globale. La soluzione proposta dal simulated annealing è di cominciare a scuotere molto (con un'alta temperatura) e poi ridurre gradualmente l'intensità dello scuotimento (riducendo la temperatura).

discesa di gradiente

Il ciclo più interno dell'algoritmo (Figura 4.14) è abbastanza simile all'hill-climbing: stavolta però, invece della mossa migliore, viene scelta una mossa casuale. Se la mossa migliora la situazione, viene sempre accettata; in caso contrario l'algoritmo la accetta con una qualche probabilità inferiore a 1. La probabilità decresce esponenzialmente con la "cattiva qualità" della mossa, misurata dal peggioramento

```

function SIMULATED-ANNEALING(problema, raffreddamento) returns uno stato soluzione
  inputs: problema, un problema
           velocità_raffreddamento, una corrispondenza dal tempo alla "temperatura"
  variabili locali: nodo_corrente, un nodo
                     successivo, un nodo
                     T, una "temperatura" che controlla la probabilità
                     di compiere passi verso il basso

  nodo_corrente  $\leftarrow$  CREA-NODO(STATO-INIZIALE[problema])
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  velocità_raffreddamento[t]
    if T = 0 then return nodo_corrente
    successivo  $\leftarrow$  un successore scelto a caso di nodo_corrente
     $\Delta E \leftarrow$  VALORE[successivo] - VALORE[nodo_corrente]
    if  $\Delta E > 0$  then nodo_corrente  $\leftarrow$  successivo
    else nodo_corrente  $\leftarrow$  successivo solo con probabilità  $e^{\Delta E/T}$ 

```

Figura 4.14 L'algoritmo simulated annealing, una versione dell'hill-climbing stocastico che permette di compiere mosse "a valle". Le mosse verso il basso sono accettate facilmente nelle prime fasi del raffreddamento ma con probabilità sempre decrescente man mano che il tempo passa. L'input *velocità_raffreddamento* determina il valore di *T* in funzione del tempo.

$$\Delta E \gg T \quad T \gg \text{bias} \quad \dots$$

ΔE della valutazione. La probabilità decresce anche con la "temperatura" T , che scende costantemente: le mosse cattive saranno accettate più facilmente all'inizio, in condizioni di temperatura alta, e diventeranno sempre meno probabili man mano che T si abbassa. Si può dimostrare che, se la temperatura decresce abbastanza lentamente, l'algoritmo troverà un ottimo globale con probabilità tendente a 1.

Il simulated annealing è stato usato estensivamente per risolvere problemi di configurazione VLSI nei primi anni '80. In seguito è stato applicato ampiamente nello scheduling delle fabbriche e in altri problemi di ottimizzazione su larga scala. Nell'Esercizio 4.16 vi sarà richiesto di confrontare le sue prestazioni con quelle dell'hill-climbing con riavvio casuale su un problema a n regine.

Ricerca local beam

ricerca local beam

Memorizzare un solo nodo può sembrare una reazione estrema ai problemi legati alle limitazioni di memoria. L'algoritmo di ricerca local beam¹⁰ tiene traccia di k stati anziché uno solo. All'inizio comincia con k stati generati casualmente: ad ogni passo, sono generati i successori di tutti i k stati. Se uno qualsiasi di essi è un obiettivo, l'algoritmo termina; altrimenti sceglie i k successori migliori dalla lista e ricomincia.

A prima vista, una ricerca local beam con k stati potrebbe sembrare nulla più di k ricerche a riavvio casuale eseguite in parallelo anziché una dopo l'altra. In realtà, i due algoritmi sono piuttosto differenti. In una ricerca a riavvio casuale, ogni processo di ricerca è del tutto indipendente dagli altri. In una ricerca local beam, informazione utile viene passata dall'uno all'altro dei k thread di ricerca paralleli. Ad esempio, se uno stato genera diversi buoni successori mentre quelli degli altri $k - 1$ stati sono tutti cattivi, è come se il primo dicesse a tutti gli altri "venite qui, l'erba è più verde!". L'algoritmo abbandona rapidamente le ricerche infruttuose e sposta la sue risorse dove si stanno verificando i progressi migliori.

Nella sua forma più semplice la ricerca local beam può soffrire di una carenza di diversificazione tra i k stati, che possono concentrarsi rapidamente in una piccola regione dello spazio degli stati rendendo così questa ricerca poco più di una versione costosa dell'hill-climbing. Una variante chiamata beam search stocastica, analoga all'hill-climbing stocastico, aiuta ad alleviare questo problema: invece di scegliere i k successori migliori tra i candidati, si scelgono k successori a caso, assegnando ai migliori una maggiore probabilità di scelta. Quest'approccio ricorda quindi il processo di selezione naturale, in cui i "successori" (progenie) di uno "stato" (organismo) popolano la generazione successiva in base al loro "valore" (adattamento o *fitness*).

beam search
stocastica

¹⁰ La ricerca local beam è un adattamento della beam search, che è un algoritmo basato sui cammini.

Algoritmi genetici

Un algoritmo genetico (o AG) è una variante della beam search stocastica in cui gli stati successori sono generati combinando due stati padre invece di modificarne uno. L'analogia con la selezione naturale è la stessa, eccetto che ora la riproduzione non è più asessuata.

Come la beam search, anche gli AG cominciano con un insieme di k stati generati casualmente, chiamati popolazione. Ogni stato, o individuo, è rappresentato da una stringa composta da simboli di un alfabeto finito – tipicamente, cifre binarie. Ad esempio, lo stato di un problema delle 8 regine deve specificare la posizione degli 8 pezzi, ognuno in una colonna di 8 caselle, e quindi richiede $8 \times \log_2 8 = 24$ bit. Alternativamente, lo stato potrebbe essere rappresentato da 8 cifre, ognuna compresa tra 1 e 8 (vedremo più avanti che le due codifiche si comportano in modo diverso). La Figura 4.15(a) presenta una popolazione di quattro stringhe di 8 cifre, che rappresentano altrettanti stati del problema.

La produzione della generazione successiva di stati è mostrata nella Figura 4.15(b)–(e). In (b), ogni stato riceve una valutazione in base a quella che, nella terminologia degli algoritmi genetici, prende il nome di funzione di fitness. Questa funzione deve restituire valori più alti per stati migliori: così, per il problema delle 8 regine, useremo il numero di coppie di regine che non si attaccano a vicenda, che nel caso di una soluzione vale 28. I valori dei quattro stati sono 24, 23, 20 e 11. In questa particolare variante dell'algoritmo genetico, la probabilità di essere prescelti per la riproduzione è direttamente proporzionale al punteggio di fitness: le relative percentuali sono indicate nella figura accanto ai punteggi.

algoritmo genetico

popolazione

individuo

funzione di fitness

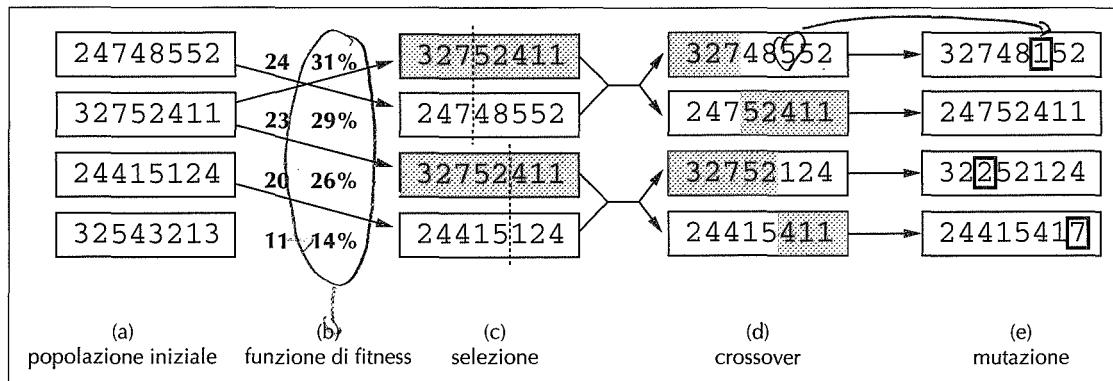
7+6+5+4
12.1

Figura 4.15 L'algoritmo genetico. La popolazione iniziale in (a) è classificata in base alla funzione di fitness in (b), dando come risultato le coppie di (c). Queste ultime danno origine alla progenie in (d), che è poi soggetta a mutazione in (e).

crossover

In (c) viene effettuata la scelta casuale di due coppie per la riproduzione, secondo le probabilità del punto (b). Notate che un individuo viene scelto due volte, un altro nessuna.¹¹ Per ogni coppia che si riproduce viene scelto casualmente un punto di crossover nelle stringhe; nella Figura 4.15 i punti di crossover capitano dopo la terza cifra nella prima coppia e dopo la quinta nella seconda.¹²

In (d) vengono generati i nuovi stati (progenie) incrociando le stringhe dei genitori nel punto di crossover. Ad esempio, il primo figlio della prima coppia riceve le prime tre cifre dal primo genitore e quelle rimanenti dal secondo; il secondo figlio invece riceve le prime tre cifre dal secondo genitore e le altre dal primo. Gli stati del problema a 8 regine coinvolti in questo passo di riproduzione sono mostrati nella Figura 4.16. L'esempio illustra il fatto che, quando due stati genitori sono molto diversi, l'operazione di crossover può produrre uno stato che si discosta molto sia dall'uno che dall'altro. Spesso si verifica che nelle prime fasi la popolazione sia composta da individui molto diversi, dimodoché il crossover (come il simulated annealing) compie grandi spostamenti nello spazio degli stati all'inizio del processo di ricerca e passi più piccoli in seguito, quando la maggior parte degli individui si rassomiglia.

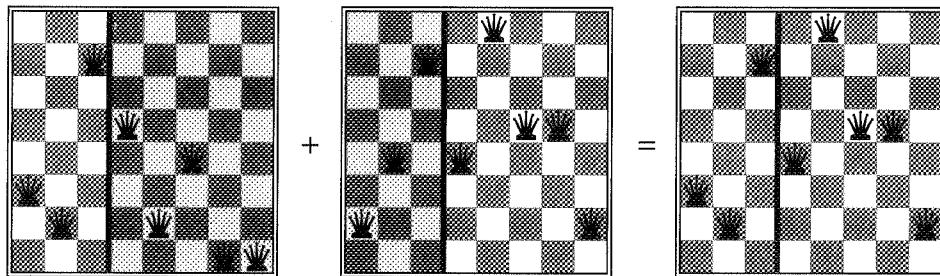


Figura 4.16 Gli stati del problema delle 8 regine corrispondenti ai primi due genitori della Figura 4.15(c) e al primo stato figlio della Figura 4.15(d). Le colonne ombreggiate sono perdute nel passo di crossover, quelle chiare mantenute.

¹¹ Questa regola di selezione ha molte varianti: il metodo del **culling** prevede che siano scartati tutti gli individui sotto una data soglia, ed è stato dimostrato che converge più velocemente della versione casuale (Baum et al., 1995).

¹² È qui che entrano in gioco le differenze nella codifica: se invece di 8 cifre si usano 24 bit, il punto di crossover ha una probabilità pari a 2/3 di capitare a metà di una cifra, in che risulta in una sua modifica sostanzialmente arbitraria.

```

function ALGORITMO-GENETICO(popolazione, FUNZIONE-FITNESS) returns un individuo
  inputs: popolazione, un insieme di individui
    FUNZIONE-FITNESS, una funzione che misura la
      "attitudine a riprodursi" di un individuo

  repeat
    nuova_popolazione  $\leftarrow$  insieme vuoto
    loop for i from 1 to DIMENSIONE(popolazione) do
      x  $\leftarrow$  SELEZIONE-CASUALE(popolazione, FUNZIONE-FITNESS)
      y  $\leftarrow$  SELEZIONE-CASUALE(popolazione, FUNZIONE-FITNESS)
      descendente  $\leftarrow$  RIPRODUZIONE(x, y)
      if (piccola probabilità casuale) then descendente  $\leftarrow$  MUTAZIONE(descendente)
      aggiungi descendente a nuova_popolazione
    popolazione  $\leftarrow$  nuova_popolazione
  until qualche individuo ha un valore di fitness sufficientemente alto,
    oppure è passato troppo tempo.
  return l'individuo migliore nella popolazione in base alla FUNZIONE-FITNESS

```

```

function RIPRODUZIONE(x, y) returns un individuo
  inputs: x, y, due individui genitori

  n  $\leftarrow$  LUNGHEZZA(x)
  c  $\leftarrow$  numero casuale che va da 1 a n
  return CONCATENA(SOTTOSTRINGA(x, 1, c), SOTTOSTRINGA(y, c + 1, n))

```

Figura 4.17 Un algoritmo genetico. L'algoritmo è lo stesso di quello rappresentato nella Figura 4.15, con una variazione: in questa versione ogni coppia di genitori produce un solo discendente, non due.

Infine, nel passo (e), ogni posizione nelle stringhe è soggetta a una **mutazione casuale con una piccola probabilità indipendente**. Come potete vedere si verifica la mutazione di una cifra nella prima, terza e quarta delle stringhe figlie. Nel problema delle 8 regine, questo corrisponde a scegliere a caso una regina e spostarla altrettanto casualmente in una casella della stessa colonna. La Figura 4.17 presenta un algoritmo che implementa tutti i passi descritti.

mutazione

Come la beam search stocastica, gli algoritmi genetici uniscono la tendenza a muoversi verso l'alto con l'esplorazione casuale e con l'interscambio di informazioni tra i diversi thread paralleli della ricerca. Il vantaggio principale degli algoritmi genetici, se ce n'è uno, scaturisce dall'operazione di crossover. Tuttavia può essere dimostrato matematicamente che, se le posizioni del codice genetico sono sottoposte inizialmente a una permutazione casuale, il crossover non offre alcun vantaggio. Intuitivamente, la forza del crossover deriva dalla sua abilità di combinare insieme grandi blocchi di lettere che si sono evolute indipendentemente, alzando così il livello di granularità della ricerca. Ad esempio, potrebbe darsi che mettere le prime tre regine nelle posizioni 2, 4 e 6 (nelle quali non si attaccano tra loro) costituisca un blocco utile che può essere combinato utilmente con altri blocchi per costruire una soluzione.

schema

La teoria degli algoritmi genetici spiega questo funzionamento usando il concetto di schemi, una sottostringa in cui alcune posizioni possono essere lasciate non specificate. Ad esempio, lo schema 246***** descrive tutti gli stati del problema a 8 regine in cui le prime tre regine sono rispettivamente nelle posizioni 2, 4 e 6. Le stringhe che corrispondono allo schema (come 24613578) sono denominate istanze di quello schema. Può essere dimostrato che, se il valore medio di fitness delle istanze di uno schema è sopra la media generale, il numero di istanze di quello schema nella popolazione crescerà nel tempo. Chiaramente quest'effetto non sarà significativo se i bit adiacenti sono totalmente scorrelati tra loro, perché in tal caso ci saranno pochi blocchi contigui capaci di fornire benefici costanti. Gli algoritmi genetici funzionano meglio quando gli schemi corrispondono a componenti significativi della soluzione. Ad esempio, se la stringa è la codifica di un'antenna, gli schemi potrebbero rappresentare componenti come riflettori e deflettori. Un buon componente probabilmente sarà utile in una varietà di progetti diversi. Questo suggerisce che il successo degli algoritmi genetici richiede una cura particolare nella rappresentazione.

Nella pratica, gli AG hanno avuto un grande impatto sui problemi di ottimizzazione, come la configurazione di circuiti e il job-shop scheduling. Oggi come oggi non è chiaro se il successo degli algoritmi genetici derivi dalle loro prestazioni o dalla loro natura così suggestiva ed esteticamente piacevole. Rimane ancora molto lavoro da svolgere per identificare le condizioni in cui lavorano al meglio.

4.4 Ricerca locale in spazi continui

Nel Capitolo 2 abbiamo spiegato la differenza tra ambienti discreti e continui, specificando che la maggior parte degli ambienti reali appartengono alla seconda categoria. Nonostante questo, nessuno degli algoritmi descritti fin qui possono gestire spazi degli stati continui; nella maggior parte dei casi, la funzione successore restituirebbe un numero infinito di stati! Questo paragrafo fornisce un'introduzione molto breve ad alcune tecniche di ricerca locale per trovare soluzioni ottime negli

EVOZIONE E RICERCA

La teoria dell'**evoluzione** è stata sviluppata nel fondamentale libro di Charles Darwin *Sull'origine delle specie per mezzo della selezione naturale* (1859). L'idea centrale è semplice: nella riproduzione si verificano variazioni (denominate **mutazioni**) che saranno conservate nelle generazioni successive in modo proporzionale al loro effetto sulla capacità di riprodursi (fitness).

La teoria di Darwin fu sviluppata senza avere conoscenza dei meccanismi con cui le caratteristiche degli organismi possono essere ereditate o modificate. Le leggi probabilistiche che governano questi processi furono identificate per la prima volta da Gregor Mendel (1866), un monaco che fece esperimenti con i piselli dolci usando quella che chiamava fertilizzazione artificiale. Molto più tardi, Watson e Crick (1953) identificarono la struttura della molecola di DNA e il suo alfabeto, AGTC (adenina, guanina, timina e citosina). Nel modello standard, le variazioni si verificano sia in punti singoli della sequenza di lettere che per "crossover" (per mezzo del quale il DNA di un individuo è generato combinando tra loro lunghe sequenze del DNA dei genitori).

Abbiamo già descritto le analogie tra questo modello e gli algoritmi di ricerca locale; la differenza principale tra la beam search stocastica e l'evoluzione sta nel fatto che la riproduzione è sessuata e i successori sono generati non da uno, ma da *più* organismi. I meccanismi reali dell'evoluzione, comunque, sono molto più ricchi della maggior parte degli algoritmi genetici. Ad esempio, le mutazioni possono coinvolgere inversioni, duplicazioni e spostamenti di grandi blocchi di DNA; alcuni virus "prendono in prestito" DNA da un organismo e lo inseriscono in un altro; e ci sono geni che non fanno altro che copiare se stessi migliaia di volte all'interno del genoma. Esistono persino geni che avvelenano le cellule dei partner potenziali che non contengono una loro copia, aumentando così la propria probabilità di replicazione. La cosa più importante è che *i geni stessi contengono i meccanismi* usati per la riproduzione del genoma e la sua trasformazione in un organismo; negli algoritmi genetici invece tali meccanismi sono racchiusi in un programma separato e non sono codificati nelle stringhe manipolate.

L'evoluzione di Darwin potrebbe sembrare un meccanismo molto inefficiente, avendo generato circa 10^{45} organismi alla cieca senza migliorare affatto le sue euristiche di ricerca. Cinquant'anni prima di Darwin, in effetti, il grande naturalista francese Jean Lamarck (1809) aveva proposto una teoria dell'evoluzione secondo la quale i tratti *acquisiti per adattamento durante la vita di un organismo* sarebbero passati alla sua progenie. Questo processo sarebbe molto efficace, ma non sembra che si verifichi in natura. Molto più tardi, James Baldwin (1896) propose una teoria simile nelle linee generali, che sosteneva che il comportamento appreso durante la vita dell'organismo potesse accelerare il ritmo dell'evoluzione. A differenza di quella di Lamarck, la teoria di Baldwin è del tutto compatibile con quella di Darwin, perché si appoggia sull'effetto della selezione sugli individui che hanno trovato degli ottimi locali nell'insieme di possibili comportamenti consentiti dal loro corredo genetico. Le moderne simulazioni al computer hanno confermato che esiste realmente un "effetto Baldwin", a patto che l'evoluzione "ordinaria" possa creare organismi la cui misura di prestazione interna è correlata in qualche modo alla loro capacità di riprodursi.

spazi continui. La letteratura sull'argomento è vasta; alcuni dei metodi base hanno avuto origine nel XVII secolo, dopo lo sviluppo del calcolo differenziale da parte di Newton e Leibniz.¹³ Avremo l'occasione di usare queste tecniche in molte parti del libro, tra cui i capitoli sull'apprendimento, la visione e la robotica: in pratica, ogni volta che dovremo gestire il mondo reale.

Cominciamo con un esempio. Supponiamo di voler costruire tre nuovi aeroporti in Romania, in posizioni tali da minimizzare la somma dei quadrati delle distanze da ogni città sulla mappa (Figura 3.2) all'aeroporto più vicino. Lo spazio degli stati sarà quindi definito dalle coordinate degli aeroporti: (x_1, y_1) , (x_2, y_2) e (x_3, y_3) . Questo spazio ha *sei dimensioni*; possiamo anche dire che gli stati sono definiti da *sei variabili* (in generale gli stati sono definiti da un vettore *n*-dimensionale \mathbf{x}). Muoversi in questo spazio significa spostare uno o più aeroporti sulla mappa. La funzione obiettivo $f(x_1, y_1, x_2, y_2, x_3, y_3)$ è abbastanza facile da calcolare in un particolare stato, una volta determinate le città più vicine, ma è abbastanza difficile da scrivere in forma generale.

Un modo di evitare il problema della continuità è semplicemente **discretizzare** l'intorno di ogni stato. Ad esempio, possiamo decidere di muovere solo un aeroporto per volta, in direzione x oppure y , e solo di una distanza prefissata $\pm \delta$. Con 6 variabili, questo significa che ogni stato avrà 12 possibili successori. Fatto questo, potremo applicare uno qualsiasi degli algoritmi di ricerca locale che abbiamo descritto. È anche possibile applicare direttamente l'hill-climbing stocastico o il simulated annealing, senza discretizzare lo spazio. Questi algoritmi scelgono casualmente i successori, cosa che può essere fatta generando vettori casuali di lunghezza δ .

Molti metodi cercano di usare il **gradiente** di un panorama per trovare un massimo. Il gradiente della funzione obiettivo è un vettore ∇f che fornisce la grandezza e la direzione della pendenza più ripida. Per il nostro problema, abbiamo

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right).$$

In alcuni casi, possiamo trovare un massimo risolvendo l'equazione $\nabla f = 0$. Questo potrebbe essere fatto, ad esempio, se stessimo piazzando un solo aeroporto; la soluzione sarebbe la media aritmetica delle coordinate di tutte le città. In molti casi, tuttavia, quest'equazione non può essere risolta in forma chiusa. Con tre aeroporti, ad esempio, l'espressione del gradiente dipende dalle città che sono più vicine a ogni

¹³ La lettura di questo paragrafo sarà facilitata se si possiede una conoscenza generale di calcolo multivariato e di aritmetica vettoriale.

aeroporto nello stato corrente. Questo significa che possiamo calcolare il gradiente *localmente* ma non *globalmente*. Anche così, possiamo ancora eseguire l'hill-climbing seguendo la massima pendenza e aggiornando lo stato secondo la formula

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}),$$

dove α è una costante piccola. In altri casi, la funzione obiettivo potrebbe non essere affatto disponibile in forma differenziabile: ad esempio, il valore di un particolare insieme di posizioni degli aeroporti potrebbe essere calcolato da un package di simulazione economica. In questi casi si può determinare il cosiddetto **gradiente empirico**, valutando la risposta a piccoli incrementi e decrementi in ogni coordinata. La ricerca con gradiente empirico è come quella hill-climbing a pendenza massima, eseguita però in una versione discretizzata dello spazio degli stati.

La frase “ α è una costante piccola” nasconde in realtà un'enorme quantità di metodi per determinare il suo valore. Il problema principale è che se α è troppo piccola sono richiesti troppi passi; se è troppo grande, la ricerca potrebbe “arrivare lunga” e superare il punto di massimo. La tecnica della **line search** (ricerca unidimensionale) cerca di superare questo dilemma estendendo la direzione del gradiente (di solito raddoppiando progressivamente il valore di α) finché f non comincia a diminuire. Il punto in cui questo accade diventa il nuovo stato corrente. Fatto ciò, ci sono diverse scuole di pensiero circa il metodo da seguire per scegliere la nuova direzione.

Per molti problemi, l'algoritmo più efficace rimane il venerabile metodo **Newton–Raphson** (Newton, 1671; Raphson, 1690). Si tratta di una tecnica generale per trovare le radici delle funzioni, ovvero per risolvere equazioni nella forma $g(x) = 0$. Il metodo funziona calcolando una nuova stima della radice x secondo la formula di Newton

$$x \leftarrow x - g(x)/g'(x).$$

Per trovare un massimo o un minimo di f , dobbiamo trovare un \mathbf{x} tale che il gradiente sia zero ($\nabla f(\mathbf{x}) = 0$). Quindi $g(x)$ nella formula di Newton diventa $\nabla f(\mathbf{x})$, e la funzione di aggiornamento può essere scritta in forma vettoriale come

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x}),$$

dove $\mathbf{H}_f(\mathbf{x})$ è la matrice **Hessiana** delle derivate seconde, i cui elementi H_{ij} sono dati da $\partial^2 f / \partial x_i \partial x_j$. Dato che la matrice Hessiana ha n^2 componenti, il metodo Newton–Raphson diventa davvero costoso negli spazi a molte dimensioni, ragion per cui ne sono state sviluppate molte versioni approssimate.

I metodi di ricerca locale soffrono i massimi locali, le creste e i plateau negli spazi continui esattamente come in quelli discreti. È possibile utilizzare le tecniche del riavvio casuale e il simulated annealing, che si rivelano spesso molto utili: gli spazi continui a molte dimensioni, comunque, sono posti molto grandi dove è sempre facile perdersi.

gradiente empirico

line search

Newton–Raphson

Hessiana

ottimizzazione vincolata

programmazione lineare

ricerca offline

ricerca online

problemi di esplorazione

Un ultimo argomento che è utile conoscere è l'**ottimizzazione vincolata**. Un problema di ottimizzazione è vincolato se le soluzioni devono soddisfare dei vincoli esplicativi sui valori di ogni variabile. Per esempio, nel nostro problema di posizionamento di aeroporti, potremmo formulare i vincoli che i siti siano all'interno della Romania e sulla terraferma (anziché nel mezzo di un lago). La difficoltà dei problemi di ottimizzazione vincolata dipende dalla natura dei vincoli e della funzione obiettivo. La categoria meglio conosciuta è quella dei problemi di **programmazione lineare**, in cui i vincoli sono espressi da diseguaglianze lineari che formano una regione *convessa* e la funzione obiettivo è anch'essa lineare. Questi problemi possono essere risolti in un tempo polinomiale con il numero di variabili. Sono stati studiati anche problemi con tipi differenti di vincoli e funzioni obiettivo: programmazione quadratica, conica del secondo ordine e così via.

4.5 Agenti per ricerca online e ambienti sconosciuti

Fin qui ci siamo concentrati su agenti che utilizzano algoritmi di **ricerca offline**, calcolando una soluzione prima ancora di mettere piede nel mondo reale (v. Figura 3.1) e poi eseguendola senza far ricorso alle percezioni. Al contrario, nella **ricerca online**¹⁴ un agente opera alternando computazione e azione: prima esegue un'azione, poi osserva l'ambiente e determina l'azione successiva. La ricerca online si presta bene ai domini dinamici o semidinamici, nei quali non c'è tempo per stare immobili a calcolare l'azione successiva; funziona ancora meglio nei domini stocastici. In generale, una ricerca offline deve sviluppare un piano che consideri tutti i possibili eventi e un numero esponenzialmente grande di contingenze; quella online invece deve considerare solo ciò che accade effettivamente. Per esempio, un agente che gioca a scacchi dovrà decidere la prima mossa molto prima di aver previsto lo svolgimento completo della partita.

La ricerca online diventa *necessaria* nei **problem di esplorazione**, nei quali gli stati e le azioni sono sconosciuti all'agente. In questo stato di ignoranza, l'agente dovrà sfruttare le sue azioni come esperimenti per capire cosa fare, e quindi alternare computazione e azione.

L'esempio classico di ricerca online è un robot che si trova in un nuovo edificio che deve esplorare per costruire una mappa per andare da *A* a *B*. I metodi per uscire dai labirinti (una conoscenza necessaria per gli aspiranti eroi dei tempi anti-

¹⁴ Il termine “online” viene usato comunemente nell’informatica per riferirsi ad algoritmi che devono processare i dati in input nel momento in cui sono ricevuti, invece di aspettare di averli tutti disponibili.

chi) rappresentano ulteriori esempi di algoritmi di ricerca online. Quella spaziale, comunque, non è l'unica forma di esplorazione. Considerate un neonato: può eseguire molte azioni, ma non conosce il risultato di nessuna, e ha sperimentato solo pochi dei possibili stati raggiungibili. La scoperta graduale del funzionamento del mondo da parte di un bambino è, in parte, un processo di ricerca online.

Problemi di ricerca online

Un problema di ricerca online può essere risolto solo da un agente che esegue azioni, e non mediante un processo puramente computazionale. Supporremo che l'agente conosca solo quanto segue:

- ◆ AZIONI(s), che restituisce una lista delle azioni permesse nello stato s ;
- ◆ la funzione di costo di passo $c(s, a, s')$: notate che non può essere usata finché l'agente non appurà che s' è il risultato dell'azione;
- ◆ TEST-OBIETTIVO(s) .

Osservate in particolare che l'agente *non può* conoscere i successori di uno stato se non provando effettivamente tutte le azioni possibili. Ad esempio, nel problema del labirinto mostrato nella Figura 4.18, l'agente non sa che muoversi verso l'*Alto* da (1,1) porta a (1,2); e neppure sa, dopo aver compiuto quell'azione, che andare in *Basso* lo riporterà indietro in (1,1). In alcune applicazioni questo grado di ignoranza può essere ridotto: ad esempio, un robot esploratore potrebbe conoscere il funzionamento dei propri movimenti e ignorare solo la posizione degli ostacoli.

Presumeremo che l'agente possa sempre riconoscere uno stato già visitato e che le sue azioni siano deterministiche (queste due ipotesi saranno rilassate nel Capitolo 17). Infine, l'agente potrebbe avere accesso a una funzione euristica ammmissibile $h(s)$ capace di stimare la distanza dallo stato corrente a uno stato obiettivo. Ad esempio, nella Figura 4.18, l'agente potrebbe conoscere la posizione dell'obiettivo ed essere capace di usare come euristica la distanza Manhattan.

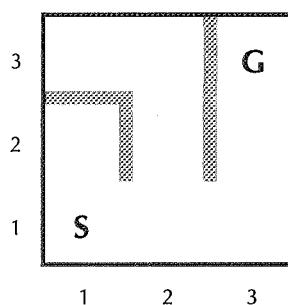


Figura 4.18
Un semplice problema di labirinto. L'agente parte da S e deve arrivare in G , ma non sa nulla dell'ambiente.

rapporto di competitività



adversary argument

esplorabile in modo sicuro

Tipicamente, lo scopo dell’agente è raggiungere uno stato obiettivo minimizzando il costo (un altro possibile scopo può essere semplicemente di esplorare l’intero ambiente). Il costo è rappresentato da quello totale del cammino effettivamente percorso dall’agente. È prassi comune confrontare questo costo con quello del cammino che l’agente potrebbe seguire *se conoscesse in anticipo lo spazio della ricerca*, cioè il cammino minimo (o l’esplorazione completa più breve). Nel linguaggio degli algoritmi online, il risultato di questo confronto viene chiamato **rappporto di competitività** (*competitive ratio*) ed è auspicabile che sia il più piccolo possibile.

Benché quest’ultima sembri una richiesta ragionevole, è facile vedere che in alcuni casi il miglior rapporto di competitività ottenibile è infinito. Ad esempio, se alcune azioni sono irreversibili, la ricerca online potrebbe accidentalmente arrivare a un vicolo cieco da cui non si può raggiungere alcuno stato obiettivo. Potreste trovare poco convincente l’uso del termine “accidentalmente” nella frase precedente: dopotutto, potrebbe sempre esserci un algoritmo che durante l’esplorazione non imbocca vicoli ciechi. Noi affermiamo, per essere più precisi, che *nessun algoritmo può evitare vicoli ciechi in tutti gli spazi degli stati*. Considerate i due spazi degli stati nella Figura 4.19(a), ognuno dei quali ha un vicolo cieco. Agli occhi di un algoritmo di ricerca online che ha visitato gli stati *S* e *A*, i due spazi degli stati appariranno *identici*, e quindi l’algoritmo dovrà prendere la stessa decisione in entrambi i casi. Ne consegue necessariamente che in uno dei due sarà destinato a fallire. Questo è un esempio di quello che gli anglosassoni chiamano **adversary argument**: possiamo immaginare un “avversario” che costruisce lo spazio degli stati mentre l’agente lo esplora, posizionando gli obiettivi e i vicoli ciechi dove gli pare.

I vicoli ciechi sono una vera difficoltà per l’esplorazione robotica: scale, rampe, scarpate e ogni sorta di ostacolo naturale rappresentano altrettante occasioni di azioni irreversibili. Per ora presumeremo semplicemente che lo spazio degli stati sia **esplorabile in modo sicuro** (*safely explorable*), ovvero che si possa sempre arrivare a uno stato obiettivo da ogni stato raggiungibile. Gli spazi degli stati con azioni reversibili, come i labirinti e i rompicapi a tasselli mobili, possono essere modellati con grafi non orientati e sono sempre esplorabili in modo sicuro.

In presenza di cammini di costo illimitato, anche negli ambienti esplorabili in modo sicuro non può essere garantito un rapporto di competitività limitato. È facile dimostrarlo in ambienti con azioni irreversibili, ma in effetti ciò rimane vero anche nel caso reversibile, come si vede nella Figura 4.19(b). Per questa ragione, solitamente le prestazioni degli algoritmi di ricerca online sono descritte in rapporto alle dimensioni dell’intero spazio degli stati e non rispetto alla sola profondità dell’obiettivo più vicino alla radice.

Agenti di ricerca online

Dopo ogni azione, un agente online riceve una percezione che gli comunica lo stato raggiunto; in base a questa esso può arricchire la sua mappa dell’ambiente. La versione corrente della mappa viene utilizzata per decidere l’azione successiva: quest’alternanza tra pianificazione e azione fa sì che gli algoritmi di ricerca online sia-

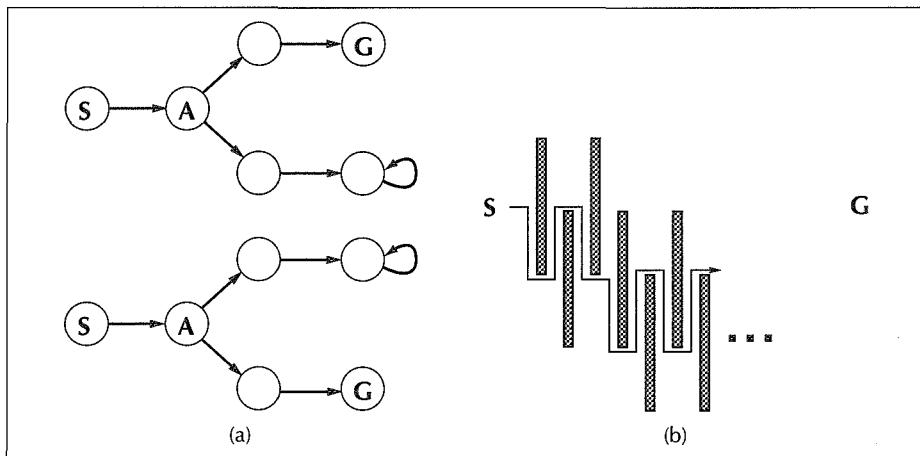


Figura 4.19 (a) Due spazi degli stati che potrebbero condurre un agente di ricerca online in un vicolo cieco. Qualsiasi dato agente fallirà in almeno uno di questi spazi. (b) Un ambiente bidimensionale che può obbligare un agente online a seguire un cammino verso l'obiettivo arbitrariamente inefficiente. Qualsiasi scelta faccia l'agente, l'avversario blocca il cammino corrispondente con un lungo muro sottile, in modo tale che il cammino finale sia molto più lungo del minimo teorico.

no molto diversi da quelli offline che abbiamo considerato finora. Ad esempio, un algoritmo offline come A* ha la possibilità di espandere un nodo in una parte dello spazio e poi subito dopo un altro da un'altra parte, perché l'espansione dei nodi viene effettuata mediante azioni simulate e non reali. Un algoritmo online, invece, può espandere solamente il nodo che sta occupando fisicamente. Per evitare di spostarsi da un capo all'altro dell'albero, sembra più sensato eseguire le espansioni seguendo un ordine *locale*. La ricerca in profondità ha esattamente questa proprietà, perché (tranne che nel caso del backtracking) il nodo successivo è sempre figlio di quello appena espanso.

La Figura 4.20 presenta un agente che svolge ricerche online in profondità. Quest'agente memorizza la sua mappa in una tabella, $risultato[a, s]$, che registra lo stato risultante dall'esecuzione dell'azione a nello stato s . Ogni volta che nello stato corrente c'è un'azione non ancora esplorata, l'agente la prova. Le difficoltà sorgeranno quando l'agente ha provato tutte le azioni in uno stato: nella ricerca offline, lo stato è semplicemente cancellato dalla coda; in quella online, l'agente deve tornare fisicamente sui propri passi. Per la ricerca in profondità, questo significa tornare nello stato più recente da cui l'agente è entrato in quello corrente. Per far questo viene tenuta in memoria una tabella che elenca, per ogni stato, gli stati predecessori a cui l'agente non è ancora ritornato con il backtracking. Se l'agente non ha più stati a cui ritornare, la sua ricerca è completa.

```

function AGENTE-ONLINE-DFS( $s'$ ) returns un'azione
  inputs:  $s'$ , una percezione che identifica lo stato corrente
  static: risultato, una tabella inizialmente vuota indicizzata con azioni e stati
         unexplored, una tabella che elenca, per ogni stato visitato,
         le azioni non ancora provate
         unbacktracked, una tabella che elenca, per ogni stato visitato,
         i backtrack non ancora provati
   $s, a$ , rispettivamente stato e azione precedenti, inizialmente null

  if TEST-OBIETTIVO( $s'$ ) then return stop
  if  $s'$  è un nuovo stato then  $\text{unexplored}[s'] \leftarrow \text{AZIONI}(s')$ 
  if  $s$  non è null then do
     $\text{risultato}[a, s] \leftarrow s'$ 
    aggiungi  $s$  nella prima posizione di unbacktracked[ $s'$ ]
  if unexplored[ $s'$ ] è vuoto then
    if unbacktracked[ $s'$ ] è vuoto then return stop
    else  $a \leftarrow$  un'azione  $b$  tale che  $\text{risultato}[b, s'] = \text{POP}(\text{unbacktracked}[s'])$ 
  else  $a \leftarrow \text{POP}(\text{unexplored}[s'])$ 
   $s \leftarrow s'$ 
  return  $a$ 

```

Figura 4.20 Un agente di ricerca online che utilizza un'esplorazione in profondità. L'agente è applicabile solo a spazi di ricerca in cui i passaggi da uno stato all'altro si possono sempre percorrere in entrambe le direzioni.

Raccomandiamo ai lettori di ricostruire su carta il progresso di AGENTE-ONLINE-DFS applicato al labirinto della Figura 4.18. È abbastanza facile vedere che l'agente, nel caso peggiore, attraverserà ogni collegamento nello spazio degli stati esattamente due volte. Per un'esplorazione, questo risultato è ottimo; per trovare un obiettivo, d'altra parte, il rapporto di competitività potrebbe risultare arbitrariamente alto se l'agente “se ne va a spasso” per una lunga escursione quando c'è uno stato obiettivo proprio accanto a quello iniziale. Una variante online della ricerca ad approfondimento iterativo risolve questo problema; se l'ambiente è un albero uniforme, il rapporto di competitività di un agente così fatto sarà una piccola costante.

Per via del suo uso del backtracking, AGENTE-ONLINE-DFS funziona solo negli spazi degli stati dove le azioni sono reversibili. Ci sono algoritmi leggermente più complessi che possono operare in tutti gli spazi degli stati, ma nessuno di essi ha un rapporto di competitività limitato.

Ricerca locale online

Come quella in profondità, anche la **ricerca hill-climbing** gode della proprietà di località nelle espansioni dei suoi nodi. In effetti, dato che tiene in memoria solamente lo stato corrente, la ricerca hill-climbing è già un algoritmo online! Sfortunatamente, nella sua forma più semplice non è molto utile perché l'agente si può bloccare in corrispondenza di un massimo locale. Per di più, non si può neppure usare il meccanismo del riavvio casuale, perché l'agente non può trasferirsi in un nuovo stato.

Invece dei riavvii casuali, per esplorare l'ambiente si potrebbe considerare l'uso di un **random walk** (letteralmente, “passeggiata casuale”). Un random walk sceglie semplicemente a caso una delle azioni possibili nello stato corrente; è possibile prediligere quelle che non sono ancora state provate. È facile dimostrare che una passeggiata casuale *prima o poi* troverà un obiettivo o completerà la sua esplorazione, a patto che lo spazio degli stati sia finito.¹⁵ D'altra parte, il processo può essere molto lento. La Figura 4.21 mostra un ambiente in cui un random walk richiederà un numero esponenziale di passi per raggiungere l'obiettivo, perché a ogni passo la probabilità di tornare indietro è doppia rispetto a quella di avanzare. Naturalmente l'esempio è fittizio, ma ci sono molti spazi degli stati nel mondo reale la cui topologia causa simili “trappole” per i random walk.

Arricchire la ricerca hill-climbing con *memoria* anziché casualità si rivela un approccio più efficace. L'idea è memorizzare la “miglior stima corrente” $H(s)$ del costo per raggiungere l'obiettivo da ogni stato visitato. $H(s)$ all'inizio non è altro che la stima euristica $h(s)$, ma viene aggiornata man mano che l'agente acquisisce

random walk

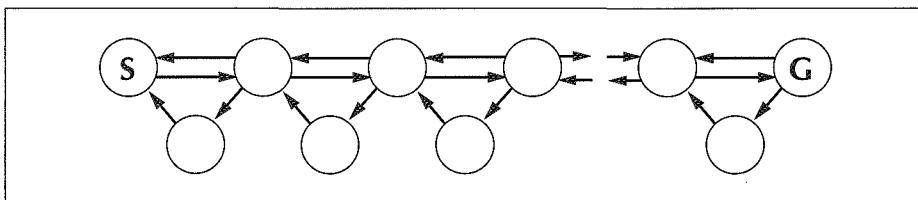


Figura 4.21 Un ambiente in cui una “passeggiata casuale” richiederà un numero di passi esponenzialmente grande per raggiungere l'obiettivo.

¹⁵ Il caso infinito è molto più spinoso. Le passeggiate casuali sono complete su griglie infinite mono- e bidimensionali, ma non quando la griglia è tridimensionale! In quest'ultimo caso, la probabilità che il cammino ritorni mai al punto iniziale è solo di circa 0,3405 (v. Hughes, 1995, per un'introduzione generale).

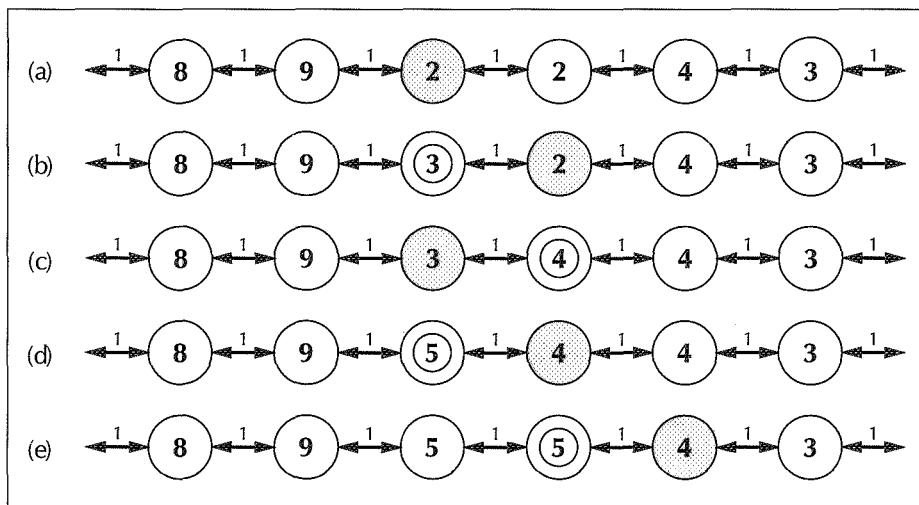


Figura 4.22 Cinque iterazioni di LRTA* in uno spazio degli stati monodimensionale. Ogni stato è etichettato con $H(s)$, la stima corrente del costo per raggiungere un obiettivo, e ogni arco è etichettato con il suo costo di passo. Lo stato ombreggiato indica la posizione dell'agente, e a ogni iterazione i valori aggiornati di $H(s)$ sono evidenziati con un cerchio doppio.

esperienza nello spazio degli stati. La Figura 4.22 mostra un semplice esempio in uno spazio monodimensionale. In (a), l'agente sembra bloccato in un minimo locale piatto, nello stato ombreggiato. Invece di restare lì, l'agente deve seguire quello che sembra il cammino migliore verso l'obiettivo in base alle stime correnti di costo per i suoi vicini. Il costo stimato per arrivare all'obiettivo passando per un vicino s' corrisponde al costo per spostarsi in s' sommato a quello stimato per muoversi da lì all'obiettivo, cioè $c(s, a, s') + H(s')$. Nell'esempio, le due azioni possibili hanno costi stimati $1 + 9$ e $1 + 2$, quindi la scelta migliore sembra andare a destra. Ora è chiaro che il costo stimato di 2 per lo stato ombreggiato era esageratamente ottimistico. Dato che partendo da quella posizione la mossa migliore costa 1 e porta a uno stato che si trova ad almeno 2 passi dall'obiettivo, lo stato ombreggiato dev'essere almeno 3 passi distante dall'obiettivo e il suo valore H dev'essere aggiornato di conseguenza, come si vede nella Figura 4.22(b). Continuando questo processo, l'agente si muoverà ancora due volte avanti e indietro, aggiornando H ogni volta e "appiattendo" il minimo locale fino a sfuggirne sulla destra.

La Figura 4.23 riporta un agente che implementa questo schema, che prende il nome di "A* con apprendimento in tempo reale" (indicato con LRTA*, acronimo di *learning real-time A**). Come AGENTE-ONLINE-DFS, anche questo costruisce una mappa dell'ambiente usando la tabella *risultato*. L'algoritmo aggiorna la stima del costo dello stato che ha appena lasciato e poi sceglie la mossa "apparentemente migliore" in base alle stime correnti dei costi. Un aspetto importante è che si pensa sempre che le azioni che non sono ancora state provate in uno stato s con-

```

function AGENTE-LRTA*( $s'$ ) returns un'azione
  inputs:  $s'$ , una percezione che identifica lo stato corrente
  static: risultato, una tabella inizialmente vuota indicizzata per azioni e stati
   $H$ , una tabella di stime di costi inizialmente vuota, indicizzata per stati
   $s, a$ , rispettivamente stato e azione precedenti, inizialmente null

  if TEST-OBIETTIVO( $s'$ ) then return stop
  if  $s'$  è un nuovo stato (non in  $H$ ) then  $H[s'] \leftarrow h(s')$ 
  unless  $s$  is null
    risultato[ $a, s$ ]  $\leftarrow s'$ 
     $H[s] \leftarrow \min_{b \in \text{AZIONI}(s)} \text{COSTO-LRTA}^*(s, b, \text{risultato}[b, s], H)$ 
   $a \leftarrow$  un'azione  $b$  in  $\text{AZIONI}(s')$  che minimizza  $\text{COSTO-LRTA}^*(s', b, \text{risultato}[b, s'], H)$ 
   $s \leftarrow s'$ 
  return  $a$ 

function COSTO-LRTA*( $s, a, s', H$ ) returns una stima di costo
  if  $s'$  è indefinito then return  $h(s)$ 
  else return  $c(s, a, s') + H[s']$ 
```

Figura 4.23 AGENTE-LRTA* seleziona un'azione in base ai valori degli stati adiacenti, che vengono aggiornati mentre l'agente si muove nello spazio degli stati.

ducano immediatamente all'obiettivo con il minimo costo possibile, cioè $h(s)$. Questo **ottimismo in condizioni di incertezza** incoraggia l'agente a esplorare nuovi cammini potenzialmente promettenti.

ottimismo in condizioni
di incertezza

Un agente LRTA* troverà sempre un obiettivo, a patto che l'ambiente sia finito ed esplorabile in modo sicuro. A differenza di A*, comunque, l'algoritmo non è completo in spazi degli stati infiniti: in alcuni casi può lasciarsi sviare senza possibilità di ritorno. Dato un ambiente di n stati, nel caso pessimo lo può esplorare in $O(n^2)$ passi, ma spesso si comporta molto meglio. LRTA* appartiene a una grande famiglia di agenti online che possono essere definiti specificando in modi diversi la regola di selezione delle azioni e quella di aggiornamento delle stime.

Apprendimento nella ricerca online

Lo stato iniziale di ignoranza degli agenti che svolgono ricerche online fornisce diverse opportunità per l'apprendimento. Prima di tutto un agente deve imparare la "mappa" dell'ambiente (o, più precisamente, le conseguenze di ogni azione in ogni stato) semplicemente memorizzando le sue esperienze. Notate che l'ipotesi che gli ambienti siano deterministici significa che è sufficiente sperimentare ogni azione una sola volta. In un secondo momento gli agenti che effettuano ricerche locali devono acquisire stime più accurate del valore di ogni stato utilizzando adeguate regole di aggiornamento, come abbiamo visto per LRTA*. Nel Capitolo 21 vedremo che questi aggiornamenti a un certo punto convergono ai valori *esatti* di ogni stato, a patto che l'agente esplori lo spazio degli stati nel modo corretto. Una volta noti i valori esatti, si possono prendere decisioni ottime semplicemente muovendosi verso il successore di valore più alto: in altre parole, la strategia di hill-climbing puro diventa ottima.

Se avete seguito il nostro suggerimento di seguire passo passo il comportamento di AGENTE-ONLINE-DFS nell'ambiente della Figura 4.18, avrete notato che non è particolarmente sagace. Ad esempio, dopo aver visto che l'azione *Alto* passa da (1,1) a (1,2), l'agente continua a non sapere che l'azione *Basso* torna indietro in (1,1), o che la stessa azione *Alto* serve anche a passare da (2,1) a (2,2), da (2,2) a (2,3) eccetera. In generale, ci piacerebbe che l'agente imparasse che *Alto* aumenta il valore della coordinata *y* (a meno che non ci sia un ostacolo), che *Basso* lo diminuisce e così via.

Affinché questo accada, servono due cose. Prima di tutto è necessaria una rappresentazione formale ed esplicitamente manipolabile di questo tipo di regole generali; finora infatti quest'informazione è rimasta chiusa in una scatola nera chiamata funzione successore. La Parte III di questo libro sarà dedicata a quest'argomento. Inoltre, ci serviranno algoritmi che possono costruire regole generali utili partendo dalle specifiche osservazioni dell'agente: questi saranno trattati nel Capitolo 18.

4.6 Riepilogo

Questo capitolo ha preso in esame l'applicazione di euristiche per ridurre i costi di ricerca. Abbiamo presentato diversi algoritmi che le utilizzano e verificato che l'ottimalità ha sempre un prezzo molto alto in termini di costo della ricerca, anche quando sono disponibili delle buone euristiche.

- ♦ La ricerca best-first non è altro che una RICERCA-GRAFO che sceglie per l'espansione i nodi non ancora espansi che, secondo qualche criterio, risultano di costo minimo. Gli algoritmi best-first usano tipicamente una funzione euristica $h(n)$ che stima il costo di una soluzione partendo da un nodo n .

- ◆ La ricerca **best-first greedy** o “golosa” espande sempre i nodi con $h(n)$ minima. Non è ottima, ma è spesso efficiente.
- ◆ La ricerca **A*** espande i nodi a cui corrisponde una minima $f(n) = g(n) + h(n)$. A* è completa e ottima se possiamo garantire che $h(n)$ sia ammissibile (per RICERCA-ALBERO) o consistente (per RICERCA-GRAFO). La complessità spaziale di A* è ancora proibitiva.
- ◆ Le prestazioni degli algoritmi di ricerca euristici dipendono dalla qualità della funzione euristica utilizzata. Talvolta si possono costruire buone euristiche rilassando la definizione del problema, memorizzando in un database di pattern i costi precalcolati delle soluzioni di alcuni sottoproblemi, o imparando dall’esperienza accumulata in una classe di problemi.
- ◆ RBFS e SMA* sono algoritmi di ricerca robusti e ottimi che usano una quantità limitata di memoria; se hanno a disposizione abbastanza tempo possono risolvere problemi che A* non può trattare a causa dei suoi requisiti spaziali.
- ◆ I metodi di *ricerca locale* come **hill-climbing** lavorano su formulazioni del problema a stato completo, tenendo in memoria solo un piccolo numero di nodi. Sono stati sviluppati diversi algoritmi stocastici, tra cui il **simulated annealing**, che fornisce soluzioni ottime quando viene impostata una “velocità di raffreddamento” appropriata. Si possono usare diversi metodi di ricerca locale anche per risolvere problemi negli spazi continui.
- ◆ Un **algoritmo genetico** è una ricerca hill-climbing stocastica in cui viene tenuta in memoria una grande popolazione di stati. I nuovi stati sono generati con la **mutazione** e il **crossover**, che genera nuovi individui mediante la combinazione di coppie di stati della popolazione.
- ◆ I **problemi di esplorazione** sorgono quando un agente non conosce gli stati e le azioni del suo ambiente. Se l’ambiente è esplorabile in modo sicuro, gli agenti possono svolgere una **ricerca online** per costruire una mappa e trovare un obiettivo, se esiste. Per sfuggire ai minimi locali un metodo efficace è aggiornare le stime euristiche in base all’esperienza.

Note storiche e bibliografiche

L’uso di informazione euristica per la risoluzione di problemi compare in uno dei primi articoli di Simon e Newell (1958), ma il termine “ricerca euristica” e l’uso di funzioni euristiche per stimare la distanza verso l’obiettivo sono di poco successive (Newell e Ernst, 1965; Lin, 1965). Doran e Michie (1966) hanno svolto estesi studi sperimentali sulla ricerca euristica applicandola a una varietà di problemi, in particolare i rompicapi a 8 e 15 tasselli. Benché Doran e Michie abbiano sviluppato un’analisi teorica sulla lunghezza dei cammini e la “penetrazione” (il rapporto

tra la lunghezza del cammino e il numero totale di nodi esaminati) nella ricerca euristica, sembra che abbiano ignorato l'informazione fornita dalla lunghezza del cammino corrente. L'algoritmo A*, che incorpora nella ricerca euristica proprio la lunghezza del cammino corrente, fu sviluppato da Hart, Nilsson e Raphael (1968), con qualche correzione successiva (Hart et al., 1972). Dechter e Pearl (1985) hanno dimostrato che A* è ottimamente efficiente.

L'articolo originale su A* introduceva il requisito della consistenza delle funzioni euristiche. Il requisito di monotonicità fu introdotto da Pohl (1977) come sostituto più semplice del precedente, ma Pearl (1984) ha dimostrato che i due sono assimilabili. Molti algoritmi prima di A* usavano l'equivalente delle liste aperte e chiuse; tra questi ci sono la ricerca in ampiezza, in profondità e a costo uniforme (Bellman, 1957; Dijkstra, 1959). Il lavoro di Bellman, in particolare, ha mostrato l'importanza dei costi di cammino additivi per semplificare gli algoritmi di ottimizzazione.

Pohl (1970, 1977) fu il primo a studiare la relazione tra gli errori nelle funzioni euristiche e la complessità temporale di A*. La dimostrazione che A* richiede solo un tempo lineare se l'errore della funzione euristica è limitato da una costante può essere trovata in Pohl (1977) e Gaschnig (1979). Pearl (1984) rinforzò questo risultato permettendo una crescita logaritmica dell'errore. Il "fattore effettivo di ramificazione" come misura dell'efficienza della ricerca euristica fu proposto da Nilsson (1971).

Ci sono molte varianti dell'algoritmo A*. Pohl (1973) propose l'uso di un *peso dinamico*, che utilizza come funzione di valutazione una somma pesata $f_w(n) = w_g g(n) + w_h h(n)$ della lunghezza corrente del cammino e della funzione euristica, anziché la semplice somma $f(n) = g(n) + h(n)$ usata da A*. I pesi w_g e w_h sono determinati dinamicamente durante lo svolgimento della ricerca. Si può dimostrare che l'algoritmo di Pohl è ε -ammissibile, ovvero che può garantire di trovare soluzioni all'interno di un fattore $1 + \varepsilon$ della soluzione ottima, dove ε è un parametro passato all'algoritmo. L'algoritmo A_ε^* (Pearl, 1984) ha stessa proprietà, e può scegliere un nodo qualsiasi nella frontiera a patto che il suo *f*-costo sia all'interno di un fattore $1 + \varepsilon$ dell'*f*-costo minimo tra tutti i nodi della frontiera. La selezione può essere effettuata in modo tale da minimizzare il costo della ricerca.

A* e altri algoritmi di ricerca nello spazio degli stati sono strettamente imparentati con le tecniche di *branch-and-bound* ampiamente diffuse nel campo della ricerca operativa (Lawler e Wood, 1966). Le relazioni tra ricerca nello spazio degli stati e branch-and-bound sono state investigate in profondità (Kumar e Kanal, 1983; Nau et al., 1984; Kumar et al., 1988). Martelli e Montanari (1978) hanno mostrato la connessione tra la programmazione dinamica (v. Capitolo 17) e certi tipi di ricerca nello spazio degli stati. Kumar e Kanal (1988) hanno tentato una "grande unificazione" di ricerca euristica, programmazione dinamica e tecniche branch-and-bound sotto il nome di CDP (acronimo di *Composite Decision Process*).

Dato che i computer tra la fine degli anni '50 e l'inizio dei '60 avevano al massimo qualche migliaio di parole di memoria, non stupisce che la ricerca euristica a memoria limitata sia stata subito oggetto di ricerca. Il Graph Traverser (Doran e Michie, 1966), uno dei primi programmi di ricerca, sceglie un'azione dopo aver svolto una ricerca best-first fino al limite della memoria disponibile. IDA* (Korf, 1985a, 1985b) è stato il primo algoritmo di ricerca euristica ottimo, a memoria limitata e di largo uso, e ne sono state sviluppate molte varianti. Un'analisi dell'efficienza di IDA* e dei suoi problemi con euristiche a valori reali è contenuta in Patrick et al. (1992).

RBFS (Korf, 1991, 1993) è in realtà un po' più complicato di quanto mostrato nella Figura 4.5, che assomiglia di più a un algoritmo sviluppato indipendentemente e chiamato **espansione iterativa**, o IE (Russell, 1992). RBFS usa un limite inferiore oltre che superiore; i due algoritmi si comportano in maniera identica con euristiche ammissibili, ma RBFS espande i nodi in ordine best-first anche quando l'euristica non è ammissibile. L'idea di tener traccia del miglior cammino alternativo è apparsa per la prima volta nell'elegante implementazione in Prolog di A* da parte di Bratko (1986) e nell'algoritmo DTA* (Russell e Wefald, 1991). Quest'ultimo lavoro discute anche gli spazi degli stati di metalivello e l'apprendimento di metalivello.

L'algoritmo MA* apparve in Chakrabarti et al. (1989). SMA*, o Simplified MA*, emerse da un tentativo di implementare MA* come algoritmo di confronto per IE (Russell, 1992). Kaindl e Khorsand (1994) hanno applicato SMA* per realizzare un algoritmo di ricerca bidirezionale molto più veloce dei precedenti. Korf e Zhang (2000) descrivono un approccio *divide et impera*, mentre Zhou e Hansen (2002) hanno introdotto la ricerca A* a memoria limitata su grafo. Korf (1995) offre una panoramica sulle tecniche di ricerca a memoria limitata.

L'idea che si possano derivare euristiche ammissibili dal rilassamento dei problemi è apparsa in un fondamentale articolo di Held e Karp (1970), che hanno usato l'euristica del "minimo albero di copertura" per risolvere il problema del commesso viaggiatore (v. Esercizio 4.8).

L'automazione del processo di rilassamento fu implementata con successo da Prieditis (1993), sviluppando il lavoro precedente di Mostow (Mostow e Prieditis, 1989). L'uso di database di pattern per derivare euristiche ammissibili è dovuto a Gasser (1995) e Culberson e Schaeffer (1998); i database di pattern disgiunti sono descritti da Korf e Felner (2002). L'interpretazione probabilistica delle euristiche è stata investigata in profondità da Pearl (1984) e Hansson e Mayer (1989).

La fonte più esaustiva sulle euristiche e gli algoritmi di ricerca che le utilizzano è di gran lunga il libro di Pearl (1984) *Heuristics*, che copre in modo particolarmente buono la grande varietà di varianti e derivazioni di A*, includendo dimostrazioni rigorose delle loro proprietà formali. Kanal e Kumar (1988) hanno curato un'antologia di articoli importanti sulla ricerca euristica. I nuovi studi sull'argomento sono documentati regolarmente sulla rivista *Artificial Intelligence*.

espansione iterativa

Le tecniche di ricerca locale hanno una lunga storia nella matematica e nell'informatica. In effetti, il metodo Newton–Raphson (Newton, 1671; Raphson, 1690) può essere considerato un metodo di ricerca locale molto efficiente per spazi continui in cui è disponibile informazione sul gradiente. Brent (1973) è un punto di riferimento classico per gli algoritmi di ottimizzazione che non richiedono tali informazioni. La beam search, che abbiamo presentato come un algoritmo di ricerca locale, è nata come una variante ad ampiezza limitata della programmazione dinamica per il riconoscimento vocale nel sistema HARPY (Lowerre, 1976). Un algoritmo simile è analizzato in profondità da Pearl (1984, Cap. 5).

In anni recenti la ricerca locale è stata rinvigorita da risultati sorprendentemente buoni per grandi problemi di soddisfacimento di vincoli come quello a n regine (Minton et al., 1992) e nel campo del ragionamento logico (Selman et al., 1992) grazie anche all'apporto della casualità, dello svolgimento di ricerche multiple simultanee e di altre migliorie. La rinascita di quelli che Christos Papadimitriou ha chiamato “algoritmi New Age” ha anche suscitato interesse tra gli informatici teorici (Koutsoupias e Papadimitriou, 1992; Aldous e Vazirani, 1994). Nel campo della ricerca operativa, ha guadagnato popolarità una variante dell'hill-climbing chiamata **ricerca tabù** (Glover, 1989; Glover e Laguna, 1997). Ispirandosi a modelli della memoria a breve termine negli esseri umani, quest'algoritmo mantiene una “lista tabù” di k stati precedentemente visitati a cui non si può ritornare; oltre a migliorare l’efficienza delle ricerche su grafo, questo può permettere all'algoritmo di evitare alcuni minimi locali. Un'altra utile miglioria dell'hill-climbing è l'algoritmo STAGE (Boyan e Moore, 1998), che usa i massimi locali trovati mediante l'hill-climbing a riavvio casuale per farsi un'idea della forma generale del panorama. L'algoritmo “adatta” una superficie liscia a un insieme di massimi locali e poi calcola analiticamente il massimo globale della superficie stessa: questa diventa il nuovo punto di partenza della ricerca. L'algoritmo ha mostrato di funzionare bene su problemi difficili. (Gomes et al., 1998) hanno dimostrato che i tempi di esecuzione degli algoritmi che effettuano sistematicamente il backtracking spesso hanno una **distribuzione a coda pesante**, che significa che la probabilità di avere un tempo di esecuzione molto lungo è maggiore di quella che si potrebbe prevedere se i tempi di esecuzione avessero una distribuzione normale. Ciò fornisce una giustificazione teorica alla tecnica dei riavvii casuali.

Il simulated annealing fu descritto per la prima volta da Kirkpatrick et al. (1983), che si ispirò direttamente all'**algoritmo Metropolis** usato per simulare complessi sistemi fisici (Metropolis et al., 1953) e che fu inventato, si dice, durante una cena sociale a Los Alamos. Il simulated annealing ora è una branca a sé stante, su cui vengono pubblicati ogni anno centinaia di articoli.

La ricerca di soluzioni ottime negli spazi continui è l'argomento di molte discipline, tra cui la **teoria dell'ottimizzazione**, la **teoria del controllo ottimo** e il **calcolo delle variazioni**. Introduzioni adeguate (e orientate alla pratica) sono fornite da Press et al. (2002) e Bishop (1995). La **programmazione lineare (LP)** è stata una delle prime applicazioni dei computer; l'**algoritmo del simplex** (Wood e

ricerca tabù

distribuzione a coda pesante

Dantzig, 1949; Dantzig, 1949) è ancora usato nonostante la sua complessità esponenziale nel caso pessimo. Karmarkar (1984) ha sviluppato un pratico algoritmo in tempo polinomiale per la programmazione lineare.

Il lavoro di Sewall Wright (1931) sul concetto di **panorama di fitness** è stato un importante precursore dello sviluppo degli algoritmi genetici. Negli anni '50 molti studiosi di statistica, tra cui Box (1957) e Friedman (1959), applicarono tecniche evolutive per risolvere problemi di ottimizzazione: l'approccio tuttavia non divenne popolare finché Rechenberg (1965, 1973) non introdusse **strategie evolutive** per risolvere problemi di ottimizzazione applicati ai profili alari. Negli anni '60 e '70 John Holland (1975) propugnò l'uso degli algoritmi genetici, sia come utile strumento che come metodo per espandere la nostra comprensione dell'adattamento, biologico e non (Holland, 1995). Il movimento della **vita artificiale** (Langton, 1995) porta quest'idea un passo oltre, considerando i prodotti degli algoritmi genetici come *organismi* anziché soluzioni di problemi. Il lavoro svolto in questo campo da Hinton e Nowlan (1987) e da Ackley e Littman (1991) ha contribuito molto a chiarire le implicazioni dell'effetto Baldwin. Per un'introduzione generale all'evoluzione, raccomandiamo caldamente Smith e Szathmáry (1999).

La maggior parte dei confronti tra gli algoritmi genetici e gli altri approcci (in particolare l'hill-climbing stocastico) hanno riscontrato che i primi convergono più lentamente (O'Reilly e Oppacher, 1994; Mitchell et al., 1996; Juels e Wattenberg, 1996; Baluja, 1997). Comprensibilmente questi risultati non sono popolari nella comunità degli algoritmi genetici, ma i recenti tentativi di formulare la ricerca basata sulla popolazione come una forma approssimata di apprendimento bayesiano potrebbero aiutare a colmare la distanza tra gli AG e i loro critici (Pelikan et al., 1999). La teoria dei **sistemi dinamici quadratici** potrebbe anche spiegare le prestazioni degli algoritmi genetici (Rabani et al., 1998). Lohn et al. (2001) contiene un esempio di applicazione degli AG alla progettazione di un'antenna, Larrañaga et al. (1999) al problema del commesso viaggiatore.

Il campo della **programmazione genetica** è strettamente imparentato con quello degli algoritmi genetici: la differenza principale sta nel fatto che le rappresentazioni sottoposte a mutazione e combinazione non sono stringhe di bit ma interi programmi. I programmi sono codificati sotto forma di alberi di espressioni; le espressioni possono essere scritte in un linguaggio standard come Lisp o in una forma speciale per rappresentare circuiti, controller di robot e così via. In modo analogo il crossover lavora su sottoalberi anziché su sottostringhe. Questa forma di mutazione garantisce che i discendenti siano costituiti da espressioni ben formate, cosa che non accadrebbe se i programmi fossero manipolati direttamente in forma di stringhe.

L'interesse nella programmazione genetica è stato ravvivato di recente dal lavoro di John Koza (Koza, 1992, 1994), ma la tecnica risale almeno ai primi esperimenti sul codice macchina di Friedberg (1958) e sugli automi a stati finiti da parte di Fogel et al. (1966). Come per gli algoritmi genetici, sull'effettiva efficacia

strategie evolutive

vita artificiale

programmazione genetica

della tecnica c'è un certo dibattito. Koza et al. (1999) descrive un certo numero di esperimenti sull'automazione della progettazione di circuiti, condotti usando la programmazione genetica.

Le riviste *Evolutionary Computation* e *IEEE Transactions on Evolutionary Computation* trattano gli algoritmi e la programmazione genetica; si possono trovare articoli anche in *Complex Systems*, *Adaptive Behavior* e *Artificial Life*. I congressi più importanti sono la *International Conference on Genetic Algorithms* e la *Conference on Genetic Programming*, recentemente unificati nella *Genetic and Evolutionary Computation Conference*. I libri di Melanie Mitchell (1996) e David Fogel (2000) offrono buone panoramiche sull'argomento.

Gli algoritmi per l'esplorazione di spazi degli stati sconosciuti sono stati studiati per molti secoli. La ricerca in profondità in un labirinto può essere implementata non staccando mai la mano sinistra dal muro; per evitare i cicli basta marcare ogni incrocio. La ricerca in profondità fallisce in presenza di azioni irreversibili; il problema più generale dell'esplorazione dei **grafi di Eulero** (in cui ogni nodo ha un numero uguale di archi entranti e uscenti) è stato risolto da un algoritmo di Hierholzer (1873). Il primo studio approfondito degli algoritmi per il problema dell'esplorazione di grafi arbitrari fu svolto da Deng e Papadimitriou (1990), che hanno sviluppato un algoritmo del tutto generale, ma hanno anche mostrato che non si può avere un rapporto di competitività limitato nell'esplorazione di grafi generici. Papadimitriou e Yannakakis (1991) hanno esaminato il problema di trovare cammini verso un obiettivo in ambienti di pianificazione geometrici (in cui tutte le azioni sono reversibili) mostrando che nel caso di ostacoli quadrati si può avere un rapporto di competitività piccolo, mentre la presenza di ostacoli rettangolari generici impedisce di ottenere un rapporto limitato (v. Figura 4.19).

L'algoritmo LRTA* fu sviluppato da Korf (1990) come parte dello studio sulla ricerca **in tempo reale**, rivolta agli ambienti in cui l'agente deve agire dopo aver condotto la ricerca per un tempo limitato (una situazione molto comune nei giochi a due giocatori). LRTA* in effetti è un caso speciale degli algoritmi di apprendimento per rinforzo per ambienti stocastici (Barto et al., 1995). La sua politica di ottimismo in condizioni di incertezza (andare sempre nello stato più vicino non visitato) può risultare in un'esplorazione che, nel caso non informato, è meno efficiente di una semplice ricerca in profondità (Koenig, 2000). Dasgupta et al. (1994) mostra che la ricerca online ad approfondimento iterativo è ottimamente efficiente quando si deve trovare un obiettivo in un albero uniforme senza informazione euristica. Sono state sviluppate molte varianti informate di LRTA* con diversi metodi per condurre la ricerca e aggiornare l'informazione nella porzione conosciuta del grafo (Pemberton e Korf, 1992). A tutt'oggi, non c'è una piena comprensione di come trovare obiettivi con efficienza ottima usando informazione euristica.

In questo capitolo non abbiamo trattato l'argomento degli algoritmi di ricerca **parallela**, anche perché ciò richiederebbe una lunga discussione sulle architetture parallele per computer. La ricerca parallela sta diventando un tema impor-

grafo di Eulero

ricerca in tempo reale

ricerca parallela

tante sia nell'IA che nell'informatica teorica. Una breve introduzione alla letteratura dal punto di vista dell'intelligenza artificiale può essere trovato in Mahanti e Daniels (1993).

Esercizi

- 4.1 Simulate passo passo lo svolgimento di una ricerca A* applicata al problema di arrivare a Bucarest partendo da Lugoj e usando l'euristica della distanza in linea d'aria, mostrando la sequenza dei nodi considerati dall'algoritmo e i loro punteggi di f , g e h .
- 4.2 L'**algoritmo del cammino euristico** è una ricerca best-first la cui funzione obiettivo è $f(n) = (2 - w)g(n) + wh(n)$. Per quali valori di w quest'algoritmo è certamente ottimo? (potete presumere che h sia ammissibile). Che tipo di ricerca viene eseguita quando $w = 0$? E quando $w = 1$? E quando $w = 2$?
- 4.3 Dimostrate ognuna delle seguenti affermazioni.
 - a. La ricerca in ampiezza è un caso speciale di quella a costo uniforme.
 - b. La ricerca in ampiezza, quella in profondità e quella a costo uniforme sono tutti casi speciali della ricerca best-first.
 - c. La ricerca a costo uniforme è un caso speciale della ricerca A*.
- 4.4 Inventate uno spazio degli stati in cui una ricerca A* che usa RICERCA-GRAFO restituisce una soluzione subottima con una funzione $h(n)$ ammissibile ma inconsistente.
- 4.5 Abbiamo visto a pagina 129 che l'euristica della distanza in linea d'aria può portare fuori strada una ricerca best-first greedy quando il problema è andare da Iasi a Fagaras. Tuttavia, l'euristica è perfetta nel caso opposto, quando si deve andare da Fagaras a Iasi. Esistono problemi nei quali l'euristica è fuorviante in entrambe le direzioni?
- 4.6 Inventate una funzione euristica per il rompicapo a 8 tasselli che commetta talvolta errori di sovrastima, e mostrate un particolare problema nel quale essa può condurre a una soluzione subottima (potete servirvi di un computer, se volete). Dimostrate che, se l'errore di sovrastima di h non supera mai c , una ricerca A* che utilizza h restituirà una soluzione il cui costo non supererà quello della soluzione ottima più c .
- 4.7 Dimostrate che se un'euristica è consistente, allora dev'essere anche ammissibile. Inventate un'euristica ammissibile ma non consistente.



- 
- 4.8 Il problema del commesso viaggiatore (TSP) può essere risolto usando l'euristica del "minimo albero di copertura" (MST, da *minimum spanning tree*), per stimare il costo del completamento di un percorso parzialmente costruito. Il costo MST di un insieme di città è la minima somma dei costi dei collegamenti di un qualsiasi albero che tocchi tutte le città.
- Mostrate che quest'euristica può essere derivata da una versione rilassata del TSP.
 - Dimostrate che l'euristica MST domina quella della distanza in linea d'aria.
 - Scrivete un generatore di problemi per istanze del TSP in cui le città siano rappresentate da punti casuali in un quadrato di lato unitario.
 - Trovate nella letteratura un algoritmo efficiente per costruire l'euristica MST, e utilizzatela con un algoritmo ammissibile di ricerca per risolvere istanze del TSP.
- 4.9 A pagina 141 abbiamo definito un rilassamento del rompicapo a 8 tasselli in cui è possibile muovere un tassello da A a B se lo spazio B è vuoto. La soluzione esatta di questo problema definisce l'*euristica di Gaschnig* (Gaschnig, 1979). Spiegate perché questa euristica è almeno accurata quanto h_1 (numero di tasselli fuori posto), e mostrate dei casi in cui è più accurata sia di h_1 che di h_2 (distanza Manhattan). Potete suggerire un metodo per calcolare l'*euristica di Gaschnig* in modo efficiente?
- 
- 4.10 Per il rompicapo a 8 tasselli abbiamo fornito due semplici euristiche: la distanza Manhattan e il numero di tasselli fuori posto. Nella letteratura ci sono molte euristiche che sostengono di essere migliori: v. ad esempio Nilsson (1971), Mostow e Priedis (1989) e Hansson et al. (1992). Mettete alla prova queste euristiche implementandole e confrontandone le prestazioni.
- 4.11 Dite il nome degli algoritmi che risultano da ognuno di questi casi speciali.
- Ricerca local beam con $k = 1$.
 - Ricerca local beam con uno stato iniziale e nessun limite al numero di stati memorizzati.
 - Simulated annealing con $T = 0$ sempre (e omettendo il test di terminazione).
 - Algoritmo genetico con dimensione della popolazione $N = 1$.
- 4.12 Talvolta non esiste una buona funzione di valutazione per un problema, ma c'è un buon metodo di confronto, cioè un modo per dire qual è il migliore tra due nodi senza assegnare loro valori numerici. Mostrate che questo è sufficiente per eseguire una ricerca best-first. C'è un'analogia con A*?
- 4.13 Descrivete la correlazione tra la complessità temporale di LRTA* e quella spaziale.

4.14 Supponete che un agente si trovi in un labirinto 3×3 come quello mostrato nella Figura 4.18. L'agente sa che la sua posizione iniziale è $(1,1)$, che l'obiettivo si trova in $(3,3)$ e che le quattro azioni *Alto*, *Basso*, *Sinistra*, *Destra* hanno l'effetto consueto se il cammino non è bloccato da un muro. L'agente *non sa* dove sono posti i muri interni. In ogni stato, l'agente percepisce l'insieme delle azioni legali; inoltre può distinguere i nuovi stati da quelli già visitati.

- Spiegate in che modo questo problema di ricerca online può essere considerato un problema offline nello spazio degli stati-credenza, dove lo stato-credenza iniziale include tutte le possibili configurazioni dell'ambiente. Quant'è grande lo stato-credenza iniziale? E lo spazio degli stati-credenza?
- Qual è il numero di diverse possibili percezioni nello stato iniziale?
- Descrivete i primi rami di un piano di contingenza per questo problema. Quanto sarà grande (approssimativamente) il piano completo? Notate che deve comprendere la soluzione per *ogni possibile ambiente* che soddisfi la descrizione data. Ne consegue che l'alternanza tra ricerca ed esecuzione non è strettamente necessaria anche se l'ambiente è sconosciuto.

4.15 In questo esercizio esploreremo l'uso dei metodi di ricerca locale per risolvere problemi TSP del tipo definito nell'Esercizio 4.8.



- Scrivete un metodo hill-climbing per risolvere problemi TSP. Confrontate i risultati con le soluzioni ottime ottenute con l'algoritmo A* e l'eistica MST (Esercizio 4.8).
- Scrivete un algoritmo genetico per il problema del commesso viaggiatore e confrontatene i risultati con quelli degli altri approcci. La consultazione di Larrañaga et al. (1999) sarà utile per avere dei suggerimenti sulle rappresentazioni.

4.16 Generate un grande numero di istanze di rompicapo a 8 tasselli e di problemi delle 8 regine e risolveteli (quando possibile) usando hill-climbing (nelle varianti a massima pendenza e con prima scelta), hill-climbing con riavvii casuali e simulated annealing. Misurate la percentuale di problemi risolti e i costi di ricerca, e disegnate un grafo che li confronta con il costo della soluzione ottima. Commentate i risultati.



4.17 In quest'esercizio esamineremo l'hill-climbing nel contesto della navigazione robotica, usando come esempio l'ambiente della Figura 3.22.

- Ripetete l'Esercizio 3.16 usando l'hill-climbing. Può capitare che il vostro agente rimanga bloccato in un minimo locale? È possibile che rimanga bloccato se gli ostacoli sono convessi?
- Costruite un ambiente con poligoni non convessi in cui l'agente possa rimanere incastrato.



- 
- c. Modificate l'algoritmo hill-climbing in modo che, invece di eseguire una ricerca a profondità 1 per decidere dove andare, ne svolga una a profondità k . L'agente deve trovare il miglior cammino di lunghezza k e percorrerne un singolo passo, quindi ripetere il processo.
 - d. C'è un k che garantisca al nuovo algoritmo di non rimanere bloccato nei minimi locali?
 - e. Spiegate come in questo caso LRTA* permette all'agente di sfuggire ai minimi locali in questo caso.
- 4.18 Confrontate le prestazioni di A* e RBFS su un insieme di problemi generati casualmente nei domini del rompicapo a 8 tasselli (con distanza Manhattan) e TSP (con MST, v. Esercizio 4.8). Commentate i risultati. Come cambiano le prestazioni di RBFS quando si aggiunge un piccolo numero casuale ai valori dell'euristica nel dominio del rompicapo a 8 tasselli?

Capitolo 5

Problemi di soddisfacimento di vincoli

Nel quale vediamo che considerare gli stati qualcosa di più di piccole scatole nere porta all'invenzione di una quantità di nuovi, potenti metodi di ricerca e a una comprensione più profonda della struttura e della complessità dei problemi.

Nei Capitoli 3 e 4 abbiamo esplorato l'idea che i problemi si possano risolvere eseguendo una ricerca in uno spazio fatto di **stati**, che possono essere valutati usando euristiche specifiche del dominio del problema e sottoposti a test per verificare se sono stati obiettivo. Dal punto di vista dell'algoritmo di ricerca, comunque, ogni stato è una **scatola nera** priva di struttura interna visibile e rappresentata da una struttura dati arbitraria a cui possono accedere solo funzioni *specifiche del problema*: la funzione successore, quella euristica e il test obiettivo.

Questo capitolo prende in esame i **problemi di soddisfacimento di vincoli**, i cui stati e il cui test obiettivo sono conformi a una **rappresentazione standard**, strutturata e molto semplice. È possibile formulare algoritmi di ricerca che sfruttano la struttura degli stati e utilizzano euristiche *di uso generale* anziché *specifiche del problema* per permettere la soluzione di problemi di grandi dimensioni (Paragrafi 5.2 e 5.3). La cosa più importante, forse, è che la rappresentazione standard del test obiettivo rivela la struttura del problema stesso (Paragrafo 5.4). Questo consente di definire metodi per la scomposizione dei problemi e di comprendere il delicato collegamento tra la struttura di un problema e la difficoltà della sua risoluzione.

scatola nera

rappresentazione

problema di soddisfacimento di vincoli
 variabili
 vincoli
 dominio
 valori
 assegnamento
 consistente
 funzione obiettivo

5.1 Problemi di soddisfacimento di vincoli

Formalmente, un **problema di soddisfacimento di vincoli** (o CSP, da *Constraint Satisfaction Problem*) è definito da un insieme di variabili, X_1, X_2, \dots, X_n e da un insieme di vincoli, C_1, C_2, \dots, C_m . Ogni variabile X_i ha un **dominio** non vuoto D_i di possibili valori. Ogni vincolo C_i coinvolge un sottoinsieme delle variabili e ne specifica le combinazioni di valori consentite. Uno stato del problema è definito dall'**assegnamento** di valori ad alcune o tutte le variabili, $\{X_i = v_i, X_j = v_j, \dots\}$. Un assegnamento che non viola alcun vincolo è chiamato **consistente** o legale. Un assegnamento è completo se menziona tutte le variabili; una **soluzione** di un CSP è un assegnamento completo che soddisfa tutti i vincoli. Alcuni CSP richiedono anche che la soluzione massimizzi una **funzione obiettivo**.

Cosa significa tutto questo? Supponiamo che, essendoci stanchi della Romania, stiamo considerando una mappa dell'Australia che mostra tutti i suoi stati e territori, come si vede nella Figura 5.1(a), e che abbiamo il compito di colorare ogni regione di rosso, verde o blu in modo tale che non esistano regioni adiacenti dello stesso colore. Per formulare questo come un CSP, definiamo una variabile per

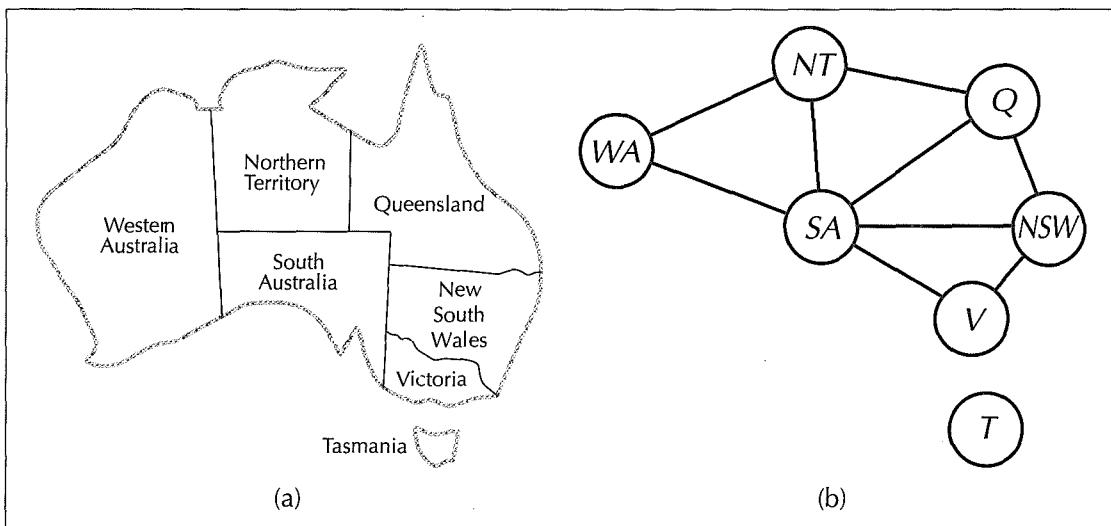


Figura 5.1 (a) I principali stati e territori dell'Australia. La coloratura di questa mappa può essere vista come un problema di soddisfacimento di vincoli. L'obiettivo è assegnare un colore a ogni regione in modo che non esistano due regioni adiacenti con lo stesso colore. (b) Il problema di coloratura di una mappa rappresentato sotto forma di grafo di vincoli.

ogni regione: WA , NT , Q , NSW , V , SA e T . Il dominio di ogni variabile è l'insieme $\{rosso, verde, blu\}$. I vincoli richiedono che le regioni adiacenti abbiano colori distinti; ad esempio, le combinazioni lecite per WA e NT sono le coppie

$$\{(rosso, verde), (rosso, blu), (verde, rosso), (verde, blu), (blu, rosso), (blu, verde)\}.$$

Il vincolo potrebbe essere rappresentato in modo più compatto usando la disegualanza $WA \neq NT$, a patto che l'algoritmo sia in grado di valutare espressioni simili. Ci sono molte soluzioni possibili, come

$$\{WA = rosso, NT = verde, Q = rosso, NSW = verde, V = rosso, SA = blu, T = rosso\}.$$

Può essere utile visualizzare un CSP come un **grafo di vincoli**, come nella Figura 5.1(b). I nodi del grafo corrispondono alle variabili del problema e gli archi ai vincoli.

grafo di vincoli

Esprimere un problema sotto forma di CSP apporta molti importanti benefici. Dato che la rappresentazione degli stati è conforme a uno *standard*, consistendo sempre in un insieme di variabili con i loro valori, la funzione successore e il test obiettivo possono essere scritti in un modo generico che si applica a tutti i CSP. Inoltre è possibile sviluppare euristiche efficaci e generiche che non richiedono esperienza aggiuntiva specifica del dominio. Infine, la struttura del grafo di vincoli può essere usata per semplificare il processo risolutivo, ottenendo in certi casi una riduzione esponenziale della complessità. La rappresentazione CSP è il primo e il più semplice di una serie di schemi di rappresentazione che svilupperemo per tutto il libro.

È facile vedere che si può dare una **formulazione incrementale** di un CSP come fosse un problema di ricerca standard.

- ◆ **Stato iniziale:** l'assegnamento vuoto $\{\}$, nel quale nessuna variabile ha un valore.
- ◆ **Funzione successore:** si può assegnare un valore a una qualsiasi delle variabili che non ce l'hanno ancora, a patto che non confligga con i valori assegnati in precedenza.
- ◆ **Test obiettivo:** l'assegnamento corrente è completo.
- ◆ **Costo di cammino:** un costo costante (per esempio 1) per ogni passo.

Ogni soluzione dev'essere un assegnamento completo e quindi avrà una profondità n con n variabili. Inoltre, l'albero di ricerca si estenderà solo fino a una profondità n . Per queste ragioni, per risolvere i CSP sono molto popolari gli algoritmi di ricerca in profondità (v. Paragrafo 5.2). Notate anche che *il cammino con cui si arriva alla soluzione è sempre irrilevante*. Possiamo quindi usare anche una **formulazione a stato completo**, nella quale ogni stato è un assegnamento completo che può soddisfare i vincoli o no. Con questa formulazione funzionano bene i metodi di ricerca locale (v. Paragrafo 5.3).



domini finiti

Nella forma più semplice di CSP le variabili sono discrete e hanno **domini finiti**. Colorare le mappe appartiene a questa categoria di problemi. Anche il problema delle 8 regine descritto nel Capitolo 3 può essere considerato un CSP a dominio finito, in cui le variabili Q_1, \dots, Q_8 sono le posizioni di ogni regina nella colonne 1, ..., 8 e ogni variabile ha dominio $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Se la dimensione massima del dominio di ogni variabile del CSP è d , il numero di possibili assegnamenti è $O(d^n)$, ovvero esponenziale con il numero di variabili. I CSP a domini finiti includono i **CSP booleani**, le cui variabili possono assumere solo i valori *true* o *false*. Alcuni casi speciali di CSP booleani sono problemi NP-completi, come 3SAT (v. Capitolo 7). Nel caso pessimo, quindi, non possiamo aspettarci di risolvere CSP a domini finiti in un tempo meno che esponenziale. Nella maggior parte delle applicazioni pratiche, comunque, algoritmi CSP di uso generale possono risolvere problemi le cui dimensioni sono *ordini di grandezza* superiori a quelle dei problemi che si possono trattare con gli algoritmi di ricerca generici che abbiamo visto nel Capitolo 3.

domini infiniti

Anche le variabili discrete possono avere **domini infiniti**, come l'insieme dei numeri naturali o quello delle stringhe. Ad esempio, quando si devono organizzare su un calendario i lavori di un cantiere, la data d'inizio di ogni attività è una variabile i cui valori possibili sono numeri interi di giorni a partire dalla data corrente. Con domini infiniti, non è più possibile descrivere i vincoli enumerando tutte le combinazioni di valori accettate: bisogna usare un **linguaggio di specifica di vincoli**. Se per esempio Job_1 , che richiede cinque giorni, deve per forza precedere Job_3 , potremo usare un linguaggio di specifica basato su diseguaglianze algebriche in modo da scrivere $StartJob_1 + 5 \leq StartJob_3$. Non è più possibile neanche risolvere i vincoli enumerando tutti i possibili assegnamenti, dato che il loro numero è infinito. Esistono degli algoritmi speciali (che non discuteremo qui) per trattare **vincoli lineari** su variabili intere, come quello che abbiamo appena scritto: in questi vincoli ogni variabile appare solo in forma lineare. Può essere dimostrato che non esiste alcun algoritmo per risolvere problemi generali con **vincoli non lineari** su variabili intere. In alcuni casi possiamo ridurre i problemi a variabili intere e farli diventare a dominio finito, semplicemente ponendo un limite superiore ai valori di tutte le variabili. Ad esempio, nel problema del cantiere, possiamo imporre come limite la somma della durata di tutte le attività da svolgere.

domini continui

I problemi di soddisfacimento di vincoli con **domini continui** sono molto comuni nel mondo reale, e sono stati studiati approfonditamente nel campo della ricerca operativa. Compilare il programma degli esperimenti per il telescopio spaziale Hubble, per esempio, richiede una temporizzazione estremamente precisa delle osservazioni; l'inizio e la fine di ognuna di esse deve rispettare una serie di vincoli astronomici, di precedenza e di gestione energetica. La categoria meglio conosciuta di CSP a dominio continuo è la **programmazione lineare**, in cui i vincoli devono essere espressi come diseguaglianze lineari che formano una regione *convessa*. I problemi di programmazione lineare possono essere risolti in un tempo polinomiale nel numero delle variabili. Sono stati studiati anche problemi con tipi diversi di vincoli e di funzioni obiettivo: programmazione quadratica, programmazione conica del secondo ordine e così via.

programmazione lineare

Oltre a esaminare i tipi di variabili che possono apparire in un CSP, è utile considerare anche i tipi di vincolo. Quello più semplice è il **vincolo unario**, che interessa il valore di una singola variabile. Ad esempio, potrebbe darsi che gli australiani del sud detestino il colore *verde*. Ogni vincolo unario può essere eliminato semplicemente riformulando il dominio della variabile corrispondente e rimuovendo tutti i valori che violano il vincolo. Un **vincolo binario**, come $SA \neq NSW$, mette in relazione due variabili. Un CSP si dice binario quando comprende solo vincoli binari; come si vede nella Figura 5.1(b) può essere rappresentato con un grafo dei vincoli.

vincolo unario

Vincoli di ordine più alto coinvolgono tre o più variabili. Un esempio ben noto sono i giochi enigmistici di **criptoaritmetica**. Normalmente in questi giochi ogni lettera corrisponde a una cifra diversa. Nel caso della Figura 5.2(a), si potrebbe scrivere un vincolo a sei variabili *Tuttediverse* (F, T, U, W, R, O). Alternativamente, si potrebbe costruire una collezione di vincoli binari come $F \neq T$. Anche i vincoli imposti dall'aritmetica dell'addizione sulle quattro colonne del rompicapo coinvolgono più variabili, e possono essere scritti come segue:

criptoaritmetica

$$O + O = R + 10 \cdot X_1$$

$$X_1 + W + W = U + 10 \cdot X_2$$

$$X_2 + T + T = O + 10 \cdot X_3$$

$$X_3 = F$$

dove X_1, X_2 e X_3 sono **variabili ausiliarie** che rappresentano la cifra (0 o 1) del rapporto da una colonna all'altra. I vincoli di ordine superiore possono essere rappresentati in un **ipergrafo di vincoli**, come quello mostrato nella Figura 5.2(b). I lettori più accorti avranno già notato che il vincolo *Tuttediverse* può essere scomposto in una serie di vincoli binari: $F \neq T$, $F \neq U$ e così via. In effetti, come l'Esercizio 5.11 vi chiede di dimostrare, ogni vincolo di ordine superiore in un dominio finito può essere ricondotto a un insieme di vincoli binari a patto di introdurre un numero sufficiente di variabili ausiliarie. Per questo motivo in questo capitolo tratteremo solo i vincoli binari.

variabili ausiliarie

ipergrafo di vincoli

I vincoli che abbiamo descritto fin qui sono tutti **assoluti**; la loro violazione impedisce di trovare una soluzione. Molti CSP nel mondo reale includono vincoli di **preferenza**, che indicano quali soluzioni sono appunto preferibili. Per esempio, compilando la tabella degli orari delle lezioni, il Prof. X potrebbe gradire di lavorare il mattino, il Prof. Y invece il pomeriggio. Un orario che prevedesse lezioni del Prof. X alle 14.00 sarebbe ancora una soluzione (a meno che il Prof. X per caso non sia anche il preside), ma non ottima. Spesso i vincoli di preferenza possono essere rappresentati come costi sugli assegnamenti di singole variabili: ad esempio, assegnare al Prof. X delle lezioni al pomeriggio costerà 2 punti nella funzione obiettivo globale, mentre le lezioni al mattino costeranno solo 1. Con questa formulazione, i CSP con vincoli di preferenza possono essere risolti usando metodi di ricerca ottimizzata, locali o basati sul cammino. Non discuteremo oltre questo tipo di CSP, ma forniremo qualche indicazione nelle note bibliografiche.

vincoli di preferenza

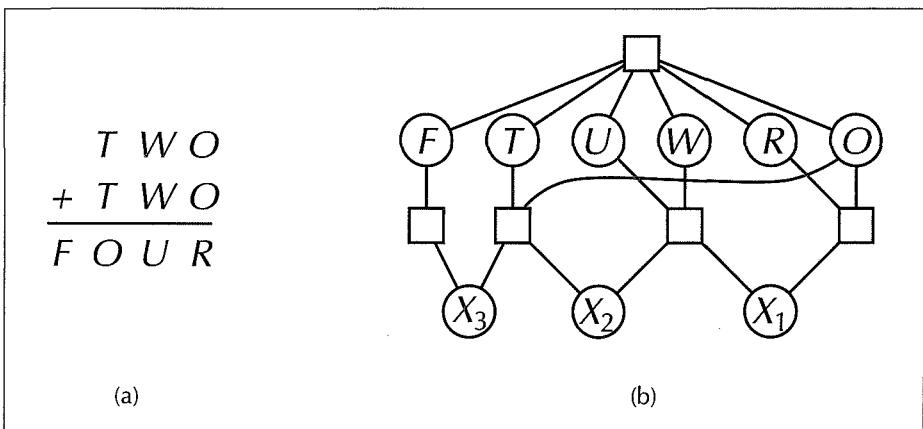


Figura 5.2 (a) Un problema di criptoaritmetica. Ogni lettera indica una singola cifra distinta; lo scopo è trovare una corrispondenza tra lettere e cifre tale che la somma risulti aritmeticamente corretta, con il vincolo aggiuntivo che non sono permessi zeri aggiuntivi alla sinistra dei numeri. (b) L'ipergrafo dei vincoli per il problema di criptoaritmetica, che mostra sia il vincolo *Tuttediverse* che quelli sull'addizione tra le colonne. Ogni vincolo è rappresentato da un box quadrato collegato alle variabili interessate.

5.2 Ricerca con backtracking per CSP

Nel paragrafo precedente abbiamo formulato i CSP come problemi di ricerca: in questo modo, per risolverli si possono usare gli algoritmi dei Capitoli 3 e 4. Supponiamo di effettuare una ricerca in ampiezza sulla formulazione generica del problema CSP del paragrafo precedente. Presto scopriremo qualcosa di terribile: il fattore di ramificazione al primo livello è nd , perché uno qualsiasi dei d valori può essere assegnato a ognuna delle n variabili. Al livello successivo, il fattore di ramificazione è $(n - 1)d$, e così via per n livelli. Anche se i possibili assegnamenti completi sono solo d^n , l'albero generato ha $n! \cdot d^n$ foglie!

La nostra formulazione del problema, apparentemente ragionevole ma in realtà ingenua, ha ignorato una proprietà fondamentale comune a tutti i CSP: la commutatività. Un problema è commutativo se l'ordine di applicazione di un qualsiasi insieme di azioni non ha effetto sul risultato finale. Questo è il caso dei CSP, dato che assegnando valori alle variabili si ottiene sempre lo stesso assegnamento parziale indipendentemente dall'ordine degli assegnamenti. Di conseguenza, *tutti gli algoritmi di ricerca CSP quando generano successori considerano i possibili assegnamenti di una sola variabile in ogni nodo dell'albero di ricerca*. Ad esempio,

commutatività



```

function RICERCA-BACKTRACKING(csp) returns una soluzione, o il fallimento
  return BACKTRACKING-RICORSIVO({ }, csp)

function BACKTRACKING-RICORSIVO(assegnamento, csp) returns una soluzione, o il fallimento
  if assegnamento è completo then return assegnamento
  var  $\leftarrow$  SCEGLI-VARIABILE-NON-ASSEGNATA(VARIABILI[csp], assegnamento, csp)
  for each valore in ORDINA-VALORI-DOMINIO(var, assegnamento, csp) do
    if valore è consistente con assegnamento in base a VINCOLI[csp] then
      aggiungi {var = valore} ad assegnamento
      risultato  $\leftarrow$  BACKTRACKING-RICORSIVO(assegnamento, csp)
      if risultato  $\neq$  fallimento then return risultato
      rimuovi {var = valore} da assegnamento
  return fallimento

```

Figura 5.3 Un semplice algoritmo di backtracking per problemi di soddisfacimento di vincoli. L'algoritmo è modellato sulla ricerca ricorsiva in profondità presentata nel Capitolo 3. Si possono usare le funzioni SCEGLI-VARIABILE-NON-ASSEGNATA e ORDINA-VALORI-DOMINIO per implementare le euristiche di uso generale discusse nel testo.

nella radice dell'albero di ricerca per la coloratura della mappa dell'Australia, potremmo scegliere tra *SA* = rosso, *SA* = verde e *SA* = blu, ma non dovremo mai scegliere tra *SA* = rosso e *WA* = blu. Con questa restrizione il numero di foglie risulta d^n .

Il termine **ricerca con backtracking** indica una ricerca in profondità che assegna valori a una variabile per volta e torna indietro quando non ci sono più valori legali da assegnare. L'algoritmo è riportato nella Figura 5.3: notate che usa effettivamente il metodo di generazione incrementale “un successore per volta”, come abbiamo descritto a pag. 102. Inoltre, per generare un successore non copia l'assegnamento corrente ma lo estende. Dato che la rappresentazione di CSP è standardizzata, non è necessario fornire a RICERCA-BACKTRACKING uno stato iniziale, una funzione successore o un test obiettivo specifici del dominio. Parte dell'albero di ricerca per il problema della mappa australiana è mostrato nella Figura 5.4, in cui le variabili sono assegnate nell'ordine *WA*, *NT*, *Q*,

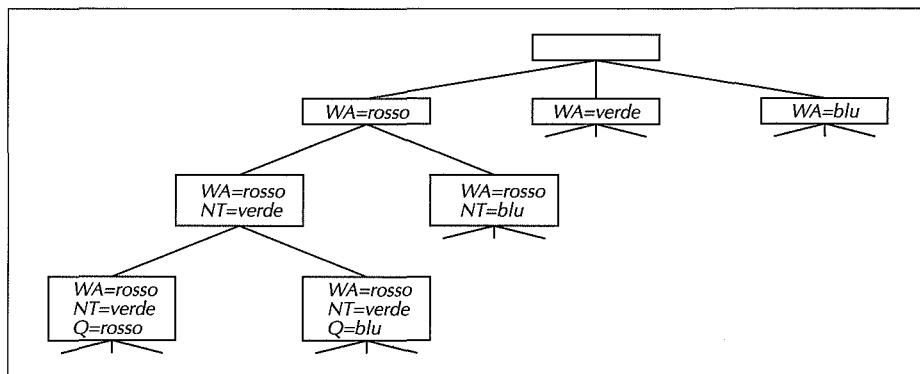
Secondo la terminologia del Capitolo 3, il backtracking semplice è un algoritmo non informato, ragion per cui ci possiamo aspettare che non sia molto efficiente per problemi grandi. I suoi risultati su alcuni problemi standard sono mostrati nella prima colonna della Figura 5.5 e confermano questa previsione.

Nel Capitolo 4 abbiamo ovviato alle scarse prestazioni degli algoritmi di ricerca non informata fornendo loro funzioni euristiche specifiche del dominio, derivate dalla nostra conoscenza del problema. I CSP invece possono essere risolti in modo efficiente senza ricorrere a conoscenze specifiche. Al loro posto, sfrutteremo metodi generali che rispondono alle seguenti domande.

ricerca con
backtracking

Figura 5.4

Una parte dell'albero di ricerca generato dal backtracking semplice per il problema di coloratura di mappa della Figura 5.1.



1. Quale variabile assegnare nel passo successivo, e in quale ordine provare i possibili valori?
2. Quali sono le conseguenze degli assegnamenti della variabile corrente sulle altre variabili non assegnate?
3. Quando un cammino fallisce perché ha raggiunto uno stato in cui una variabile non ha valori legali, è possibile che la ricerca eviti di ripetere tale fallimento nei cammini successivi?

I prossimi sottoparagrafi rispondono a queste domande.

Problema	Backtracking	BT+MRV	Verifica in avanti	VA+MRV	Min-Conflicts
USA	(> 1.000K)	(> 1.000K)		2K	60
n -Regine	(> 40.000K)	13.500K	(> 40.000K)	817K	4K
Zebra	3.859K	1K	35K	0,5K	2K
Casuale 1	415K	3K	26K	2K	
Casuale 2	942K	27K	77K	15K	

Figura 5.5 Confronto di vari algoritmi CSP su diversi problemi. Da sinistra a destra, gli algoritmi sono backtracking semplice, backtracking con euristica MRV, verifica in avanti, verifica in avanti con MRV e ricerca locale a minimi conflitti. In ogni posizione è riportato il numero medio di controlli di consistenza richiesti per risolvere il problema su cinque iterazioni; notate che tutti i numeri, tranne i due in alto a destra, rappresentano delle migliaia (K). Se un numero è indicato tra parentesi significa che non è stata trovata alcuna soluzione nel numero fissato di controlli. Il primo problema consiste nella coloratura a 4 colori dei 50 stati USA. I problemi restanti sono presi da Bacchus e van Run (1995), Tabella 1. Il secondo problema conta il numero totale di controlli richiesti per risolvere tutti i problemi a n regine per n che va da 2 a 50. Il terzo è il "rompicapo della Zebra", descritto nell'Esercizio 5.13. Gli ultimi due sono problemi casuali artificiali (su cui non è stato provato l'algoritmo min-conflicts). I risultati suggeriscono che su tutti questi problemi la verifica in avanti con euristica MRV è preferibile agli altri algoritmi con backtracking, ma non sempre migliore della ricerca locale a minimi conflitti.

Ordinamento di variabili e valori

L'algoritmo di ricerca con backtracking contiene la riga

```
var ← SCEGLI-VARIABILE-NON-ASSEGNATA(VARIABILI[csp], assegnamento, csp).
```

Per default, SCEGLI-VARIABILE-NON-ASSEGNATA seleziona semplicemente la prima variabile non assegnata nell'ordine indicato dalla lista VARIABILI[*csp*]. Quest'ordinamento statico delle variabili raramente dà come risultato la ricerca più efficiente. Ad esempio, dopo gli assegnamenti $WA = \text{rosso}$ e $NT = \text{verde}$ c'è un solo valore possibile per SA , per cui la cosa più sensata come passo successivo sarebbe assegnare $SA = \text{blu}$ invece di occuparsi dell'assegnamento di Q . In effetti, dopo l'assegnamento di SA le scelte per Q , NSW e V sono tutte forzate. L'idea di scegliere la variabile con il minor numero di valori "legali" è chiamata **euristica MRV** (*Minimum Remaining Values*): è stata chiamata anche euristica "della variabile più vincolata" o "fail-first", perché sceglie la variabile per cui è maggiore la probabilità di arrivare presto a un fallimento, potando così l'albero di ricerca. Se c'è una variabile X che non ha nessun possibile valore legale, l'euristica MRV sceglierà X e il fallimento sarà rilevato immediatamente, evitando così ricerche inutili su altre variabili che giungerebbero comunque al fallimento una volta arrivati a X . La seconda colonna della Figura 5.5, etichettata BT+MRV, mostra le prestazioni di quest'euristica che a seconda del problema è da 3 a 3.000 volte superiore al backtracking semplice. Notate che la nostra misura di performance ignora il costo aggiuntivo dovuto al calcolo dei valori dell'euristica; il prossimo sottoparagrafo presenta un metodo che rende gestibile tale costo.

L'euristica MRV non è di nessun aiuto nella scelta della prima regione da colorare, perché all'inizio ognuna di esse ha esattamente tre colori legali. In questo caso, viene in aiuto l'**euristica di grado** (*degree heuristic*). Quest'euristica cerca di ridurre il fattore di ramificazione delle scelte future scegliendo la variabile coinvolta nel maggior numero di vincoli con le altre variabili non assegnate. Nella Figura 5.1, SA è la variabile di grado più alto, 5; le altre variabili hanno grado 2 o 3, eccetto T , che è di grado 0. In effetti, una volta scelta SA , l'applicazione dell'euristica di grado risolve il problema senza passi falsi: per ogni regione si può scegliere uno qualsiasi dei colori accettabili e arrivare in ogni caso a una soluzione senza ricorrere al backtracking. L'euristica MRV solitamente è più potente, ma quella di grado può tornare utile nel caso si verifichino dei "pareggi" nell'ordinamento.

Una volta scelta una variabile, l'algoritmo deve decidere l'ordine con cui esaminare i suoi possibili valori. Per far questo, in alcuni casi può essere efficace l'**euristica del valore meno vincolante**, che predilige il valore che lascia più libertà alle variabili adiacenti sul grafo dei vincoli. Per esempio, supponete che nella Figura 5.1 abbiamo generato l'assegnamento parziale $WA = \text{rosso}$ e $NT = \text{verde}$, e che la nostra prossima scelta riguardi Q . Il blu sarebbe una cattiva scelta, perché elimina l'ultimo valore legale per il vicino di Q , SA . L'euristica del valore meno vincolante

euristica MRV

euristica di grado

valore meno vincolante

preferisce quindi il colore rosso. In generale, l'euristica cerca sempre di lasciare la massima flessibilità ai successivi assegnamenti di variabili. Naturalmente, se stiamo cercando di trovare tutte le soluzioni di un problema e non solo la prima, l'ordine dei valori non ha importanza perché dovremo in ogni caso considerarli tutti. La stessa cosa vale se un problema non ammette alcuna soluzione.

Propagazione di informazioni attraverso i vincoli

Finora il nostro algoritmo di ricerca ha esaminato i vincoli su una variabile solo nel momento in cui questa è selezionata da SCEGLI-VARIABILE-NON-ASSEGNAZIONATA. Ma possiamo ridurre drasticamente lo spazio di ricerca considerando alcuni vincoli in un momento precedente della ricerca, o addirittura prima del suo stesso inizio.

Verifica in avanti

verifica in avanti

Un modo di sfruttare meglio i vincoli durante la ricerca è la cosiddetta **verifica in avanti** (*forward checking*). Ogni volta che una variabile X è assegnata, il processo di verifica in avanti guarda ogni variabile non assegnata Y collegata a X da un vincolo e cancella dal dominio di Y ogni valore non consistente con quello scelto per X . La Figura 5.6 mostra il progresso di una ricerca con coloratura di mappa che sfrutta la verifica in avanti. Ci sono due punti da notare in quest'esempio. Prima di tutto, notate che dopo aver assegnato $WA = \text{rosso}$ e $Q = \text{verde}$, i domini di NT e SA sono ridotti a un solo valore; abbiamo eliminato completamente la ramificazione su queste variabili propagando informazione da WA a Q . L'euristica MRV, "compagna naturale" della verifica in avanti, selezionerebbe automaticamente come passi successivi SA e NT . In effetti, potremmo considerare la verifica in avanti come un modo efficiente di calcolare in modo incrementale le informazioni utilizzate dall'euristica MRV per svolgere il proprio lavoro. Un secondo aspetto da notare è che, dopo l'assegnamento $V = \text{blu}$, il dominio di SA è vuoto. Ne consegue che la verifica in avanti ha rilevato che l'assegnamento parziale $\{WA = \text{rosso}, Q = \text{verde}, V = \text{blu}\}$ è inconsistente con i vincoli del problema, e l'algoritmo potrà quindi tornare immediatamente indietro con il backtracking.

	WA	NT	Q	NSW	V	SA	T
domini iniziali	R V B	R V B	R V B	R V B	R V B	R V B	R V B
dopo $WA=\text{rosso}$	(R)	V B	R V B	R V B	R V B	V B	R V B
dopo $Q=\text{verde}$	(R)	B	(V)	R B	R V B	B	R V B
dopo $V=\text{blu}$	(R)	B	(V)	R		(B)	R V B

Figura 5.6 Il progresso di una ricerca con coloratura di mappa che sfrutta la verifica in avanti. Prima di tutto si assegna $WA = \text{rosso}$; fatto questo, la verifica in avanti cancella il colore *rosso* dai domini delle variabili adiacenti NT e SA . Dopo $Q = \text{verde}$, tale colore è rimosso dai domini di NT , SA e NSW . Dopo $V = \text{blu}$, anche *blu* è cancellato dai domini di NSW e SA , dimodoché SA non ha più valori legali.

Propagazione dei vincoli

La verifica in avanti rileva molte inconsistenze, ma non tutte. Considerate ad esempio la terza riga della Figura 5.6. Quando WA è *rosso* e Q è *verde*, sia NT che SA sono obbligati a essere blu; però sono adiacenti, quindi non possono avere lo stesso valore. La verifica in avanti non si accorge di questa inconsistenza perché non guarda abbastanza avanti. **Propagazione dei vincoli** è il termine generale per il passaggio di informazioni alle altre variabili riguardanti le conseguenze dei vincoli esistenti su una di esse; in questo caso dobbiamo propagare da WA e Q a NT e SA (cosa che veniva già fatta dalla verifica in avanti) e poi ancora sul vincolo tra NT e SA per rilevare finalmente l'inconsistenza. Tutto questo vogliamo farlo velocemente: se spendessimo più tempo propagando vincoli che eseguendo la ricerca equivalente, la cosa non avrebbe molto senso.

Il concetto di **consistenza d'arco** fornisce un metodo veloce per propagare vincoli, sensibilmente più potente della semplice verifica in vanti. Qui “arco” si riferisce a un lato *orientato* del grafo dei vincoli, come quello che va da SA a NSW . Dati i domini correnti di SA e NSW , l'arco è consistente se, per *ogni* valore x di SA , c'è *qualche* valore y di NSW che è consistente con x . Nella terza riga della Figura 5.6, i domini correnti di SA e NSW sono rispettivamente $\{blu\}$ e $\{rosso, blu\}$. Per $SA = blu$ c'è un assegnamento consistente di NSW , e precisamente $NSW = rosso$; quindi l'arco da SA a NSW è consistente. D'altra parte, l'arco inverso da NSW a SA non è consistente: all'assegnamento $NSW = blu$ non corrisponde alcun assegnamento consistente di SA . L'arco può essere reso consistente cancellando il valore blu dal dominio di NSW .

Possiamo applicare questo concetto di consistenza all'arco che va da SA a NT nella stessa fase del processo di ricerca. La terza riga della tabella nella Figura 5.6 mostra che entrambe le variabili hanno il dominio $\{blu\}$. Ne consegue che *blu* dev'essere cancellato dal dominio di SA , lasciandolo così vuoto. L'applicazione della consistenza d'arco ha quindi permesso di rilevare anticipatamente un'inconsistenza che non è stata rilevata dalla semplice verifica in avanti.

Il controllo della consistenza d'arco può essere applicato sia prima dell'inizio della ricerca, come preprocessing, sia come passo di propagazione (allo stesso modo della verifica in avanti) dopo ogni assegnamento di variabile: in quest'ultimo caso l'algoritmo viene talvolta chiamato MAC, dall'inglese *Maintaining Arc Consistency*. In ogni caso il processo dev'essere applicato *ripetutamente* finché non sono state eliminate tutte le inconsistenze, perché ogni volta che un valore viene cancellato dal dominio di una variabile per rimuovere un'inconsistenza, un'altra potrebbe sorgere negli archi che puntano a quella variabile. L'algoritmo completo per il controllo della consistenza d'arco, chiamato AC-3, utilizza una coda per tener traccia degli archi che devono ancora essere controllati (v. Figura 5.7). Ogni arco (X_i, X_j) viene rimosso a turno dalla coda e controllato; se vengono cancellati dei valori dal dominio di X_i , allora ogni arco (X_k, X_i) che punta a X_i dev'essere inserito ancora nella coda per essere ricontrollato. La complessità di questo processo può essere analizzata

propagazione dei
vincoli

consistenza d'arco

function AC-3(*csp*) **returns** il CSP, possibilmente con domini ridotti

inputs: *csp*, un CSP binario con variabili $\{X_1, X_2, \dots, X_n\}$

variabili locali: *queue*, una coda di archi, inizialmente tutti quelli in *csp*

while *queue* non è vuota **do**

$(X_i, X_j) \leftarrow \text{RIMUOVI-PRIMO}(\text{queue})$

if RIMUOVI-VALORI-INCONSISTENTI(X_i, X_j) **then**

for each X_k in ADIACENTI[X_i] **do**

 aggiungi (X_k, X_i) alla *queue*

function RIMUOVI-VALORI-INCONSISTENTI(X_i, X_j) **returns** true se e solo se viene rimosso un valore

removed $\leftarrow \text{false}$

for each x in DOMINIO[X_i] **do**

if nessun valore y in DOMINIO [X_j] permette a (x, y) di soddisfare il vincolo tra X_i e X_j

then rimuovi x da DOMINIO[X_i]; **removed** $\leftarrow \text{true}$

return **removed**

Figura 5.7 L'algoritmo AC-3 per il controllo della consistenza d'arco. Dopo l'applicazione di AC-3 ogni arco è arco-consistente, oppure qualche variabile avrà un dominio vuoto, il che significa che il CSP non può essere reso arco-consistente, e quindi non è possibile trovare una soluzione. Il nome "AC-3" fu usato dall'inventore dell'algoritmo (Mackworth, 1977) perché è la terza versione presentata all'interno dell'articolo.

come segue: un CSP binario ha al massimo $O(n^2)$ archi; ognuno di essi può essere inserito nella coda solo d volte, perché ci sono al massimo d valori da cancellare; il controllo di consistenza su un arco può essere fatto in un tempo $O(d^2)$; ne consegue che nel caso pessimo il tempo totale è $O(n^2d^3)$. Benché questo sia decisamente più costoso della semplice verifica in avanti, di solito ne vale la pena.¹

Dato che i CSP includono 3SAT come caso speciale, non ci dovremmo aspettare di trovare un algoritmo che può decidere in tempo polinomiale se un dato CSP è consistente. Ne dobbiamo dedurre che la verifica della consistenza d'arco non rileva tutte le possibili inconsistenze. Ad esempio, nella Figura 5.1, l'assegnamento parziale $\{\text{WA} = \text{rosso}, \text{NSW} = \text{rosso}\}$ è inconsistente, ma AC-3 non lo rileverà. Forme più potenti di propagazione possono essere definite usando il concetto di k -

k-consistenza

¹ L'algoritmo AC-4, di Mohr e Henderson (1986), ha una complessità temporale $O(n^2d^2)$. V. Esercizio 5.10.

consistenza. Un CSP è k -consistente se, per ogni insieme di $k-1$ variabili e ogni loro assegnamento consistente, è sempre possibile assegnare un valore consistente a ogni k -esima variabile. Ad esempio, 1-consistenza significa che ogni singola variabile è consistente in sé; questa prende anche il nome di **consistenza di nodo**. La 2-consistenza corrisponde alla consistenza d'arco. La 3-consistenza significa che ogni coppia di variabili adiacenti può sempre essere estesa includendo una terza variabile adiacente; questa si chiama anche **consistenza di cammino**.

Un grafo è **fortemente k -consistente** se è k -consistente e anche $(k-1)$ -consistente, $(k-2)$ -consistente, ... fino a 1-consistente. Ora supponete di avere un problema CSP con n nodi e di renderlo fortemente n -consistente (come dire, fortemente k -consistente con $k = n$). Allora è possibile risolvere quel problema senza ricorrere al backtracking. Per prima cosa scegliamo un valore consistente per X_1 . Dato che il grafo è 2-consistente allora saremo certamente in grado di scegliere un valore per X_2 , ma anche per X_3 vale lo stesso discorso perché il grafo è 3-consistente, e così via: per ogni variabile X_i dobbiamo solo cercare nei d valori del dominio per trovare un valore consistente con X_1, \dots, X_{i-1} . Questo garantisce di trovare una soluzione in un tempo $O(nd)$. Naturalmente, non si può ottenere un simile risultato gratis: ogni algoritmo che stabilisce la n -consistenza, nel caso pessimo, avrà una complessità esponenziale in n .

Ci sono molte scelte intermedie tra i due casi estremi della n -consistenza e della consistenza d'arco: i controlli più forti prenderanno più tempo, ma saranno più efficaci nel ridurre il fattore di ramificazione e rilevare gli assegnamenti parziali inconsistenti. È possibile calcolare il più piccolo valore k tale che la k -consistenza di un problema assicuri che il problema stesso può essere risolto senza backtracking (v. Paragrafo 5.4), ma spesso non conviene. Nella pratica, determinare il livello appropriato di verifica della consistenza è in gran parte una scienza empirica.

consistenza di nodo

consistenza di cammino
fortemente k -consistente

Gestire vincoli speciali

Alcuni tipi di vincoli si verificano frequentemente nei problemi reali, e possono essere gestiti con algoritmi speciali più efficienti dei metodi generici descritti fin qui. Ad esempio, il vincolo *Tuttediverse* richiede che le variabili specificate abbiano tutti valori diversi (come nel problema di criptoaritmetica). Per il vincolo *Tuttediverse* un semplice metodo di rilevazione delle inconsistenze può essere il seguente: se il vincolo coinvolge m variabili, e se queste hanno n possibili valori distinti, e $m > n$, allora il vincolo non può essere soddisfatto.

Questa considerazione porta al semplice algoritmo: si rimuove prima di tutto ogni variabile coinvolta nel vincolo che ha un dominio con un solo valore, cancellando tale valore dai domini di tutte le altre variabili. Si ripete finché ci sono variabili con domini a un solo valore: se in qualsiasi momento un dominio rimane vuoto o ci sono più variabili che valori rimasti, è stata scoperta un'inconsistenza.

Possiamo applicare questo metodo per rilevare l'inconsistenza dell'assegnamento parziale $\{WA = \text{rosso}, NSW = \text{rosso}\}$ nella Figura 5.1. Notate che le variabili SA , NT e Q sono effettivamente collegate dal vincolo *Tuttediverse* dato che ogni

coppia dev'essere di un colore diverso. Dopo aver applicato AC-3 all'assegnamento parziale, il dominio di ogni variabile è ridotto a $\{\text{verde}, \text{blu}\}$. Abbiamo tre variabili e due soli colori, quindi il vincolo *Tuttediverse* è violato: una semplice procedura applicata a un vincolo di ordine superiore può rivelarsi più efficace del calcolo della consistenza d'arco a un insieme equivalente di vincoli binari.

vincolo sulle risorse

Tra quelli di ordine superiore, il più importante è forse il **vincolo sulle risorse**, talvolta chiamato *atmost* ("al massimo"). Ad esempio, supponiamo che PA_1, \dots, PA_4 indichino il numero di persone assegnate a quattro diverse attività. Il vincolo che in totale non siano assegnate più di 10 persone si scrive *atmost* (10, PA_1, PA_2, PA_3, PA_4). Per rilevare un'eventuale inconsistenza basta controllare la somma dei valori minimi dei domini correnti; se ad esempio ogni variabile ha il dominio $\{3, 4, 5, 6\}$ allora il vincolo *atmost* non può essere soddisfatto. Possiamo assicurare la consistenza cancellando il valore massimo di un dominio se non è consistente con i valori minimi degli altri: così, se ogni variabile del nostro esempio ha il dominio $\{2, 3, 4, 5, 6\}$, i valori 5 e 6 possono essere eliminati dai domini di tutte le variabili.

Per grandi problemi a risorse limitate e valori interi, come i problemi logistici che richiedono di spostare migliaia di persone con centinaia di veicoli, normalmente è impossibile rappresentare il dominio di ogni variabile come un grande insieme di interi per poi ridurlo gradualmente con i metodi di verifica della consistenza. In questi casi i domini sono rappresentati invece dagli estremi superiore e inferiore dell'intervallo di valori consentiti e sono gestiti mediante la propagazione degli estremi stessi. Supponiamo ad esempio che ci siano due voli, 271 e 272, i cui aerei abbiano rispettivamente una capacità di 165 e 385 passeggeri. I domini iniziali saranno allora

$$Volo271 \in [0, 165] \text{ e } Volo272 \in [0, 385].$$

Supponiamo ora di avere un vincolo aggiuntivo, e cioè che i due voli insieme debbano portare esattamente 420 passeggeri: $Volo271 + Volo272 \in [420, 420]$. Propagando i vincoli sugli estremi, possiamo ridurre i domini a

$$Volo271 \in [35, 165] \text{ e } Volo272 \in [255, 385].$$

propagazione degli estremi

Diciamo che un CSP ha gli estremi consistenti se per ogni variabile X , e per entrambi i suoi estremi inferiore e superiore, esiste qualche valore di Y che soddisfa i vincoli tra X e Y , per ogni variabile Y . Questo tipo di **propagazione degli estremi** è ampiamente usata nella pratica.

backtracking cronologico

Backtracking intelligente: guardarsi indietro

L'algoritmo RICERCA-BACKTRACKING della Figura 5.3 ha una politica molto semplice riguardo a cosa fare quando un ramo della ricerca fallisce: tornare indietro alla variabile precedente e provare a usare un valore diverso. Questo si chiama **backtracking cronologico**, perché viene rivisitato il punto decisionale *più recente*. Come vedremo in questo sottoparagrafo, esistono strategie molto più efficienti.

Considerate quello che succede nella Figura 5.1 quando applichiamo un backtracking semplice con ordinamento fisso delle variabili Q, NSW, V, T, SA, WA, NT . Supponiamo di aver generato l'assegnamento parziale $\{Q = \text{rosso}, NSW = \text{verde}, V = \text{blu}, T = \text{rosso}\}$. Quando prendiamo la variabile successiva, SA , appuriamo che ogni possibile valore viola un vincolo. Allora torniamo indietro in T e assegniamo un colore diverso alla Tasmania! Palesemente questa è un'azione alquanto scioccata: cambiare il colore della Tasmania non potrà risolvere un problema nell'Australia del Sud.

Un approccio più intelligente al backtracking è tornare indietro fino a una delle variabili che *ha causato il fallimento*. L'insieme di queste variabili è chiamato **insieme dei conflitti**; in questo caso l'insieme dei conflitti per SA è $\{Q, NSW, V\}$. In generale, l'insieme dei conflitti per una variabile X è l'insieme delle variabili precedentemente assegnate che sono collegate a X da vincoli. Il metodo del **backjumping** (letteralmente, “salto all'indietro”) torna indietro fino alla variabile *più recente* nell'insieme dei conflitti; in questo caso saltando la Tasmania e cercando un altro valore per V . Si può facilmente implementare tutto ciò modificando BACKTRACKING-SEARCH in modo che memorizzi l'insieme dei conflitti mentre cerca un valore legale da assegnare. Se non trova alcun valore legale, insieme all'indicazione del fallimento dovrà restituire l'elemento più recente dell'insieme dei conflitti.

I lettori più attenti avranno già notato che la verifica in avanti può fornire l'insieme dei conflitti senza lavoro aggiuntivo: ogni volta che la verifica in avanti causata da un assegnamento a X fa sì che venga cancellato un valore dal dominio di Y , X dev'essere aggiunto all'insieme dei conflitti di Y . Inoltre, ogni volta che l'ultimo valore viene cancellato dal dominio di Y , tutte le variabili dell'insieme dei conflitti di Y vanno aggiunte all'insieme dei conflitti di X . In questo modo, quando arriviamo a considerare Y , sappiamo immediatamente fin dove ritornare in caso di backtracking.

I lettori *veramente* attenti avranno notato qualcosa di strano: il backjumping si verifica quando ogni valore in un dominio configge con l'assegnamento corrente; ma la verifica in avanti rileva questo evento e impedisce alla ricerca di raggiungere quel nodo in ogni caso! In effetti, si può dimostrare che *ogni ramo dell'albero potato dal backjumping è parimenti potato dalla verifica in avanti*. Di conseguenza, il semplice backjumping è ridondante in una ricerca con verifica in avanti, e ancor di più quando si usano verifiche di consistenza più potenti come MAC.

Nonostante quest'osservazione, il backjumping rimane una buona idea: vale la pena di basare il processo di backtracking sulle ragioni del fallimento. Il backjumping rileva un fallimento quando il dominio di una variabile diventa vuoto, ma in molti casi un ramo è “condannato” molto prima che questo si verifichi. Consideriamo ancora l'assegnamento parziale inconsistente $\{WA = \text{rosso}, NSW = \text{rosso}\}$. Supponiamo di provare come passo successivo $T = \text{rosso}$ e poi assegnare valori alle variabili NT, Q, V, SA . Sappiamo che per queste ultime quattro

insieme dei conflitti

backjumping



non c'è un possibile assegnamento, per cui a un certo punto esauriremo i valori per NT . A questo punto, la domanda è: dove ritornare con il backtracking? Il backjumping non può funzionare, perché NT ha effettivamente valori consistenti con le variabili assegnate in precedenza: non esiste un insieme dei conflitti completo che ha causato il suo fallimento. Sappiamo comunque che le quattro variabili NT , Q , V e SA , prese insieme, falliscono a causa di un insieme delle variabili precedenti, che devono essere quelle che confliggono direttamente con queste quattro. Questo ci porta a una comprensione più profonda dell'insieme dei conflitti per una variabile come NT : è l'insieme delle variabili precedenti che ha fatto sì che NT , insieme alle variabili successive, non avesse alcuna soluzione consistente. In questo caso l'insieme è composto da WA e NSW , quindi l'algoritmo dovrebbe ritornare a NSW saltando la Tasmania. Un algoritmo di backjumping che sfrutta insiemi dei conflitti definiti in questo modo è chiamato **backjumping guidato dai conflitti**.

Ora dobbiamo spiegare come calcolare questi nuovi insiemi dei conflitti. In effetti, il metodo è piuttosto semplice. Il fallimento "terminale" di un ramo della ricerca si verifica sempre perché il dominio di una variabile diventa vuoto; quella variabile ha un insieme dei conflitti standard. Nel nostro esempio, SA fallisce, e il suo insieme dei conflitti è (poniamo) $\{WA, NT, Q\}$. Saltiamo indietro fino a Q , che *assorbe* l'insieme dei conflitti di SA (tranne se stesso, naturalmente) integrandolo nel suo proprio insieme dei conflitti diretti, che è $\{NT, NSW\}$; il nuovo insieme è $\{WA, NT, NSW\}$. Questo significa che non c'è soluzione da Q in avanti, dati i precedenti assegnamenti a $\{WA, NT, NSW\}$. Di conseguenza, torniamo indietro con il backtracking fino a NT , il più recente tra questi. NT assorbe $\{WA, NT, NSW\} - \{NT\}$ nel suo insieme dei conflitti diretti $\{WA\}$, e il risultato è $\{WA, NSW\}$. Adesso l'algoritmo salta indietro fino a NSW , come ci saremmo augurati. Per riassumere: sia X_j la variabile corrente e $conf(X_j)$ il suo insieme dei conflitti. Se ogni possibile valore di X_j fallisce, si salti indietro (backjump) alla variabile più recente X_i in $conf(X_j)$, e sia

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_j\}.$$

Il backjumping guidato dai conflitti ci riporta indietro fino al punto giusto dell'albero di ricerca, ma non ci impedisce di commettere gli stessi errori in un altro ramo dell'albero. L'apprendimento dei vincoli modifica a tutti gli effetti il CSP aggiungendo un nuovo vincolo che viene dedotto da questo tipo di conflitti.

5.3 Ricerca locale per problemi di soddisfacimento di vincoli

Gli algoritmi di ricerca locale (v. Paragrafo 4.3) si rivelano molto efficaci per la soluzione di svariate tipologie di CSP. La formulazione usata è quella a stato completo: lo stato iniziale assegna un valore a ogni variabile e normalmente la funzione successore cambia il valore di una variabile per volta. Ad esempio, nel problema delle 8 regine, lo stato iniziale potrebbe essere una configurazione casuale delle 8 regine nelle 8 colonne e la funzione successore prenderà una regina e valuterà i suoi spostamenti nella stessa colonna. Un'altra possibilità potrebbe essere quella di cominciare con le 8 regine, una per colonna in una permutazione delle 8 traverse (righe), e di generare un successore scambiando le traverse di due regine.² In effetti abbiamo già visto un esempio d'uso di ricerca locale per risolvere un CSP: l'applicazione dell'hill-climbing al problema a n regine (v. pag. 148). Un altro esempio potrebbe essere l'applicazione di WALKSAT (v. pag. 286) per risolvere problemi di soddisficiabilità, che sono un caso speciale di CSP.

Scegliendo un nuovo valore per una variabile, l'euristica più ovvia è scegliere il valore che risulta nel numero minimo di conflitti con le altre variabili: questa si chiama appunto euristica **min-conflicts**. L'algoritmo è riportato nella Figura 5.8; la sua applicazione al problema delle 8 regine è illustrata nella Figura 5.9 e quantificata nella Figura 5.5.

L'euristica min-conflicts è sorprendentemente efficace per molti CSP, in particolare quando le viene fornito uno stato iniziale ragionevole. Le sue prestazioni sono riportate nell'ultima colonna della Figura 5.5. Ciò che è davvero stupefacente, nel problema a n regine, è che se non si conta il piazzamento iniziale dei pezzi il tempo di esecuzione di min-conflicts è quasi *indipendente dalle dimensioni del problema*. È capace di risolvere anche il problema con *un milione* di regine in una media di 50 passi (dopo l'assegnamento iniziale). Quest'osservazione notevole è stata lo sprone che negli anni '90 ha portato a una grande mole di studi sulla ricerca locale e la distinzione tra problemi facili e difficili, che tratteremo nel Capitolo 7. A grandi linee, si può dire che il problema delle n regine è facile per la ricerca locale perché le soluzioni sono distribuite densamente nello spazio degli stati. Min-conflicts peraltro funziona bene anche su problemi difficili: è stato usato per pianificare le osservazioni del telescopio Hubble, riducendo il tempo richiesto per organizzare una settimana di osservazioni da tre settimane (!) a circa 10 minuti.

min-conflicts

² La ricerca locale può essere facilmente estesa a CSP con funzioni obiettivo. In tal caso, per ottimizzare la funzione obiettivo si possono usare tutte le tecniche di hill-climbing e simulated annealing già viste.

function MIN-CONFLICTS(*csp*, *max_passi*) **returns** una soluzione, o il fallimento

inputs: *csp*, un problema di soddisfacimento di vincoli

max_passi, il numero di passi consentiti prima di abbandonare

current \leftarrow un assegnamento completo iniziale per *csp*

for *i* = 1 to *max_passi* **do**

if *current* è una soluzione di *csp* **then return** *current*

var \leftarrow una variabile in conflitto, scelta a caso in VARIABILI[*csp*]

valore \leftarrow il valore *v* di *var* che minimizza CONFLITTI(*var*, *v*, *current*, *csp*)

scrivi *var* = *valore* dentro *current*

return fallimento

Figura 5.8 L'algoritmo MIN-CONFLICTS per la risoluzione dei CSP mediante ricerca locale. Lo stato iniziale può essere scelto a caso o con un processo di assegnamento greedy che sceglie il valore di minimo conflitto per ogni variabile. La funzione CONFLITTI, dato l'assegnamento corrente, conta il numero di vincoli violati da un particolare valore.

Un altro vantaggio della ricerca locale è che può essere usata in un ambiente online, in cui il problema stesso può cambiare: ciò è particolarmente importante nei problemi di scheduling. Una linea aerea potrebbe dover gestire in una settimana migliaia di voli e decine di migliaia di assegnamenti di personale, e il verificarsi di cattivo tempo in un solo aeroporto può rendere inattuabile il piano originario.

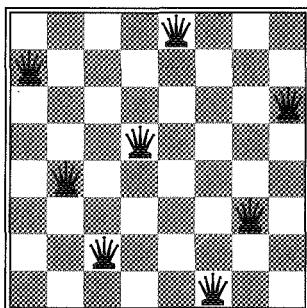
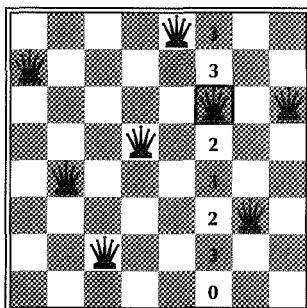
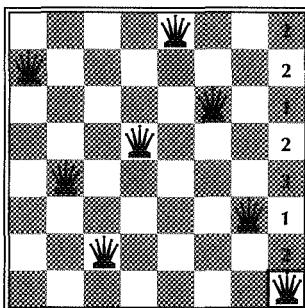


Figura 5.9 Una soluzione in due passi per un problema delle 8 regine, ottenuta con l'algoritmo min-conflicts. A ogni passo una regina viene scelta e riposizionata nella sua colonna. Il numero di conflitti (in questo caso, quello delle regine attaccanti) è indicato in ogni casella. L'algoritmo sposta la regina nella casa corrispondente al numero minimo di conflitti, risolvendo casualmente le situazioni di parità.

È fondamentale essere in grado di costruire un piano alternativo con il numero minimo di cambiamenti. Questo può essere fatto facilmente da un algoritmo di ricerca locale, fornendo come stato iniziale il piano da modificare. Una ricerca con backtracking che prendesse in considerazione il nuovo insieme di vincoli richiederebbe molto più tempo e potrebbe trovare una soluzione troppo diversa dal piano originario.

5.4 La struttura dei problemi

In questo paragrafo esamineremo i modi in cui si può sfruttare la *struttura* del problema, rappresentata dal grafo dei vincoli, per trovare rapidamente le soluzioni. La maggior parte degli approcci presentati sono molto generali, e applicabili anche ad altri problemi oltre i CSP, come ad esempio il ragionamento probabilistico. L'unico modo in cui possiamo sperare di gestire la complessità del mondo reale è scomporlo in molti sottoproblemi. Guardando ancora una volta la Figura 5.1(b) con l'intento di identificare la struttura del problema, un fatto salta agli occhi: la Tasmania non è collegata al continente.³ Intuitivamente, appare chiaro che la coloratura della Tasmania e quella del continente principale sono **sottoproblemi indipendenti** – qualsiasi soluzione per il continente, unita a una qualsiasi soluzione per la Tasmania, darà luogo a una soluzione globale per l'intera mappa. L'indipendenza si può appurare semplicemente cercando **componenti collegati** sul grafo dei vincoli. Ogni componente corrisponde a un sottoproblema CSP_i . Se l'assegnamento S_i è una soluzione di CSP_i , allora $\cup_i S_i$ è una soluzione di $\cup_i CSP_i$. Perché questo fatto è importante? Considerate quanto segue: supponiamo che ogni CSP_i abbia c variabili prese da un totale di n variabili, dove c è una costante. Allora ci sono n/c sottoproblemi, ognuno dei quali richiede al massimo d^c “lavoro” per essere risolto. Quindi il lavoro totale da svolgere è $O(d^c n/c)$, che cresce *linearmente* con n ; senza la scomposizione il lavoro sarebbe $O(d^n)$, che cresce esponenzialmente. Facciamo un esempio più concreto: suddividere un CSP booleano con $n = 80$ in quattro sottoproblemi con $c = 20$ riduce il tempo necessario per trovare una soluzione nel caso pessimo dalla durata della vita dell'universo a meno di un secondo.

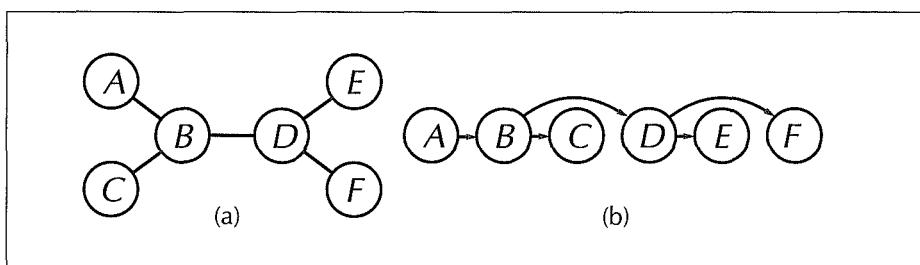
I sottoproblemi completamente indipendenti quindi ci piacciono moltissimo; purtroppo però sono anche rari. Nella maggior parte dei casi, i sottoproblemi di un CSP sono collegati. Il caso più semplice si ha quando il grafo dei vincoli

sottoproblemi
indipendenti

componenti collegati

³ Un cartografo molto professionale, o un nativo della Tasmania particolarmente patriottico, potrebbero obiettare che la Tasmania non dovrebbe avere lo stesso colore della regione più vicina sulla terraferma, proprio per evitare l'impressione che *potrebbe* far parte di quello stato.

Figura 5.10
 (a) Il grafo dei vincoli di un CSP strutturato ad albero. (b) Un ordinamento lineare delle variabili consistente con l'albero, con A come radice.



forma un **albero**: due variabili qualsiasi sono collegate al più da un cammino. La Figura 5.10(a) mostra un esempio schematico.⁴ Ora mostreremo che *ogni CSP strutturato come un albero può essere risolto in un tempo che cresce linearmente con il numero delle variabili*. L'algoritmo è composto dai seguenti passi:

1. scegliete una qualsiasi variabile come radice dell'albero e ordinate le altre dalla radice alle foglie in modo che il padre di un nodo lo preceda nell'ordinamento (v. Figura 5.10(b)). Etichettate le variabili X_1, \dots, X_n nell'ordine. Ora ogni variabile eccetto la radice ha esattamente una variabile padre;
2. per j che va da n a 2 a ritroso, verificate la consistenza dell'arco (X_i, X_j) , dove X_i è il nodo padre di X_j , rimuovendo alcuni valori da $\text{DOMINIO}[X_i]$ se necessario;
3. per j che va da 1 a n , assegnate a X_j un valore consistente con quello assegnato a X_i , dove X_i è il nodo padre di X_j .

Ci sono due punti da notare con particolare attenzione: prima di tutto, dopo il passo 2 il CSP ha gli archi tutti consistenti in una direzione, per cui l'assegnamento di valori del passo 3 non richiede backtracking (v. la discussione sulla k -consistenza a pagina 191). In secondo luogo, verificando la consistenza degli archi in ordine inverso nel passo 2, l'algoritmo si assicura che ogni valore cancellato non metta a rischio la consistenza degli archi già processati. L'algoritmo completo ha una complessità temporale $O(nd^2)$.

Ora che abbiamo un algoritmo efficiente per gli alberi, possiamo considerare se è possibile ridurre in qualche modo ad alberi dei grafici di vincoli generici. Per far questo ci sono due metodi principali, uno basato sulla rimozione di nodi e l'altro sulla loro fusione.

Il primo approccio prevede che si assegnino dei valori ad alcune variabili in modo che quelle rimanenti formino un albero. Considerate ancora il grafo dei vincoli per la mappa dell'Australia, mostrato ancora nella Figura 5.11(a). Se potessi-

⁴ Purtroppo ben poche regioni del mondo hanno una mappa strutturata ad albero, con la possibile eccezione del Sulawesi.

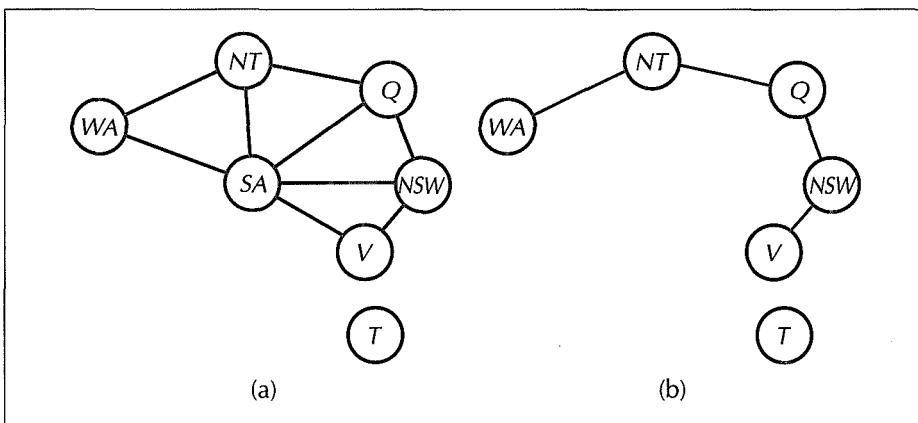


Figura 5.11
 (a) Il grafo dei vincoli originale della Figura 5.1.
 (b) Il grafo dei vincoli dopo la rimozione di *SA*.

mo cancellare l’Australia del Sud (*SA*) il grafo diventerebbe un albero, come si vede in (b). Fortunatamente possiamo farlo (sul grafo, non in Oceania) fissando un valore per *SA* e cancellando dai domini delle altre variabili tutti i valori che sono inconsistenti con quello.

Fatto questo, ogni soluzione per il CSP dopo la rimozione di *SA* e dei suoi vincoli sarà consistente col valore prescelto per *SA* (questo funziona nei CSP binari; la situazione è più complessa se entrano in gioco dei vincoli di ordine superiore). Possiamo quindi risolvere l’albero rimanente con l’algoritmo fornito qui sopra e risolvere così l’intero problema. Naturalmente, nel caso generale il valore scelto per *SA* potrebbe essere sbagliato (cosa che non può succedere con la coloratura di una mappa), per cui potremmo dover provare anche gli altri. L’algoritmo generale risultante è il seguente.

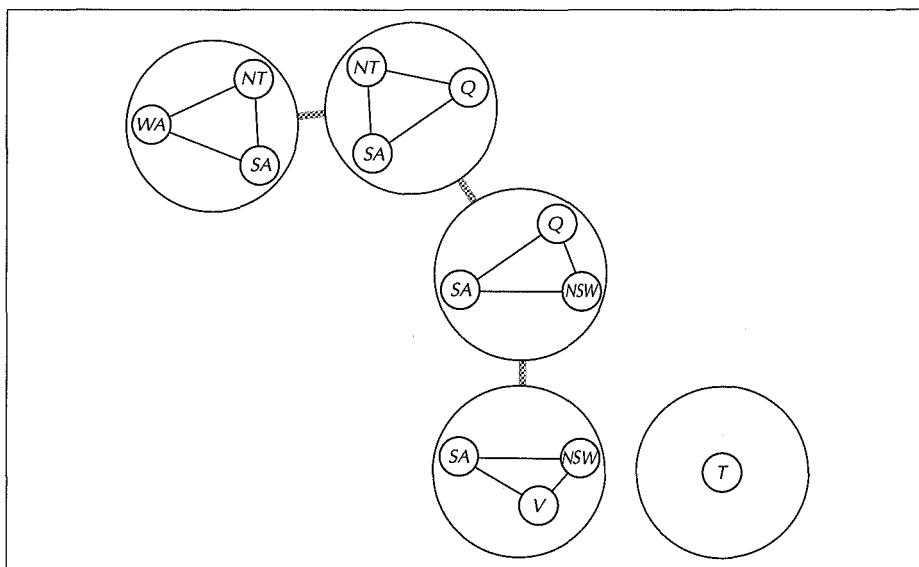
1. Scegliete un sottoinsieme *S* di *VARIABILI[csp]* tale che il grafo dei vincoli diventi un albero dopo la rimozione di *S*. *S* prende il nome di **insieme di taglio dei cicli** (*cycle cutset*).
2. Per ogni possibile assegnamento delle variabili in *S* che soddisfa tutti i vincoli su *S*,
 - (a) rimuovete dal dominio delle variabili rimanenti tutti i valori non consistenti con gli assegnamenti in *S*, e
 - (b) se il CSP risultante ha una soluzione, restituitela insieme all’assegnamento per *S*.

insieme di taglio dei cicli

Se l’insieme di taglio dei cicli ha dimensioni *c*, il tempo di esecuzione totale è $O(d^c \cdot (n - c)d^2)$. Se il grafo è “quasi un albero” *c* sarà piccolo, e il risparmio di tempo rispetto al backtracking risulterà enorme. Nel caso pessimo, però, *c* può ar-

Figura 5.12

Una scomposizione ad albero del grafo di vincoli della Figura 5.11(a).



rivare anche a $(n - 2)$. Trovare il *più piccolo* insieme di taglio dei cicli è un problema NP-difficile, ma si conoscono diversi algoritmi per farlo in modo approssimato ed efficiente. Quest’approccio algoritmico è generalmente chiamato **condizionamento con insieme di taglio** (*cutset conditioning*); lo ritroveremo nel Capitolo 14 (nel 2° vol.), dove sarà usato per ragionare sulle probabilità.

Il secondo approccio è basato sulla costruzione di una **scomposizione ad albero** (*tree decomposition*) del grafo di vincoli in un insieme di sottoproblemi collegati. Ogni sottoproblema viene risolto indipendentemente e le soluzioni sono poi combinate insieme: come la maggior parte degli algoritmi *divide et impera*, anche questo funziona bene se nessun sottoproblema è troppo grande. La Figura 5.12 mostra una scomposizione ad albero del problema della coloratura in cinque sottoproblemi. Una scomposizione deve soddisfare i seguenti tre requisiti:

- ◆ ogni variabile del problema originale deve comparire in almeno uno dei sottoproblemi;
- ◆ se due variabili sono collegate da un vincolo nel problema originale, devono comparire insieme (con il vincolo) in almeno uno dei sottoproblemi;
- ◆ se una variabile compare in due sottoproblemi sull’albero, deve essere presente anche in tutti i sottoproblemi che compongono il cammino che li collega.

condizionamento con
insieme di taglio

scomposizione ad
albero

Le prime due condizioni assicurano che tutte le variabili e i vincoli siano presenti nella scomposizione. La terza sembra riguardare un aspetto tecnico, ma riflette semplicemente il fatto che ogni variabile deve avere lo stesso valore in ognuno dei sottoproblemi in cui appare; i collegamenti tra i sottoproblemi sull'albero assicurano il rispetto di questo vincolo. Ad esempio, SA appare in tutti i quattro sottoproblemi collegati nella Figura 5.12. Potete confrontare quest'ultima con la Figura 5.11 per verificare che questa scomposizione è valida.

Ogni sottoproblema viene risolto in modo indipendente; se uno di essi non ha soluzione, sappiamo che non ce l'ha neppure il problema originale. Se risolviamo tutti i sottoproblemi, possiamo cercare di costruire una soluzione globale nel modo seguente. Prima di tutto consideriamo ogni sottoproblema come una “mega-variabile” il cui dominio è l’insieme di tutte le sue soluzioni. Ad esempio, il sottoproblema più a sinistra nella Figura 5.12 è un problema di coloratura di mappa con tre variabili e quindi avrà sei soluzioni; una sarà $\{WA = \text{rosso}, SA = \text{blu}, NT = \text{verde}\}$. A questo punto possiamo risolvere i vincoli collegando tra loro i sottoproblemi, utilizzando l’efficiente algoritmo che abbiamo visto poco fa. I vincoli tra sottoproblemi si limitano a insistere sul fatto che le rispettive soluzioni devono concordare sui valori delle variabili comuni. Ad esempio, data la soluzione $\{WA = \text{rosso}, SA = \text{blu}, NT = \text{verde}\}$ per il primo sottoproblema, l’unica soluzione consistente per quello successivo è $\{SA = \text{blu}, NT = \text{verde}, Q = \text{rosso}\}$.

Un grafo di vincoli ammette diverse scomposizioni ad albero; la scelta deve cercare di rendere i sottoproblemi più piccoli possibile. Data una scomposizione, la dimensione del sottoproblema più grande meno uno si chiama **larghezza d’albero** (*tree width*) della scomposizione stessa; la larghezza d’albero di un grafo è definita come la larghezza d’albero minima tra tutte le sue scomposizioni. Se un grafo ha larghezza d’albero w , e possediamo la scomposizione corrispondente, allora il problema può essere risolto in tempo $O(nd^{w+1})$. Si può quindi dire che *i CSP i cui grafi dei vincoli hanno larghezza d’albero limitata sono risolvibili in tempo polinomiale*. Sfortunatamente, trovare la scomposizione con larghezza d’albero minima è un problema NP-difficile, ma esistono metodi euristici che funzionano bene in pratica.

larghezza d’albero



5.5 Riepilogo

- ♦ **I problemi di soddisfacimento di vincoli** (spesso indicati con l’acronimo CSP) consistono in un insieme di variabili su cui sono imposti dei vincoli. Molti importanti problemi del mondo reale possono essere espressi sotto forma di CSP. La struttura di un CSP può essere rappresentata per mezzo del suo **grafo dei vincoli**.
- ♦ Per risolvere i CSP si usa comunemente la **ricerca con backtracking**, una forma particolare di ricerca in profondità.

- ◆ L'euristica **MRV** e quella **di grado** sono indipendenti dal dominio e possono essere usate durante la ricerca con backtracking per scegliere quale considerare tra le variabili rimanenti. Analogamente, l'euristica del **valore meno vincolante** aiuta a ordinare i valori di una variabile.
- ◆ L'algoritmo con backtracking può ridurre parecchio il fattore di ramificazione di un problema propagando le conseguenze degli assegnamenti parziali già costruiti. Per far questo, il metodo più semplice è la **verifica in avanti**. La **consistenza d'arco** è una tecnica più potente, ma può essere costosa da eseguire.
- ◆ Il backtracking si verifica quando non è più possibile assegnare un valore legale a una variabile. Il **backjumping guidato dai conflitti** torna direttamente all'origine del problema.
- ◆ La ricerca locale con l'euristica **min-conflicts** è stata applicata con grande successo ai problemi di soddisfacimento di vincoli.
- ◆ La complessità della soluzione di un CSP è fortemente dipendente dalla struttura del suo grafo dei vincoli. I problemi strutturati ad albero possono essere risolti in tempo lineare. Il **condizionamento con insieme di taglio** può ridurre un CSP generico in uno strutturato ad albero, ed è molto efficiente se si riesce a trovare un insieme di taglio piccolo. Le tecniche di **scomposizione ad albero** trasformano il CSP in un albero di sottoproblemi e sono efficienti se la **larghezza d'albero** del grafo dei vincoli è piccola.

Note storiche e bibliografiche

I primi lavori dedicati al soddisfacimento di vincoli si occupavano per lo più di vincoli numerici. Vincoli espressi sotto forma di equazioni con domini interi furono studiati dal matematico indiano Brahmagupta nel VII secolo; spesso sono chiamate **equazioni diofantine** dal matematico greco Diofanto (c. 200–284), che in effetti considerò il dominio dei razionali positivi. Metodi sistematici per la soluzione di equazioni lineari mediante eliminazione di variabili furono studiati da Gauss (1829); la soluzione dei vincoli di disegualanza lineare risalgono a Fourier (1827).

Anche i problemi di soddisfacimento di vincoli a dominio finito hanno una lunga storia. Ad esempio, la **coloratura di grafo** (di cui la coloratura di mappe è un caso speciale) è un vecchio problema matematico. Secondo Biggs et al. (1986), la congettura dei quattro colori (che ipotizza che ogni grafo piano possa essere colorato con quattro colori o meno) fu formulata per la prima volta da Francis Guthrie, uno studente di De Morgan, nel 1852. Il problema resistette a ogni tentativo di soluzione (anche se ne furono pubblicate diverse) fino alla prova definitiva, ottenuta con l'ausilio del computer, di Appel e Haken (1977).

Per tutta la storia dell'informatica sono state considerate classi specifiche di problemi di soddisfacimento di vincoli. Uno dei primi esempi più influenti fu il sistema SKETCHPAD (Sutherland, 1963), che risolveva vincoli geometrici su dia-

equazioni diofantine

coloratura di grafo

grammi e fu il precursore dei moderni programmi di disegno e CAD. L'identificazione dei CSP come classe *generale* è dovuta a Ugo Montanari (1974). La riduzione dei CSP di ordine superiore a CSP binari puri con l'aiuto di variabili ausiliarie (v. Esercizio 5.11) è stata effettuata per la prima volta nel XIX secolo dal logico Charles Sanders Peirce. Fu poi introdotta nella letteratura moderna dei CSP da Dechter (1990b) ed elaborata da Bacchus e van Beek (1998). I CSP che hanno preferenze nelle soluzioni sono studiati ampiamente nella letteratura dedicata all'ottimizzazione; v. Bistarelli et al. (1997) per una generalizzazione dell'infrastruttura CSP che permette di specificare preferenze. Ai problemi di ottimizzazione può essere applicato anche l'algoritmo della "bucket elimination" (Dechter, 1999).

L'applicazione al soddisfacimento di vincoli della ricerca con backtracking è dovuta a Bitner e Reingold (1975), anche se gli stessi autori fanno risalire l'algoritmo base al XIX secolo. Bitner e Reingold hanno anche introdotto l'euristica MRV, sotto il nome di euristica della *variabile più vincolata*. Brelaz (1979) ha usato l'euristica di grado per risolvere i "pareggi" dopo l'applicazione della MRV. L'algoritmo risultante, nonostante la sua semplicità, è ancora il metodo migliore per la k -coloratura di grafi arbitrari. Haralick e Elliot (1980) hanno proposto la *euristica del valore meno vincolante*.

I metodi di propagazione dei vincoli furono resi popolari dal successo di Waltz (1975) sui problemi di etichettatura di linee poliedrici nel campo della visione artificiale. Waltz ha mostrato che, in molti problemi, la propagazione elimina completamente il backtracking. Montanari (1974) ha introdotto il concetto di reti di vincoli e di propagazione mediante la consistenza dei cammini. Alan Mackworth (1977) ha proposto l'algoritmo AC-3 per assicurare la consistenza d'arco oltre all'idea generale di combinare il backtracking con la verifica di consistenza. AC-4, una versione più efficiente dell'algoritmo per la consistenza d'arco, fu sviluppato da Mohr e Henderson (1986). Subito dopo l'apparizione dell'articolo di Mackworth, i ricercatori hanno cominciato a sperimentare vari compromessi tra il costo delle verifiche di consistenza e i loro benefici in termini di riduzione dei tempi di ricerca. Haralick e Elliot (1980) hanno favorito l'algoritmo minimo di verifica in avanti descritto da McGregor (1979), mentre Gaschnig (1979) ha suggerito di verificare completamente la consistenza d'arco dopo ogni assegnamento di variabile; un algoritmo che in seguito fu chiamato MAC da Sabin e Freuder (1994). Quest'ultimo articolo fornisce prove abbastanza convincenti che, per i CSP più difficili, una piena verifica della consistenza d'arco risulta conveniente. Freuder (1978, 1982) ha investigato il concetto di k -consistenza e le sue relazioni con la complessità delle soluzioni dei CSP. Apt (1999) descrive un'infrastruttura algoritmica generica all'interno della quale si possono analizzare gli algoritmi di propagazione della consistenza.

Metodi speciali per la gestione di vincoli di ordine superiore sono stati sviluppati soprattutto nel contesto della **programmazione logica dei vincoli**. Marriott e Stuckey (1998) offrono un'eccellente panoramica della ricerca in quest'area.

Il vincolo *Tuttediverse (Alldiff)* è stato studiato da Regin (1994). I vincoli sugli estremi furono incorporati nella programmazione logica dei vincoli da Van Hentenryck et al. (1998).

Il metodo base del backjumping è dovuto a John Gaschnig (1977, 1979). Kondrak e van Beek (1997) hanno mostrato che quest'algoritmo è sostanzialmente sostituibile dalla verifica in avanti. Il backjumping guidato dai conflitti fu inventato da Prosser (1993). La forma più generale e potente di backtracking intelligente in effetti fu sviluppata molto presto da Stallman e Sussman (1977): la loro tecnica, il **backtracking guidato dalle dipendenze**, portò allo sviluppo dei sistemi di mantenimento della verità (Doyle, 1979), che discuteremo nel Paragrafo 10.8. Il collegamento tra le due aree è analizzato da de Kleer (1989).

Il lavoro di Stallman e Sussman ha anche introdotto l'idea della **memorizzazione dei vincoli**, in cui i risultati parziali della ricerca possono essere salvati e riutilizzati più tardi. L'idea fu integrata formalmente nella ricerca con backtracking da Dechter (1990a). Il **backmarking** (Gaschnig, 1979) è un metodo particolarmente semplice basato sulla memorizzazione di coppie di assegnamenti consistenti e inconsistenti per evitare di controllare nuovamente i vincoli. Il backmarking può essere combinato con il backjumping guidato dai conflitti; Kondrak e van Beek (1997) presentano un algoritmo ibrido che probabilmente è in grado di sostituire ognuno dei due metodi presi separatamente. Il metodo del **backtracking dinamico** (Ginsberg, 1993) conserva gli assegnamenti parziali di successo effettuati in sottoinsiemi di variabili considerate successivamente a quella che si sta scavalcando con il backtracking, nel caso in cui tali assegnamenti non risultino invalidati.

La ricerca locale nei problemi di soddisfacimento di vincoli è stata resa popolare dal lavoro di Kirkpatrick et al. (1983) sul **simulated annealing** (v. Capitolo 4), ampiamente usato nei problemi di scheduling. L'euristica *min-conflicts* fu proposta per la prima volta da Gu (1989) e sviluppata indipendentemente da Minton et al. (1992). Sosic e Gu (1994) hanno mostrato come poteva essere applicata per risolvere il problema con 3.000.000 di regine in meno di un minuto. L'incredibile successo della ricerca locale con euristica min-conflicts sul problema a n regine portò a una riconsiderazione della natura e della prevalenza dei problemi "facili" e "difficili". Peter Cheeseman et al. (1991) ha studiato la difficoltà dei CSP generati casualmente e ha scoperto che quasi tutti i problemi così creati sono banalmente facili oppure non hanno soluzioni: per trovare istanze di problemi "difficili" si devono configurare i parametri del generatore di problemi in un determinato, piccolo intervallo, all'interno del quale è risolvibile circa la metà dei problemi. Discuteremo ulteriormente questo fenomeno nel Capitolo 7.

Gli studi sulla struttura e la complessità dei CSP hanno avuto origine con Freuder (1985), che ha dimostrato che la ricerca su alberi arco-consistenti non ha bisogno di backtracking. Un risultato simile, esteso agli ipergrafi aciclici, fu sviluppato nella comunità dei database (Beeri et al., 1983). Dalla pubblicazione di questi articoli sono stati fatti molti progressi e ottenuta una comprensione più generale sulla correlazione tra la complessità della soluzione di un CSP e la struttura del

backtracking guidato
dalle dipendenze

memorizzazione dei
vincoli

backmarking

backtracking dinamico

suo albero dei vincoli. Il concetto di larghezza d'albero fu introdotto dai teorici dei grafi Robertson e Seymour (1986). Dechter e Pearl (1987, 1989), partendo dal lavoro di Freuder, applicarono lo stesso concetto (che chiamarono **larghezza indotta**) ai problemi di soddisfacimento di vincoli, sviluppando la tecnica di scomposizione ad albero che abbiamo descritto nel Paragrafo 5.4. Sviluppando questo lavoro, insieme ad alcuni risultati della teoria dei database, Gottlob et al. (1999a, 1999b) svilupparono il concetto di **larghezza di iperalbero**, basato sulla descrizione del CSP come ipergrafo. Oltre a dimostrare che ogni CSP con larghezza di iperalbero w può essere risolto in un tempo $O(n^{w+1} \log n)$, hanno anche mostrato che la larghezza di iperalbero domina tutte le misure di "larghezza" precedentemente formulate, nel senso che ci sono casi in cui la larghezza di iperalbero è limitata e le altre misure no.

Esistono molte buone trattazioni generali delle tecniche per risolvere i CSP, tra cui quelle di Kumar (1992), Dechter e Frost (1999) e Bartak (2001); sono utili anche le voci di enciclopedia di Dechter (1992) e Mackworth (1992). Pearson e Jeavons (1997) offrono una panoramica delle classi trattabili di CSP, trattando sia i metodi di scomposizione strutturale che quelli che si appoggiano a proprietà dei dominî o degli stessi vincoli. Kondrak e van Beek (1997) hanno svolto uno studio analitico degli algoritmi di ricerca con backtracking, mentre il lavoro di Bacchus e van Run (1995) ha un taglio più empirico. I libri di Tsang (1993) e di Marriott e Stuckey (1998) approfondiscono l'argomento molto più di quanto abbiano potuto fare in questo capitolo. Una collezione curata da Freuder and Mackworth (1994) descrive molte interessanti applicazioni. Articoli sul soddisfacimento di vincoli compaiono regolarmente su *Artificial Intelligence* e sulla rivista specializzata, *Constraints*. Il più importante congresso è l'International Conference on Principles and Practice of Constraint Programming, chiamata spesso brevemente *CP*.

Esercizi

- 5.1 Definite con parole vostre i concetti: problema di soddisfacimento di vincoli, vincolo, ricerca con backtracking, consistenza d'arco, backjumping e min-conflicts.
- 5.2 Quante soluzioni esistono per il problema di coloratura di mappa della Figura 5.1?
- 5.3 Spiegate perché in una ricerca CSP è una buona euristica scegliere la variabile che ha *più* vincoli, ma il valore *meno* vincolante.

- pianificazione rettilinea
di pavimenti
- orario delle lezioni
- 
- 5.4 Considerate il problema di costruire (non risolvere) dei cruciverba:⁵ incassare parole in una griglia rettangolare. La griglia, che è fornita come parte del problema, specifica le caselle bianche e quelle nere. Presumete di avere a disposizione una lista di parole (dizionario) e che lo scopo sia riempire gli spazi bianchi della griglia usando un sottoinsieme di tale lista. Formulate il problema con precisione in due modi:
- come un generico problema di ricerca. Scegliete un algoritmo appropriato e specificate la funzione euristica, se pensate che sia necessaria. È meglio riempire gli spazi una lettera per volta o una parola per volta?
 - come un problema di soddisfacimento di vincoli. Le variabili devono essere parole o lettere?
- Quale formulazione pensate che funzionerà meglio? Perché?
- 5.5 Fornite formulazioni precise come CSP di ognuno dei seguenti problemi.
- Pianificazione rettilinea di pavimenti:** trovate spazi non sovrapposti all'interno di un grande rettangolo per piazzare un certo numero di rettangoli più piccoli.
 - Orario delle lezioni:** viene dato un numero prefissato di professori e classi, una lista di lezioni e una lista di possibili orari per ogni lezione. A ogni professore è associato l'insieme delle lezioni che può tenere.
- 5.6 Risolvete il problema di criptoaritmetica della Figura 5.2 a mano, usando il backtracking, la verifica in avanti, l'euristica MRV e quella del valore meno vincolante.
- 5.7 La Figura 5.5 riporta i risultati di vari algoritmi per il problema della n regine. Provate gli stessi algoritmi su problemi di coloratura di mappe generandoli casualmente come segue: spargete n punti in un quadrato di lato unitario; scegliete a caso un punto X e collegatelo con un segmento di retta al più vicino punto Y tale che X non vi sia già collegato e la linea non ne attraversi un'altra; ripetete il passo precedente finché non è possibile aggiungere nuove connessioni. Costruite la tabella delle prestazioni per il più grande n che riuscite a gestire, usando sia $d = 3$ che $d = 4$ colori. Commentate i risultati.
- 5.8 Usate l'algoritmo AC-3 per mostrare che la consistenza d'arco è in grado di rilevare l'inconsistenza dell'assegnamento parziale $\{WA = \text{rosso}, V = \text{blu}\}$ nel problema mostrato nella Figura 5.1.

⁵ Ginsberg et al. (1990) discutono diversi metodi per la creazione di cruciverba. Littman et al. (1999) affrontano il più difficile problema della loro risoluzione.

- 5.9 Qual è la complessità, nel caso pessimo, dell'esecuzione di AC-3 su un CSP strutturato ad albero?
- 5.10 AC-3 rimette nella coda *ogni* arco (X_k, X_i) ogni volta che *un qualsiasi* valore è cancellato dal dominio di X_i , anche se tutti i valori di X_k sono consistenti con diversi valori rimanenti di X_i . Supponete di tener traccia, per ogni arco (X_k, X_i) , del numero di valori restanti di X_i che sono consistenti con *ogni* valore di X_k . Spiegate come tenere aggiornati queste informazioni in modo efficiente e mostrate di conseguenza che la consistenza d'arco può essere assicurata in un tempo totale $O(n^2d^2)$.
- 5.11 Mostrate come un singolo vincolo ternario come " $A + B = C$ " può essere trasformato in tre vincoli binari usando una variabile ausiliaria. Potete presumere che i domini siano finiti (*suggerimento*: considerate una nuova variabile che assume valori che sono coppie di altri valori, e vincoli come " X è il primo elemento della coppia Y "). Quindi mostrate come possono essere trattati in maniera analoga vincoli con più di tre variabili. Infine, dimostrate che vincoli unari possono essere eliminati modificando il dominio delle variabili. Questo completa la dimostrazione che qualsiasi CSP può essere trasformato in un CSP che ha solamente vincoli binari.
- 5.12 Supponete che si sappia che un grafo ha un insieme di taglio dei cicli non più grande di k nodi. Descrivete un semplice algoritmo per trovare l'insieme di taglio minimo: il suo tempo di esecuzione non dev'essere molto più alto di $O(n^k)$ per un CSP con n variabili. Consultate la letteratura riguardante i metodi per trovare approssimazioni degli insiemi di taglio minimi in tempo polinomiale nelle dimensioni dell'insieme. L'esistenza di questi algoritmi rende applicabile nella pratica il metodo degli insiemi di taglio dei cicli?
- 5.13 Considerate il seguente rompicapo logico: in cinque case, ognuna di un colore diverso, vivono cinque persone di diversa nazionalità, ognuna delle quali preferisce una particolare (e distinta) marca di sigarette, bevanda e animale da compagnia. Dati i seguenti fatti, rispondete alla domanda: "dove vive la zebra, e in quale casa si beve acqua?".

L'inglese vive nella casa rossa.

Lo spagnolo ha il cane.

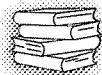
Il norvegese vive nella prima casa a sinistra.

Nella casa gialla si fumano sigarette Esportazione.

L'uomo che fuma le Alfa vive nella casa accanto a quella dell'uomo che possiede la volpe.

Il norvegese vive accanto alla casa blu.

Chi fuma le Stop possiede delle lumache.



Chi fuma le Nazionali beve succo di frutta.

L'ucraino beve il tè.

Il giapponese fuma sigarette Diana.

Le sigarette Esportazione sono fumate nella casa accanto a quella dove vive il cavallo.

Nella casa verde si beve caffè.

La casa verde è immediatamente alla destra (di chi guarda) di quella color avorio.

Nella casa centrale si beve latte.

Discutete diverse possibili rappresentazioni di questo problema sotto forma di CSP. Perché si potrebbe preferire una particolare rappresentazione rispetto alle altre?

Capitolo 6

Ricerca con avversari

In cui prendiamo in esame i problemi che sorgono quando cerchiamo di pianificare in un mondo in cui altri agenti pianificano contro di noi.

6.1 Giochi

Nel Capitolo 2 abbiamo parlato di ambienti multiagente, nei quali ogni agente deve considerare le azioni degli altri e in particolare i loro effetti sul proprio benessere. L'impredictibilità delle azioni altrui può dar luogo a molte possibili contingenze nel processo di risoluzione del problema, come abbiamo visto nel Capitolo 3. Abbiamo anche introdotto la distinzione tra ambienti multiagente cooperativi e competitivi. Gli ambienti competitivi, in cui gli obiettivi degli agenti sono in conflitto, danno origine a problemi di ricerca con avversari (*adversarial search*), più spesso indicati con il nome di giochi.

La teoria dei giochi matematica, una branca dell'economia, considera ogni ambiente multiagente come un gioco, indipendentemente dal fatto che l'interazione sia cooperativa o competitiva, a patto che l'influenza di ogni agente sugli altri sia "significativa".¹ Nell'IA i "giochi" appartengono normalmente a una categoria piuttosto specializzata, quella che i teorici chiamano giochi a somma zero con informazione perfetta, a turni e a due giocatori. Nella nostra terminologia questo significa ambienti deterministici e completamente osservabili, in cui ci sono due agenti

giochi

giochi a somma zero
informazione perfetta

¹ Gli ambienti con una quantità di agenti molto grande si trattano meglio se vengono considerati economie anziché giochi.

le cui azioni si devono alternare e le cui funzioni di utilità, alla fine della partita, sono sempre uguali ma di segno opposto. Ad esempio, se un giocatore vince una partita a scacchi (+1), l'altro deve necessariamente perderla (-1). È proprio quest'opposizione delle funzioni di utilità dei due agenti che ci fa parlare di "avversari". In questo capitolo considereremo brevemente anche i giochi multiplayer (cioè a più giocatori), quelli a somma non-zero e quelli stocastici, ma rimanderemo la discussione della teoria dei giochi vera e propria al Capitolo 17, nel 2º volume.

I giochi hanno impegnato le facoltà intellettuali degli esseri umani (talvolta in modo preoccupante) da quando esiste la civiltà. Per gli studiosi di intelligenza artificiale, la loro natura astratta li rende un oggetto di ricerca molto interessante. Lo stato di un gioco è facile da rappresentare, e solitamente le azioni degli agenti sono ristrette a un piccolo insieme le cui conseguenze sono definite da regole precise. I giochi fisici, come il croquet e l'hockey su ghiaccio, hanno descrizioni molto più complicate, una gamma molto più vasta di possibili azioni, e regole alquanto imprecise circa la legalità delle azioni stesse. Con l'eccezione del calcio per robot, i giochi fisici non hanno suscitato molto interesse nella comunità dell'IA.

I giochi sono una delle prime attività studiate dall'IA. Già nel 1950, praticamente non appena i computer erano diventati programmabili, gli scacchi erano stati affrontati da Konrad Zuse (inventore del primo computer programmabile e del primo linguaggio di programmazione), da Claude Shannon (inventore della teoria dell'informazione), da Norbert Wiener (fondatore della moderna teoria del controllo) e da Alan Turing. Da allora sono stati fatti progressi costanti nella qualità del gioco, fino al punto che le macchine hanno sorpassato gli esseri umani nella dama e nell'Othello, sconfitto i campioni mondiali (anche se non sempre) a scacchi e backgammon, e si sono dimostrati competitivi in molti altri giochi. L'eccezione principale è il Go, in cui i computer non riescono a superare un livello amatoriale.

I giochi, a differenza della maggior parte dei "problemi giocattolo" che abbiamo visto nel Capitolo 3, sono interessanti proprio perché sono troppo difficili da risolvere. Ad esempio, il fattore di ramificazione medio degli scacchi è di circa 35, e le partite spesso arrivano a 50 mosse per giocatore, per cui l'albero di ricerca avrà circa 35^{100} o 10^{154} nodi (benché il grafo di ricerca abbia "solo" circa 10^{40} nodi distinti). I giochi quindi, come il mondo reale, richiedono l'abilità di prendere una qualche decisione quando il calcolo di quella ottima non è realizzabile e penalizzano severamente l'inefficienza. Laddove l'implementazione di una ricerca A* che è efficiente solo la metà rispetto a un'altra richiederà semplicemente il doppio di tempo per l'esecuzione, un programma per scacchi che sfrutta il tempo a sua disposizione con un'efficienza dimezzata, a parità di tutte le altre variabili, sarà probabilmente spazzato via dai suoi avversari. La ricerca sui giochi ha quindi dato origine a tutta una serie di idee interessanti su come sfruttare al meglio il tempo disponibile.

Cominceremo con la definizione di mossa ottima e di un algoritmo per trovarla. Esamineremo poi le tecniche per scegliere una buona mossa quando il tempo a disposizione è limitato. La potatura ci permette di ignorare porzioni dell'albero di ricerca che non influiscono sulla scelta finale, mentre funzioni di valutazione euristiche ci consentono di approssimare l'utilità reale di uno stato senza eseguire una ricerca completa. Il Paragrafo 6.5 discute giochi come il backgammon, che contengono un fattore casuale; tratteremo anche il bridge, che include elementi di informazione imperfetta perché non tutte le carte sono visibili a ogni giocatore. Infine, esamineremo il livello raggiunto dai programmi più evoluti contro avversari umani e i possibili sviluppi futuri.

informazione
imperfetta

6.2 Decisioni ottime nei giochi

Prenderemo in considerazione i giochi per due giocatori, che chiameremo **MAX** e **MIN** per ragioni che saranno palesi tra poco. MAX muove per primo, dopodiché entrambi giocano a turno fino alla fine della partita: a questo punto vengono assegnati punti al vincitore e penalità al perdente. Un gioco può essere definito formalmente come un tipo di problema di ricerca con i seguenti componenti.

- ◆ Lo stato iniziale, che include la configurazione del tavoliere e l'indicazione del primo giocatore che muove.
- ◆ Una funzione successore, che restituisce una lista di coppie (*mossa, stato*), ognuna delle quali comprende una mossa legale e lo stato risultante dalla sua esecuzione.
- ◆ Un test di terminazione, che determina se la partita è finita. Gli stati che fanno finire la partita sono chiamati stati terminali.
- ◆ Una funzione utilità (chiamata anche *funzione obiettivo* o *funzione di payoff*), che assegna un valore numerico agli stati terminali. Negli scacchi i possibili risultati sono vittoria, sconfitta o pareggio, e i valori corrispondenti +1, -1 e 0. Altri giochi hanno una gamma più vasta di possibili risultati; nel backgammon il payoff va da +192 a -192. Questo capitolo si occupa principalmente dei problemi a somma zero, anche se saranno brevemente menzionati anche gli altri.

test di terminazione

Lo stato iniziale e le mosse legali di ogni giocatore definiscono un albero di gioco. La Figura 6.1 mostra una parte dell'albero di gioco per il tic-tac-toe (gioco del tris). Partendo dallo stato iniziale MAX ha nove possibili mosse. Il gioco si alterna tra il piazzamento di una X da parte di MAX e di una O da parte di MIN finché non viene raggiunto un nodo foglia che corrisponde a uno stato terminale, in cui un giocatore ha fatto tris oppure sono state riempite tutte le caselle. Il numero associato a ciascun nodo foglia indica il valore di utilità di quello stato terminale dal

albero di gioco

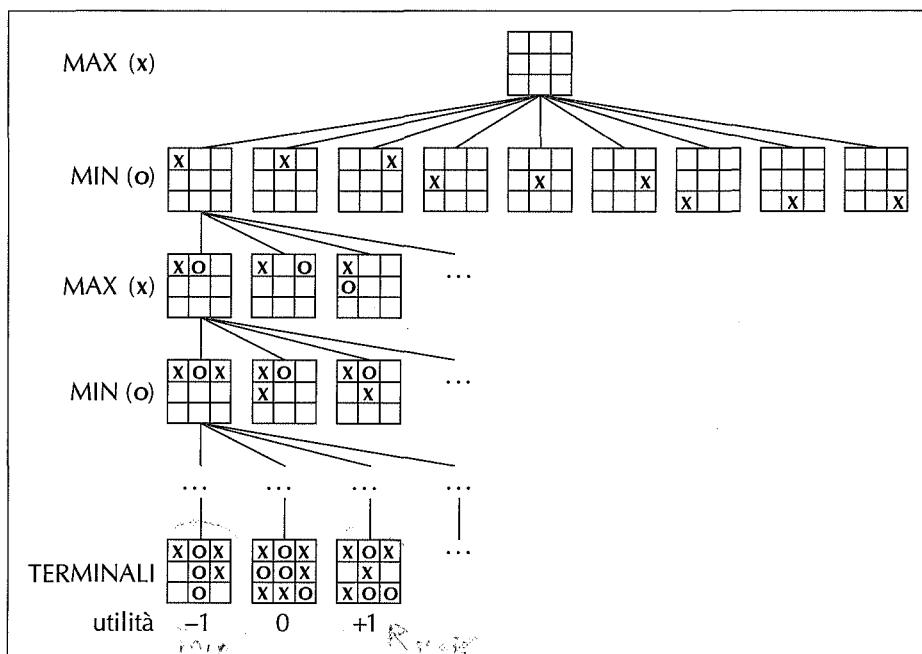


Figura 6.1 Un albero di ricerca (parziale) per il gioco del tris o tic-tac-toe. Il nodo più in alto è lo stato iniziale, e MAX muove per primo, ponendo una X in un riquadro vuoto. Abbiamo riportato una parte dell'albero di ricerca, indicando le mosse alternate di MIN (che usa il simbolo O) e MAX, fino all'eventuale raggiungimento degli stati terminali, a cui possono essere associati valori di utilità in base alle regole del gioco.

punto di vista di MAX; i valori alti sono buoni per MAX e cattivi per MIN (ed ecco spiegato come abbiamo assegnato i nomi ai giocatori). È compito di MAX usare l'albero di ricerca (e in particolare il valore di utilità degli stati terminali) per determinare la mossa migliore.

Strategie ottime

In un normale problema di ricerca, la soluzione ottima è costituita da una sequenza di mosse che portano a uno stato obiettivo; in questo caso uno stato terminale corrispondente alla vittoria dell'agente. In un gioco, d'altra parte, MIN può dire la sua: MAX quindi deve trovare una strategia contingente che specifichi la sua mossa nello stato iniziale, quindi le mosse in tutti gli stati possibili risultanti dalla prima mossa di MIN, poi le mosse negli stati risultanti dalle mosse di MIN in risposta a quelle, e così via. Una strategia ottima porta a un risultato che è almeno pari a quello di qualsiasi altra strategia, presumendo che si stia giocando contro un av-

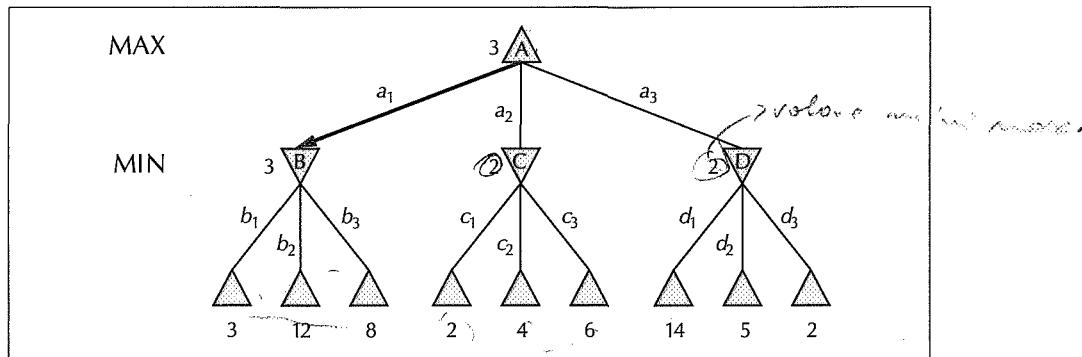


Figura 6.2 Un albero di gioco a due strati. I nodi Δ sono "nodi MAX", in cui è il turno di MAX a muovere, quelli ∇ sono "nodi MIN". I nodi terminali indicano i valori di utilità per MAX: gli altri sono etichettati con i loro valori minimax. Alla radice la mossa migliore di MAX è a_1 , perché porta al successore con il più alto valore minimax, mentre la replica migliore per MIN è b_1 , perché porta al successore con valore minimax più basso.

versario infallibile. Cominceremo col mostrare come trovare questa strategia ottima, anche se MAX troverà molto difficile calcolarla per giochi più complessi del tic-tac-toe.

Anche un gioco semplice come il tris è troppo complesso per poter disegnare l'intero albero di gioco, così useremo come esempio la partita banale illustrata nella Figura 6.2. Le mosse possibili per MAX nel nodo radice sono etichettate a_1 , a_2 , e a_3 . Le possibili risposte di MIN alla mossa a_1 sono b_1 , b_2 , b_3 e così via. Questa particolare partita finisce dopo una sola mossa di MAX e una di MIN. In gergo specialistico si dice che l'albero è profondo una mossa, e ogni mossa consiste in due "mezze mosse" ognuna delle quali è chiamata ply (o, in italiano, strato). In questa partita i valori di utilità degli stati terminali vanno da 2 a 14.

Dato un albero di gioco, la strategia ottima può essere determinata esaminando il valore minimax di ogni nodo, che scriveremo $\text{VALORE-MINIMAX}(n)$. Il valore minimax di un nodo corrisponde all'utilità (per MAX) di trovarsi nello stato corrispondente, presumendo che entrambi gli agenti giochino in modo ottimo da lì alla fine della partita. Ovviamente, il valore minimax di uno stato terminale si riduce alla sua utilità. Inoltre, avendone la possibilità, MAX preferirà muoversi in uno stato di valore massimo, MIN in uno di valore minimo. Risulta quindi ciò che segue:

$$\text{VALORE-MINIMAX}(n) =$$

$$\begin{cases} \text{UTILITÀ}(n) & \text{se } n \text{ è uno stato terminale} \\ \max_{s \in \text{Successori}(n)} \text{VALORE-MINIMAX}(s) & \text{se } n \text{ è un nodo MAX} \\ \min_{s \in \text{Successori}(n)} \text{VALORE-MINIMAX}(s) & \text{se } n \text{ è un nodo MIN.} \end{cases}$$

ply

valore minimax

decisione minimax

Applichiamo queste definizioni all'albero di gioco nella Figura 6.2: i nodi terminali al livello più basso sono già etichettati con i loro valori di utilità. Il primo nodo MIN, etichettato B , ha tre successori con valori 3, 12 e 8, cosicché il suo valore minimax è 3. Similmente, gli altri due nodi MIN hanno un valore minimax di 2. La radice è un nodo MAX; i suoi successori hanno valori minimax di 3, 2 e 2; quindi la radice ha un valore minimax di 3. Possiamo anche identificare la **decisione minimax** nella radice: l'azione a_1 è la scelta ottima per MAX, perché porta al successore con il più alto valore minimax.

Questa definizione di strategia ottima per MAX presume che anche MIN giochi in modo ottimo: in altre parole, massimizza il risultato di MAX nel *caso pessimo*. Che succede se MIN non gioca ottimamente? In tal caso è facile dimostrare (Esercizio 6.2) che MAX farà ancora meglio. Possono esistere strategie che si comportano meglio di quella minimax nel *caso di avversari subottimi*; tali strategie però otterranno invariabilmente risultati peggiori contro avversari ottimi.

algoritmo minimax

L'algoritmo minimax

L'**algoritmo minimax** (Figura 6.3) calcola la decisione minimax per lo stato corrente usando un semplice calcolo ricorsivo del valore minimax di ogni stato successore, implementando direttamente le equazioni che lo definiscono. La ricorsione percorre tutto l'albero sino alle foglie, quindi i valori minimax sono "portati su" (*backed-up*) attraverso l'albero durante la fase di ritorno. Ad esempio, nella Figura 6.2, l'algoritmo prima raggiunge con la ricorsione i tre nodi in basso a sinistra e usa la funzione UTILITÀ per scoprire che i loro valori sono rispettivamente 3, 12 e 8. Poi prende il minimo di questi valori, 3, e lo restituisce come valore backed-up del nodo B . Un processo simile fornisce i valori 2 per C e 2 per D . Infine, il massimo tra 3, 2 e 2 fornisce il valore 3 della radice.

L'algoritmo minimax esegue una completa visita in profondità dell'albero di gioco. Se la profondità massima dell'albero è m , e in ogni punto ci sono b mosse legali, la complessità temporale dell'algoritmo sarà $O(b^m)$. Quella spaziale è $O(bm)$ se l'algoritmo genera i successori tutti insieme, oppure $O(m)$ se li genera uno per volta (v. pag. 102). Per giochi reali, ovviamente, il costo temporale è assolutamente inaccettabile, ma quest'algoritmo fornisce la base per un'analisi matematica dei giochi e per sviluppare algoritmi più pratici.

Decisioni ottime nei giochi multiplayer

Molti giochi popolari permettono la presenza di più giocatori. Vediamo come si può estendere il concetto di minimax ai giochi multiplayer: dal punto di vista tecnico non ci sono problemi, ma questo permette di sollevare nuove questioni interessanti.

Prima di tutto, dobbiamo sostituire il valore singolo di ogni nodo con un vettore di valori. Ad esempio, in un gioco a tre con i giocatori A , B e C , a ogni nodo si dovrà associare un vettore $\langle v_A, v_B, v_C \rangle$. Per gli stati terminali, il vettore fornirà

function DECISIONE-MINIMAX(*stato*) returns un'azione

inputs: *stato*, lo stato corrente nel gioco

$v \leftarrow \text{VALORE-MAX}(\text{stato})$

return l'azione in SUCCESSORI(*stato*) con valore v

function VALORE-MAX(*stato*) returns un valore di utilità

if TEST-TERMINALE(*stato*) **then return** UTILITÀ(*stato*)

$v \leftarrow -\infty$

for a, s in SUCCESSORI(*stato*) **do**

$v \leftarrow \text{MAX}(v, \text{VALORE-MIN}(s))$

return v

function VALORE-MIN(*stato*) returns un valore di utilità

if TEST-TERMINALE(*stato*) **then return** UTILITÀ(*stato*)

$v \leftarrow \infty$

for a, s in SUCCESSORI(*stato*) **do**

$v \leftarrow \text{MIN}(v, \text{VALORE-MAX}(s))$

return v

Figura 6.3 Un algoritmo per calcolare decisioni minimax. L'algoritmo restituisce l'azione corrispondente alla miglior mossa possibile, cioè quella che porta allo stato di massima utilità, sotto l'ipotesi che lo scopo dell'avversario sia minimizzare la stessa funzione. Le funzioni VALORE-MAX e VALORE-MIN attraversano l'intero albero fino alle foglie per determinare il valore "backed-up" di uno stato.

l'utilità dello stato dal punto di vista di ogni giocatore (nei giochi a due a somma zero, il vettore è superfluo perché i due valori sono sempre uno l'opposto dell'altro). Il modo più semplice di implementare tutto ciò è far sì che la funzione UTILITÀ restituisca un vettore.

Ora consideriamo gli stati non terminali: prendiamo il nodo marcato con la X nell'albero di gioco della Figura 6.4. In quello stato, il giocatore C deve decidere cosa fare. Le due scelte portano a stati terminali con vettori di utilità $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ e $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Dato che 6 è maggiore di 3, C dovrebbe scegliere la prima mossa. Questo significa che se si raggiunge lo stato X , le mosse successive porteranno a uno stato terminale con guadagni $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Ne consegue che il valore di X "portato su" al nodo superiore è questo vettore. In generale, il valore che viene passato verso l'alto (backed-up) da un nodo n è il vettore di utilità del successore più favorevole a chi sta scegliendo la mossa in n .

giocatore che muove

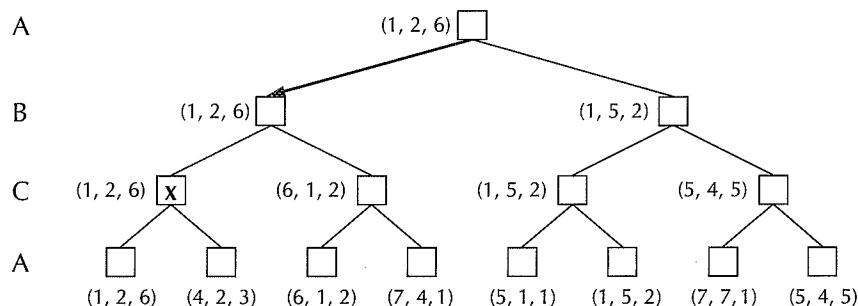


Figura 6.4 I primi tre livelli, o strati, di un albero di gioco con tre partecipanti (A, B, C). Ogni nodo è etichettato con i valori dal punto di vista di ogni giocatore. La mossa migliore partendo dalla radice è evidenziata con la freccia.

alleanze

Chiunque abbia provato un gioco multiplayer, come ad esempio il famoso Diplomacy™, si è ben presto accorto che le cose sono molto più complicate del caso a due. I giochi multiplayer normalmente prevedono alleanze tra i giocatori, esplicitamente formalizzate o no, che vengono strette e poi spezzate durante lo svolgimento della partita. Come possiamo considerare un comportamento simile? Le alleanze sono una conseguenza naturale delle strategie ottime di ogni giocatore? I risultati dicono che potrebbe essere così. Supponiamo ad esempio che A e B si trovino in una posizione debole e C in una forte. In questo caso la scelta ottima per A e B spesso consiste nell'attaccare C invece dell'altro, per impedirgli di distruggere entrambi individualmente. In questo modo, la collaborazione emerge da un comportamento puramente egoistico. Naturalmente, non appena C si indebolisce sotto l'attacco congiunto, l'alleanza perde valore, e sia A che B potrebbero violare gli accordi. In alcuni casi, la presenza di alleanze esplicite rende semplicemente più concreto ciò che si sarebbe verificato comunque. In altri casi la rottura di un'alleanza costituisce una macchia sulla propria reputazione, per cui i giocatori devono bilanciare i vantaggi immediati del tradimento con l'inconveniente a lungo termine di essere notoriamente inaffidabili.

Se il gioco non è a somma zero, si può verificare una collaborazione anche tra due soli giocatori. Supponiamo per esempio che ci sia uno stato terminale con guadagni $\langle v_A = 1000, v_B = 1000 \rangle$, e che 1000 sia il valore di utilità più alto per ogni giocatore. Allora la strategia ottima per ambedue i giocatori è raggiungere questo stato; questo significa che i due coopereranno automaticamente per raggiungere un obiettivo desiderabile per entrambi.

6.3 Potatura alfa-beta

(*alfa - beta pruning*)

Il problema della ricerca minimax è che il numero degli stati da esaminare cresce esponenzialmente con il numero di mosse. Sfortunatamente non possiamo eliminare l'esponente, ma possiamo ridurlo alla metà: infatti è possibile calcolare la decisione minimax corretta senza guardare tutti i nodi dell'albero di gioco. Possiamo prendere in prestito l'idea della potatura, che abbiamo visto nel Capitolo 4, per evitare di prendere in considerazione grandi sezioni dell'albero: in particolare esamineremo una tecnica chiamata potatura alfa-beta. Applicata a un albero minimax standard, la potatura restituisce lo stesso risultato della tecnica minimax pura, ma "pota" i rami che non possono influenzare la decisione finale.

Consideriamo ancora l'albero a due strati della Figura 6.2. Calcoliamo ancora una volta la decisione ottima, ponendo particolare attenzione alle informazioni che abbiamo a disposizione in ogni punto del processo. I vari passi sono illustrati nella Figura 6.5. Il risultato è che possiamo arrivare alla decisione minimax senza neppure valutare due dei nodi foglia.

Un altro modo di considerare questa tecnica è come una semplificazione della formula per calcolare VALORE-MINIMAX. Siano x e y i valori dei due successori non valutati del nodo C nella Figura 6.5, e sia z il minimo tra x e y . Il valore del nodo radice è dato da

$$\begin{aligned} \text{VALORE-MINIMAX}(radice) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \quad \text{dove } z \leq 2 \\ &= 3. \end{aligned}$$

In altre parole, il valore della radice e di conseguenza la decisione minimax sono indipendenti dal valore delle foglie potate x e y .

La potatura alfa-beta può essere applicata ad alberi di qualunque profondità, e spesso invece di foglie è possibile potare interi sottoalberi. Il principio generale è questo: considerate un nodo n da qualche parte nell'albero (v. Figura 6.6), tale che Giocatore abbia la facoltà di muoversi in quel nodo. Se c'è una scelta migliore m a livello del nodo padre o di un qualunque nodo precedente, allora n non sarà mai raggiunto in tutta la partita. Possiamo quindi potare n non appena abbiamo raccolto abbastanza informazioni (esaminando alcuni dei suoi discendenti) da raggiungere questa conclusione.

Ricordate che quella minimax è una ricerca in profondità, per cui in ogni momento dobbiamo considerare solo i nodi lungo un singolo cammino dell'albero. La potatura alfa-beta prende il suo nome dai seguenti due parametri che descrivono i limiti sui valori "portati su" in un qualsiasi punto del cammino:

α = il valore della scelta migliore (quella con valore più alto) per MAX che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino;

β = il valore della scelta migliore (quella con valore più basso) per MIN che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino.

potatura alfa-beta



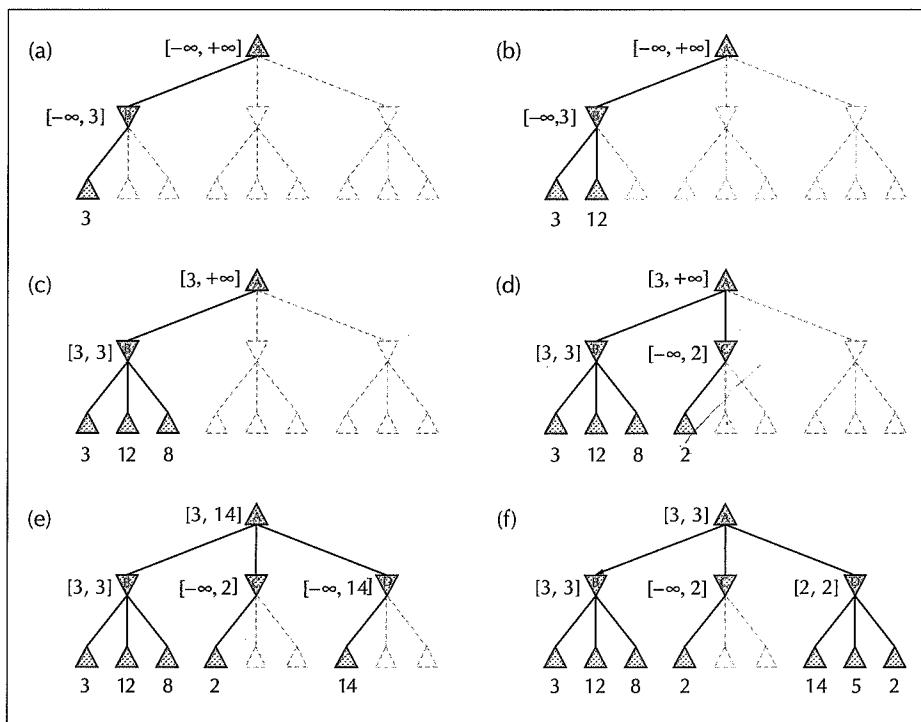


Figura 6.5 Fasi del calcolo della decisione ottima per l'albero di gioco della Figura 6.2: in ogni punto viene indicato l'intervallo dei possibili valori di ogni nodo. (a) La prima foglia sotto B ha valore 3; ne segue che B , che è un nodo MIN, ha un valore *massimo* di 3. (b) La seconda foglia sotto B ha valore 12; MIN la eviterebbe, per cui il suo valore è ancora al massimo 3. (c) La terza foglia sotto B ha valore 8; ora abbiamo considerato tutti i successori di B , per cui possiamo dire che il suo valore è esattamente 3. Ne possiamo dedurre che il valore *minimo* della radice è 3, perché MAX ha appunto a disposizione tale scelta. (d) La prima foglia sotto C ha valore 2; ne segue che C , che è un nodo MIN, ha un valore *massimo* di 2. Ma sappiamo che B vale 3, per cui MAX non sceglierebbe C in nessun caso: non c'è quindi alcuna ragione di considerare gli altri successori di C . Questo è un esempio di potatura alfa-beta. (e) La prima foglia sotto D ha valore 14, cosicché D vale *al massimo* 14: questo valore è ancora superiore alla migliore alternativa per MAX (che è 3), per cui dobbiamo continuare a esplorare i successori di D . Notate che ora conosciamo gli estremi di tutti i successori della radice, e possiamo dire che il suo valore massimo è 14. (f) Il secondo successore di D vale 5, e così dobbiamo ancora continuare a esplorare. Il terzo successore ha valore 2, per cui D vale esattamente 2. La decisione di MAX nella radice è muovere nello stato B , che offre un valore di 3.

La ricerca alfa-beta aggiorna i valori di α e β man mano che procede e pota i rami restanti che escono dal nodo (ovvero, fa terminare le chiamate ricorsive) non appena determina che il valore del nodo corrente è peggio di quello di α per MAX o, rispettivamente, di β per MIN. L'algoritmo completo è riportato nella Figura 6.7. Incoraggiamo i lettori a seguirne passo passo l'esecuzione sull'albero della Figura 6.5.

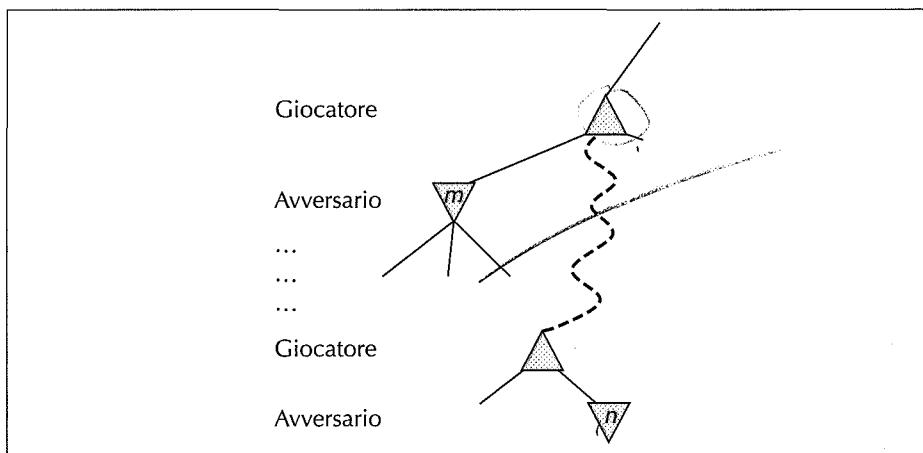


Figura 6.6
Potatura alfa–beta:
il caso generale.
Se per Giocatore
 m è preferibile a
 n , durante il gioco
non arriveremo
mai a n .

L'efficacia della potatura alfa–beta dipende fortemente dall'ordine in cui sono esaminati i successori. Ad esempio, nella Figura 6.5(e) e (f), non abbiamo potuto potare alcun successore di D perché tutti i successori peggiori (dal punto di vista di MIN) sono stati generati prima. Se il terzo successore fosse stato generato per primo, saremmo stati in grado di potare gli altri due. Questo ci suggerisce che è una buona idea cercare di esaminare per primi i successori più promettenti.

Se diamo per ipotesi che questo possa essere fatto,² risulta che per scegliere la mossa migliore la ricerca alfa–beta deve esaminare solo $O(b^{m/2})$ nodi, invece degli $O(b^m)$ richiesti da minimax. Questo significa che il fattore di ramificazione effettivo diventa \sqrt{b} invece di b : nel caso degli scacchi, 6 invece di 35. Per esprimere in un altro modo lo stesso risultato, si può dire che la ricerca alfa–beta può guardare nel futuro due volte più lontano di quanto possa fare minimax nello stesso lasso di tempo. Se i successori sono esaminati in ordine casuale anziché best-first, il numero totale di nodi esaminati sarà circa $O(b^{3m/4})$ per b di grandezza moderata. Per gli scacchi, una funzione di ordinamento abbastanza semplice (cercare prima di catturare pezzi, poi di minacciarli, poi le mosse in avanti, infine quelle all'indietro) consente di arrivare a un fattore 2 di distanza dal risultato best-first $O(b^{m/2})$. Aggiungere schemi dinamici di ordinamento delle mosse, come provare per prima cosa l'insieme delle mosse migliori calcolate nel turno precedente, ci porta molto vicini al limite teorico.

² Naturalmente non può essere fatto in modo perfetto; altrimenti la funzione di ordinamento potrebbe essere sfruttata per giocare una partita perfetta!

function RICERCA-ALFA-BETA(*stato*) **returns** un'azione

inputs: *stato*, lo stato corrente nel gioco

$v \leftarrow \text{VALORE-MAX}(stato, -\infty, +\infty)$

return l'azione in SUCCESSORI(*stato*) con valore v

function VALORE-MAX(*stato*, α , β) **returns** un valore di utilità

inputs: *stato*, lo stato corrente nel gioco

α , il valore della migliore alternativa per MAX lungo il cammino verso *stato*

β , il valore della migliore alternativa per MIN lungo il cammino verso *stato*

if TEST-TERMINALE(*stato*) **then return** UTILITÀ(*stato*)

$v \leftarrow -\infty$

for a, s in SUCCESSORI(*stato*) **do**

$v \leftarrow \text{MAX}(v, \text{VALORE-MIN}(s, \alpha, \beta))$

if $v \geq \beta$ **then return** v

$\alpha \leftarrow \text{MAX}(\alpha, v)$

return v

function VALORE-MIN(*stato*, α , β) **returns** un valore di utilità

inputs: *stato*, lo stato corrente nel gioco

α , il valore della migliore alternativa per MAX lungo il cammino verso *stato*

β , il valore della migliore alternativa per MIN lungo il cammino verso *stato*

if TEST-TERMINALE(*stato*) **then return** UTILITÀ(*stato*)

$v \leftarrow +\infty$

for a, s in SUCCESSORI(*stato*) **do**

$v \leftarrow \text{MIN}(v, \text{VALORE-MAX}(s, \alpha, \beta))$

if $v \leq \alpha$ **then return** v

$\beta \leftarrow \text{MIN}(\beta, v)$

return v

Figura 6.7 L'algoritmo di ricerca alfa-beta. Notate che le procedure sono le stesse di quelle MINIMAX della Figura 6.3, eccetto le due righe in VALORE-MAX e VALORE-MIN che mantengono aggiornati α e β (e il codice che serve a passare questi parametri da una procedura all'altra).

Nel Capitolo 3 abbiamo notato che gli stati ripetuti nell'albero di ricerca possono dare origine a un aumento esponenziale del costo di ricerca. Nei giochi, gli stati ripetuti si verificano di frequente a causa delle trasposizioni: permutazioni diverse

della stessa sequenza di mosse che finiscono tutte nella stessa configurazione della scacchiera. Ad esempio, se Bianco ha una mossa a_1 a cui Nero risponderà con b_1 e, dall'altra parte della scacchiera, una mossa indipendente a_2 a cui Nero può contrapporre la risposta b_2 , le sequenze $[a_1, b_1, a_2, b_2]$ e $[a_1, b_2, a_2, b_1]$ si concluderanno entrambe nella stessa posizione (così come le permutazioni che cominciano con la mossa a_2). Vale la pena di memorizzare la valutazione di questa posizione in una tabella di hash la prima volta che viene incontrata, in modo da non doverla ricalcolare ogni volta che si ripresenta. La tabella di hash di posizione precedentemente calcolate viene tradizionalmente chiamata tabella delle trasposizioni; è essenzialmente identica alla lista chiusa di RICERCA-GRAFO (v. pag. 111). L'uso di una tabella di trasposizione può rappresentare un vantaggio notevole, arrivando talvolta negli scacchi a raddoppiare la profondità raggiungibile. D'altra parte, se si valutano milioni di nodi al secondo, non si può tenerli tutti nella tabella: per scegliere quelli più utili sono state sviluppate diverse strategie.

tabella delle trasposizioni

6.4 Decisioni imperfette in tempo reale

L'algoritmo minimax genera l'intero spazio di ricerca del gioco, mentre quello alfa-beta ci permette di poterne una buona parte. Ciononostante, anche alfa-beta deve condurre la ricerca fino agli stati terminali, almeno per una porzione dello spazio di ricerca. Questa profondità normalmente non è gestibile, perché le mosse devono essere calcolate in un tempo ragionevole, che tipicamente non può superare qualche minuto. L'articolo di Shannon del 1950, Programming a computer for playing chess, proponeva che i programmi "tagliassero" la ricerca prima di raggiungere le foglie applicando una funzione di valutazione euristica agli stati, di fatto trasformando i nodi non terminali in foglie. In altre parole, Shannon suggeriva di modificare minimax o alfa-beta in due modi: sostituendo la funzione di utilità con quella euristica di valutazione EVAL che fornisce una stima dell'utilità della posizione raggiunta; rimpiazzando il test di terminazione con un test di taglio (cutoff test), che decide quando applicare EVAL.

funz euristiche
di valutazione
test di taglio

Funzioni di valutazione

Una funzione di valutazione restituisce una stima del guadagno atteso in una determinata posizione, proprio come le funzioni euristiche del Capitolo 4 forniscono una stima della distanza dall'obiettivo. L'idea non era nuova quando Shannon la propose: per secoli, i giocatori di scacchi (e gli appassionati di altri giochi) hanno sviluppato metodi per giudicare il valore di una posizione, anche perché gli esseri umani hanno una capacità di ricerca ancora più limitata di quella dei computer. È chiaro che le prestazioni di un programma dipendono dalla qualità della sua funzione di valutazione: se è inaccurata, potrebbe guidare l'agente verso posizioni che si rivelano perdenti. Come si fa, precisamente, a progettare delle buone funzioni di valutazione?

Prima di tutto, la funzione di valutazione dovrebbe ordinare gli stati *terminali* nello stesso modo della vera funzione *di utilità*; in caso contrario l'agente potrebbe scegliere mosse subottime anche quando è in grado di “vedere” fino alla fine della partita. In secondo luogo, i calcoli non dovrebbero richiedere troppo tempo! Dopotutto la funzione di valutazione potrebbe invocare DECISIONE-MINIMAX come subroutine e calcolare così il valore esatto della posizione, ma questo vanificherebbe l'intero scopo del discorso, che è quello di risparmiare tempo. In terzo luogo, per gli stati non terminali, la funzione di valutazione dovrebbe avere una forte correlazione con la probabilità reale di vincere la partita.

L'uso della frase “probabilità di vincere” potrebbe stupire qualcuno. Alla fine, gli scacchi non sono un gioco di fortuna: lo stato corrente è noto con sicurezza, e non ci sono dadi. Ma se la ricerca dev'essere interrotta in stati non terminali, l'algoritmo sarà necessariamente *incerto* sui risultati finali di quegli stati. Questo tipo di incertezza deriva da limiti computazionali, non dalla mancanza di informazioni. Data la quantità limitata di calcoli che la funzione di valutazione può svolgere in un determinato stato, il meglio che può fare è fornire una stima del risultato finale.¹¹¹

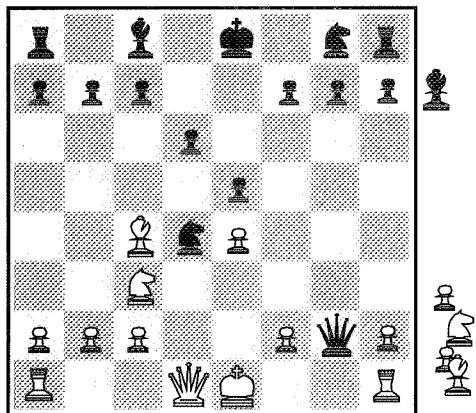
caratteristiche

Rendiamo più concreto questo concetto. La maggior parte delle funzioni di valutazione ragiona in base alle caratteristiche di uno stato, ad esempio, in una partita a scacchi, il numero di pedoni posseduti da ogni giocatore. Le caratteristiche, prese insieme, definiscono *categorie o classi di equivalenza*: gli stati di una categoria hanno lo stesso valore per tutte le caratteristiche. In generale, ogni categoria potrà contenere stati che portano alla vittoria, altri che portano al pareggio e altri ancora alla sconfitta. La funzione di valutazione non può sapere quali sono, ma può restituire un singolo valore che riflette la *proporzione* di stati per ogni risultato. Supponiamo ad esempio che la nostra esperienza suggerisca che il 72% degli stati in una categoria conducano a una vittoria (utilità +1); il 20% a una sconfitta (-1) e l'8% a un pareggio (0). In questo caso una valutazione ragionevole per gli stati appartenenti a tale categoria è la media pesata o *valore atteso*: $(0,72 \times +1) + (0,20 \times -1) + (0,08 \times 0) = 0,52$. In via di principio, si può determinare il valore atteso di ogni categoria, fornendo così una funzione di valutazione applicabile a ogni stato. Come con gli stati terminali, non è neppure necessario che la funzione di valutazione restituisca effettivamente i valori attesi, finché non viene alterato l'*ordinamento* degli stati.

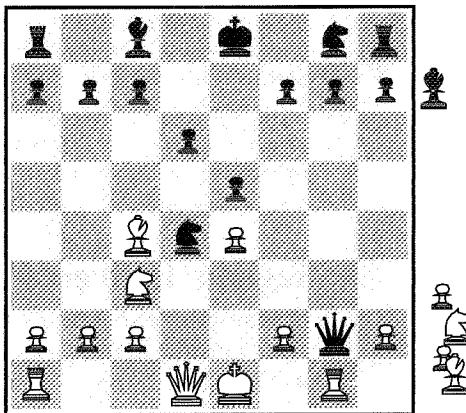
valore atteso

valore del materiale

Nella pratica, per stimare correttamente le probabilità di vittoria questo tipo di analisi richiede troppe categorie e quindi una quantità eccessiva di esperienza. La maggior parte delle funzioni di valutazione calcolano valori separati per ogni caratteristica, *combinandoli* poi insieme per formare il valore finale. I libri di scacchi per principianti, ad esempio, forniscono un *valore del materiale* approssimativo per ogni pezzo: ogni pedone vale 1, un cavallo o alfiere 3, una torre 5 e la regina 9. Altre caratteristiche come “una buona disposizione dei pedoni” o “la difesa del re” potrebbero valere, poniamo, mezzo pedone. Tutte queste caratteristiche sono semplicemente sommate per ottenere la valutazione di una posizione. Un vantaggio



(a) il Bianco muove



(b) il Bianco muove

Figura 6.8 Due posizioni leggermente diverse sulla scacchiera. In (a), il Nero ha un vantaggio di un cavallo e due pedoni e vincerà la partita. In (b), il Nero perderà dopo che il Bianco avrà catturato la regina.

sicuro equivalente a un pedone rappresenta una buona possibilità di vittoria, e un vantaggio sicuro equivalente a tre pedoni dovrebbe garantire quasi certamente la vittoria, come si vede nella Figura 6.8(a). Matematicamente questo tipo di valutazione è chiamata **funzione lineare pesata**, perché può essere espressa come

$$\text{EVAL}(s) = \hat{w}_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s),$$

funzione lineare
pesata

dove ogni w_i è un peso (*weight*) e ogni f_i è una caratteristica (*feature*) di una posizione. Per gli scacchi, ogni f_i potrebbe essere il numero dei pezzi di un particolare tipo sulla scacchiera, w_i il loro valore (1 per i pedoni, 3 per gli alfieri, etc.).

Sommare i valori delle caratteristiche sembra una cosa ragionevole, ma in effetti presume che si accetti l'ipotesi che il contributo di ogni caratteristica sia indipendente dal valore delle altre. Ad esempio, assegnare il valore 3 a un alfiere non tiene conto del fatto che questi pezzi sono più potenti nei finali delle partite, quando hanno più spazio per muoversi. Per questa ragione, i programmi più recenti per gli scacchi e altri giochi utilizzano combinazioni di caratteristiche *non-lineari*: una coppia di alfieri potrebbe valere leggermente più del doppio di un alfiere singolo, che a sua volta avrà più valore nel finale che all'inizio della partita.

I lettori più attenti avranno già notato che le caratteristiche degli stati e i pezzi *non fanno parte* delle regole degli scacchi! Sono state sviluppate dagli esseri umani nel corso dei secoli. Data la forma lineare della valutazione, le caratteristiche e i

pesi forniscono la migliore approssimazione del vero ordinamento degli stati in base al loro valore. In particolare, l'esperienza suggerisce che un vantaggio sicuro di materiale superiore a un punto permetterà probabilmente di vincere la partita, tenendo ferme tutte le altre variabili; un vantaggio di tre punti rappresenta una vittoria quasi certa. Nei giochi in cui non è disponibile questo tipo di conoscenza, i pesi della funzione di valutazione possono essere stimati usando tecniche di apprendimento automatico: la loro applicazione agli scacchi ha confermato che un alfiere vale effettivamente circa tre pedoni.

Tagliare la ricerca

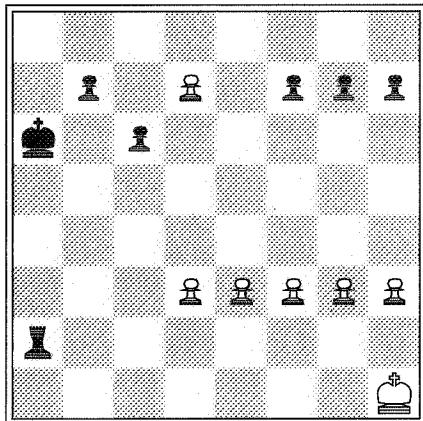
Il prossimo passo è modificare RICERCA-ALFA-BETA in modo che invochi la funzione euristica EVAL quando è il momento di tagliare la ricerca. Per quanto riguarda l'implementazione, si sostituiscono le due righe della Figura 6.7 che eseguono TEST-TERMINALE con la seguente:

```
if TEST-TAGLIO(stato, profondità) then return EVAL(stato)
```

È necessario anche modificare il codice in modo che la *profondità corrente* sia incrementata a ogni chiamata ricorsiva. Il modo più semplice di controllare la quantità di ricerca effettuata è stabilire un limite alla *profondità*, in modo tale che TEST-TAGLIO (stato, profondità) restituiscia *true* ogni volta che questa supera un valore prefissato *d* (naturalmente deve anche restituire *true* per tutti gli stati terminali, come faceva TEST-TERMINALE). La profondità *d* dev'essere scelta in modo tale che la quantità di tempo impiegato non superi quella consentita dalle regole del gioco.

Un approccio più robusto è rappresentato dalla ricerca ad approfondimento iterativo, che abbiamo definito nel Capitolo 3. Quando scade il tempo, il programma restituisce la mossa calcolata con la più profonda ricerca completata. Questi approcci possono comunque portare a commettere errori, a causa della natura approssimata della funzione di valutazione. Consideriamo ancora la semplice funzione di valutazione per gli scacchi basata sul *vantaggio di materiale*. Supponiamo che il programma esegua la ricerca fino al *limite di profondità*, raggiungendo la posizione della Figura 6.8(b), in cui il Nero ha un vantaggio di un cavallo e due pedoni. Il programma assocerebbe quindi a questo stato un valore euristico molto positivo, ritenendo che la situazione porterà a una molto probabile vittoria del Nero. Ma la mossa successiva del Bianco cattura la regina nera senza alcuna compensazione: la posizione quindi assicura una vittoria quasi certa del Bianco, ma questo si può appurare soltanto guardando ancora una mossa avanti.

Palesemente serve un test di taglio più sofisticato. La funzione di valutazione dovrebbe essere applicata solo a posizioni quiescenti: quelle, cioè, per cui è improbabile che si verifichino grandi variazioni di valore nelle mosse immediatamente successive. Negli scacchi, ad esempio, le posizioni in cui possono essere effettuate catture favorevoli non sono quiescenti per una funzione di valutazione che si limita a contare il materiale. Le posizioni non quiescenti possono essere espansse ulteriormente fino a raggiungere posizioni quiescenti. Questa ricerca aggiuntiva pren-



il Nero muove

Figura 6.9 L'effetto orizzonte: una serie di scacchi da parte della torre nera spinge l'inevitabile promozione del pedone bianco "oltre l'orizzonte" e fa sì che questa posizione appaia vincente per il Nero, quando in realtà si tratta di una sicura vittoria del Bianco.

de il nome di ricerca di quiescenza; talvolta viene ristretta affinché consideri solo certi tipi di mosse, come le catture, che possono risolvere velocemente le situazioni di incertezza.

L'effetto orizzonte è più difficile da eliminare. Questo problema sorge quando il programma deve considerare una mossa dell'avversario che causa grave danno e, a conti fatti, non è evitabile. Considerate la posizione nella Figura 6.9. il Nero ha un vantaggio di materiale, ma il Bianco può avanzare il pedone dalla settima all'ottava traversa, ottenendo così una promozione a regina e assicurandosi una facile vittoria. Il Nero può rimandare la promozione per 7 mosse continuando a mettere il re avversario sotto scacco, ma non può evitare la promozione. Il problema della ricerca a profondità limitata è che il programma crederà davvero che queste sette mosse possano prevenire la promozione del pedone: si dice che le mosse hanno spinto la promozione "oltre l'orizzonte della ricerca", abbastanza lontano dallo stato corrente da non poter essere più rilevata.

Man mano che le migliorie nell'hardware porteranno a ricerche sempre più profonde, è sensato prevedere che l'effetto orizzonte si verificherà con frequenza sempre minore: lunghe sequenze tese a ritardare una mossa inevitabile diventeranno più rare. Un metodo efficace per evitare l'effetto orizzonte senza aggiungere un costo di ricerca troppo alto è costituito dalle estensioni singole. Un'estensione singola è una mossa "palesemente migliore" di tutte le altre in una data posizione. Una ricerca basata sulle estensioni singole può andare molto oltre il normale limi-

ricerca di quiescenza

effetto orizzonte

estensioni singole

potatura in avanti

te di profondità senza grandi costi, perché il suo fattore di ramificazione è 1. La ricerca di quiescenza può essere considerata una variante delle estensioni singole. Nella Figura 6.9, una ricerca a estensione singola arriverà a capire che la promozione del pedone è inevitabile, a patto che le mosse della torre nera e del re bianco possano essere identificate come “palesemente migliori” delle alternative.

Fin qui abbiamo considerato il taglio della ricerca a un certo livello e una potatura alfa–beta che sicuramente non influenza il risultato finale. È anche possibile eseguire una **potatura in avanti**, il che significa che in un dato nodo alcune mosse saranno immediatamente potate senz'altra considerazione. Del resto, è chiaro che la maggior parte degli scacchisti umani, in una determinata posizione, considera solo poche mosse (almeno a livello consci). Sfortunatamente quest'approccio è piuttosto pericoloso, perché non c'è alcuna garanzia di non potare la mossa migliore. Un simile errore avrebbe conseguenze disastrose se si dovesse verificare vicino alla radice dell'albero di ricerca, perché in tal caso il programma non vedrebbe mosse “ovvie”. La potatura in avanti può essere usata con sicurezza in alcune situazioni speciali (ad esempio quando due mosse sono simmetriche o altrimenti equivalenti, dimodoché se ne può considerare una sola) oppure in nodi molto lontani dalla radice.

Un programma che sfrutta tutte le tecniche che abbiamo descritto sarà in grado di giocare decentemente a scacchi (o ad altri giochi). Supponiamo di aver implementato una funzione di valutazione per gli scacchi, un test di taglio ragionevole con ricerca di quiescenza e una grande tabella di trasposizione. Supponiamo anche, dopo mesi di faticosa programmazione, di poter generare e valutare circa un milione di nodi al secondo su un moderno PC, cosa che secondo i tempi standard (tre minuti per mossa) ci permetterebbe di considerare più o meno 200 milioni di nodi per mossa. Il fattore di ramificazione degli scacchi in media è intorno a 35, e 35⁵ corrisponde a circa 50 milioni, quindi una ricerca minimax potrebbe guardare avanti solo cinque livelli o strati dell'albero. Benché non sia totalmente incompetente, un programma siffatto potrebbe essere facilmente battuto da un giocatore umano medio, che all'occasione può pianificare sei/otto livelli avanti. Con la ricerca alfa–beta potremmo arrivare a 10 livelli, o cinque mosse complete, che corrisponde a un livello di gioco esperto. Nel Paragrafo 6.7 descriveremo ulteriori tecniche di potatura che possono estendere la profondità effettiva della ricerca fino a circa 14 livelli. Per arrivare al livello di un grande maestro servirebbe anche una funzione di valutazione accuratamente perfezionata e un grande database di aperture e finali. Non guasterebbe avere anche a disposizione un supercomputer per eseguire il programma.

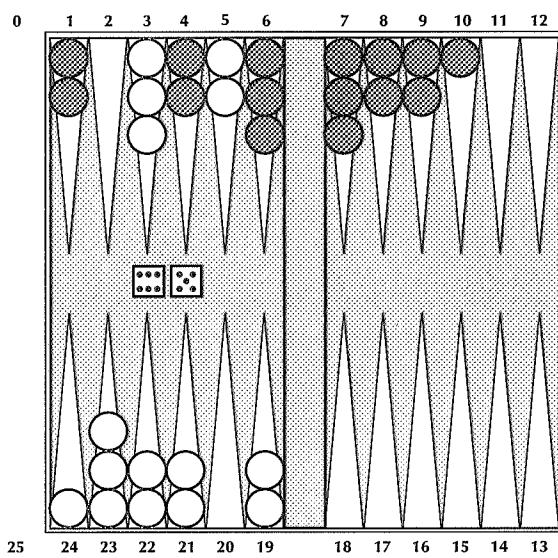


Figura 6.10 Una tipica posizione del backgammon: lo scopo del gioco è far uscire tutti i propri pezzi dal tavoliere. Il Bianco muove in senso orario verso la posizione indicata dal numero 25, il Nero in senso antiorario verso lo 0. Un pezzo può muovere in qualsiasi posizione, a meno che non contenga più pezzi avversari; se ce n'è uno solo viene catturato e deve ricominciare dall'inizio. Nella posizione indicata, il Bianco ha tirato 6–5 e deve scegliere tra quattro mosse legali: (5–10, 5–11), (5–11, 19–24), (5–10, 10–16) e (5–11, 11–16).

6.5 Giochi che includono elementi casuali

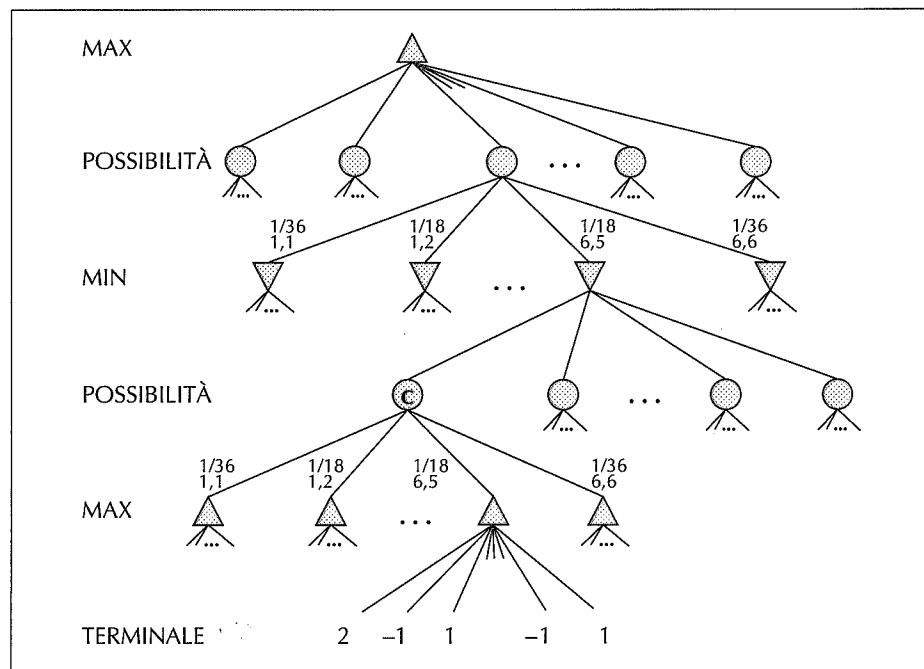
Nella vita reale, si verificano molti eventi esterni che ci mettono in situazioni impreviste. Molti giochi rispecchiano quest'imprevedibilità includendo elementi casuali, come il getto di un dado. Così facendo, compiono un passo in direzione del realismo, ed è interessante vedere come ciò influenzi il processo decisionale.

Il backgammon è un tipico gioco che combina fortuna e abilità. All'inizio di ogni turno si gettano due dadi per determinare le possibili mosse. Nella posizione illustrata nella Figura 6.10, ad esempio, il Bianco ha appena tirato 6–5 e ha quattro mosse possibili.

Benché il Bianco conosca le proprie mosse legali, non può sapere il risultato dei dadi del Nero e quindi neppure le sue potenziali mosse. Questo significa che non può costruire un albero di gioco standard come quelli che abbiamo visto per gli scacchi e il tic-tac-toe. Oltre ai nodi MAX e MIN, un albero di gioco per il backgammon deve includere nodi di possibilità (nella Figura 6.11 sono rappresentati

nodi di possibilità

Figura 6.11
Uno schema di
albero di gioco
per il back-
gammon.



come cerchi). I rami uscenti da ogni nodo di possibilità rappresentano i diversi esiti dei dadi; ognuno è etichettato con la configurazione dei dadi e la rispettiva probabilità. Ci sono 36 modi di tirare due dadi, tutti equiprobabili; dato però che ai fini del gioco 6–5 è lo stesso di 5–6, i possibili tiri distinti diventano 21. Ogni tiro doppio (da 1–1 a 6–6) ha una probabilità di verificarsi di 1/36, gli altri 15 tiri hanno ognuno una probabilità di 1/18.

Il passo successivo è capire come prendere le decisioni corrette. Naturalmente vogliamo ancora scegliere la mossa che porta alla posizione migliore. Ora però le posizioni non hanno più valori minimax definiti: al loro posto possiamo solo calcolare il valore atteso, la cui stima prende in considerazione tutti i possibili tiri di dado. Questo ci porta a generalizzare quello che per i giochi deterministici era il valore minimax in un valore expectiminimax applicabile ai giochi con nodi di possibilità. I nodi terminali, quelli MAX e quelli MIN (per cui è noto il tiro dei dadi) sono esattamente identici ai precedenti; quelli di possibilità sono valutati prendendo la media pesata dei valori risultanti da tutti i possibili tiri di dado. Quindi

EXPECTIMINIMAX(n) =

$$\left\{ \begin{array}{ll} \text{UTILITÀ}(n) & \text{se } n \text{ è uno stato terminale} \\ \text{Max}_{s \in \text{Successori}(n)} \text{EXPECTIMINIMAX}(s) & \text{se } n \text{ è un nodo MAX} \\ \text{Min}_{s \in \text{Successori}(n)} \text{EXPECTIMINIMAX}(s) & \text{se } n \text{ è un nodo MIN} \\ \sum_{s \in \text{Successori}(n)} P(s) \cdot \text{EXPECTIMINIMAX}(s) & \text{se } n \text{ è un nodo di possibilità} \end{array} \right.$$

dove la funzione successore di un nodo di possibilità n aggiunge semplicemente allo stato ogni possibile tiro dei dadi per produrre ogni successore s e $P(s)$ è la probabilità che quel tiro si verifichi. Queste equazioni si possono riportare ricorsivamente verso l'alto per tutto l'albero fino alla radice, proprio come con il minimax. Lasciamo come esercizio lo svolgimento dei dettagli dell'algoritmo.

Valutazione della posizione nei giochi con nodi di possibilità

Come nel caso del minimax, l'approssimazione più ovvia con expectiminimax è di tagliare la ricerca a un certo livello applicando una funzione di valutazione a ogni foglia. Si potrebbe pensare che le funzioni di valutazione per giochi come il backgammon siano analoghe a quelle degli scacchi: il loro compito sarebbe quindi semplicemente quello di assegnare valori più alti alle posizioni migliori. In realtà, la presenza dei nodi di possibilità modifica lo stesso significato delle valutazioni. La Figura 6.12 illustra cosa accade: con una funzione di valutazione che assegna alle foglie i valori $[1, 2, 3, 4]$, la mossa migliore è A_1 ; con i valori $[1, 20, 30, 400]$, la mossa migliore diventa A_2 . Quindi il programma si comporta in modo totalmente diverso se solo cambiamo la scala dei valori di valutazione! Per evitare questa dipendenza dalla scala la funzione di valutazione dev'essere una trasformazione lineare positiva della probabilità di vincere partendo da una determinata posizione (o, più in generale, dell'utilità attesa della posizione).

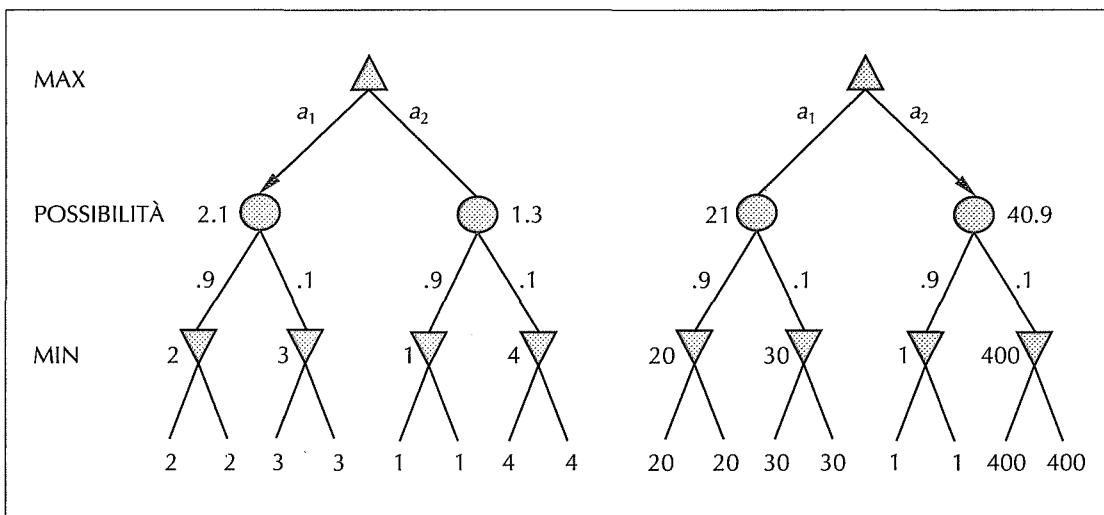


Figura 6.12 Una trasformazione dei valori delle foglie che ne preserva l'ordinamento modifica tuttavia la scelta della mossa migliore.

La complessità di expectiminimax

Se il programma conoscesse in anticipo tutti i tiri di dado di una partita, una soluzione per i giochi che li utilizzano si potrebbe ottenere allo stesso modo che per quelli che non hanno elementi casuali, cosa che minimax fa in un tempo $O(b^m)$. Dato che expectiminimax considera anche tutte le possibili sequenze di tiri, la complessità sale a $O(b^m n^m)$, dove n è il numero di tiri di dado distinti.

Anche limitando la profondità della ricerca a un livello d'abbastanza piccolo, il costo aggiuntivo rispetto a quello di minimax rende poco realistica la possibilità di guardare molto avanti nei giochi con elementi casuali. Nel backgammon n vale 21 e b solitamente è intorno a 20, ma in alcune situazioni può anche arrivare a 4000 quando si ottiene lo stesso risultato con entrambi i dadi (cosa che nel backgammon consente di muovere quattro volte anziché due). Con tutta probabilità, non sarà possibile guardare oltre il terzo livello dell'albero.

Un altro modo di esprimere questo concetto è il seguente: il vantaggio di alfa–beta è che permette di ignorare sviluppi futuri che, dato uno svolgimento ottimo della partita, sono semplicemente impossibili; in questo modo la ricerca si può concentrare sulle mosse future più probabili. Nei giochi basati sui dadi, *non esistono* sequenze di mosse probabili, perché la legalità di ogni mossa dipende unicamente dall'esito casuale del tiro. Questo è un problema generale ogni volta che entra in gioco l'incertezza: le possibilità si moltiplicano enormemente e diventa inutile formulare piani d'azione dettagliati, perché il mondo li renderà con ogni probabilità irrealizzabili.

Senza dubbio qualche lettore avrà pensato che forse è possibile applicare tecniche come la potatura alfa–beta agli alberi di gioco con nodi di possibilità: in effetti si può. Per i nodi MIN e MAX l'analisi non cambia, ma con un po' d'ingegno possiamo potare anche qualche nodo di possibilità. Considerate il nodo C nella Figura 6.11 e quello che succede al suo valore quando esaminiamo e valutiamo i suoi figli. Ricordando che questo è ciò che serve alla tecnica alfa–beta per potare un sottoalbero, è possibile trovare un limite superiore al valore di C prima di considerare tutti i suoi nodi figli? A prima vista potrebbe sembrare impossibile, perché il valore di C è la *media* del valore dei nodi figli: finché non abbiamo considerato tutti i possibili esiti dei dadi, la media potrebbe avere qualsiasi valore, così come i figli non esaminati. Ma se limitiamo i possibili valori della funzione di utilità, possiamo derivare dei limiti anche per la media. Ad esempio, se diciamo che i valori di utilità sono sempre compresi tra +3 e -3, il valore delle foglie risulta vincolato, e di conseguenza *possiamo* derivare un limite superiore al valore di un nodo di possibilità senza considerare tutti i suoi figli.

Giochi di carte

I giochi di carte sono interessanti per molte ragioni oltre che per la loro correlazione con il gioco d'azzardo. Tra la grandissima varietà di giochi esistenti, ci focalizzeremo su quelli in cui le carte sono distribuite a caso all'inizio del gioco e la mano di ogni giocatore è nascosta agli altri. Questa categoria include bridge, whist, hearts (noto in alcune regioni italiane come "la peppa") e alcuni tipi di poker.

A prima vista potrebbe sembrare che i giochi di carte siano molto simili a quelli che usano i dadi: le carte sono distribuite casualmente e determinano le mosse consentite a ogni giocatore, ma è come se all'inizio della partita i dadi siano già stati tirati tutti! Approfondiremo più avanti questa considerazione, che si rivelerà di notevole uso pratico. Tuttavia è anche sbagliata, per ragioni molto interessanti.

Immaginiamo due giocatori, MAX e MIN, che per impraticarsi giocano a bridge a carte scoperte. Per semplicità supporremo che ogni giocatore ne riceva solo quattro. Le carte sono distribuite come segue, e MAX è il primo a giocare:

MAX : ♡ 6 ♦ 6 ♣ 9 8 MIN: ♡ 4 ♠ 2 ♢ 10 5 .

Supponiamo che MAX attacchi con ♣ 9. MIN deve rispondere con lo stesso seme, giocando ♣ 10 o ♣ 5. MIN gioca ♣ 10 e vince la presa. Ora MIN deve giocare e prosegue con ♠ 2. MAX non ha picche (e non può vincere la presa), quindi deve scartare una carta. La scelta naturale è ♦ 6, dato che le altre due sono vincenti. Ora, indipendentemente dalla carte giocate in seguito da MIN, MAX vincerà entrambe le mani rimaste e la partita finirà in parità, due prese a due. È facile mostrare, usando una variante adeguata di minimax (Esercizio 6.12), che l'apertura di MAX con ♣ 9 è effettivamente la scelta ottima.

Ora modifichiamo la mano di MIN, sostituendo ♡ 4 con ♦ 4 :

MAX : ♡ 6 ♦ 6 ♣ 9 8 MIN: ♦ 4 ♠ 2 ♢ 10 5 .

I due casi sono del tutto simmetrici: il gioco sarà identico, tranne che nella seconda mano MAX scarterà ♡ 6. Ancora una volta la partita finirà in parità con due prese a testa, e l'apertura con ♣ 9 è una scelta ottima.

Fin qui tutto bene. Ora nascondiamo una delle carte di MIN: MAX sa che l'avversario ha ricevuto la prima mano (quella con ♡ 4) oppure la seconda (con ♦ 4), ma non sa quale delle due. MAX ragiona come segue:

♣ 9 è la scelta ottima contro la prima mano di MIN e anche contro la seconda, per cui dev'essere ottima anche adesso, visto che MIN ha ricevuto una delle due.

Più generalmente, MAX sta usando una tecnica che potremmo chiamare "fare la media sulla chiaroveggenza". L'idea è valutare una possibile azione determinando per prima cosa il suo valore minimax per ogni possibile configurazione di carte, per calcolare poi il valore atteso su tutte le configurazioni in base alle loro rispettive probabilità.

Se pensate che questo ragionamento abbia senso (o non avete capito nulla perché non conoscete il bridge), considerate la seguente storiella:

Giorno 1: la strada *A* porta a un pugno di monete d'oro; la strada *B* porta a un bivio.

Al bivio andate a sinistra e troverete un mucchio di gioielli, se andate a destra invece sarete investiti da un autobus.

Giorno 2: la strada *A* porta a un pugno di monete d'oro; la strada *B* porta a un bivio.

Al bivio andate a destra e troverete un mucchio di gioielli, se andate a sinistra invece sarete investiti da un autobus.

Giorno 3: la strada *A* porta a un pugno di monete d'oro; la strada *B* porta a un bivio.

Al bivio indovinate la strada giusta e troverete un mucchio di gioielli, però se sbagliate a indovinare sarete investiti da un autobus.

È chiaro che, i primi due giorni, non è irragionevole scegliere la strada *B*. Nessuna persona dotata di buonsenso, comunque, la sceglierrebbe anche il terzo giorno. Eppure questo è precisamente quanto consiglia la media sulla chiaroveggenza: la strada *B* è la scelta ottima nelle situazioni del Giorno 1 e del Giorno 2; quindi dev'essere per forza ottima anche il terzo giorno, visto che corrisponde per forza a uno dei primi due. Torniamo al nostro gioco di carte: dopo che MAX attacca con ♠ 9, MIN vince con ♠ 10. Come prima, MIN nella mano successiva gioca ♠ 2, e ora MAX si trova a un bivio senza nessuna istruzione. Se MAX getta via ♥ 6 e MIN ha ancora ♥ 4, quest'ultima diventa una carta vincente e MAX perde la partita. In modo analogo, se MAX scarta ♦ 6 e MIN ha ancora ♦ 4, MAX perde. Quindi, giocare per primo ♠ 9 porta a una situazione in cui MAX ha il 50% di possibilità di perdere: sarebbe stato molto meglio giocare per primi ♥ 6 e ♦ 6, assicurandosi così il pareggio.

La lezione da trarre da tutto questo è che quando manca informazione, si deve considerare *quanta informazione sarà disponibile* in ogni punto del gioco. Il problema dell'algoritmo di MAX è che dà per scontato che, per ogni configurazione delle carte, il gioco procederà come *se fossero tutte visibili*. Come il nostro esempio dimostra, questo porta MAX ad agire come se tutte le incertezze *future* potranno essere risolte al momento opportuno. Inoltre l'algoritmo di MAX non deciderà mai di *raccogliere* informazione (o *fornirla* a un eventuale compagno), perché una volta fissata una particolare configurazione di carte non ce n'è bisogno: eppure nei giochi come il bridge spesso è una buona idea giocare una carta che aiuti a determinare quelle possedute dell'avversario, o che aiuti il compagno a capire le proprie. Questo tipo di comportamenti è generato automaticamente da un algoritmo ottimo per giochi a informazione imperfetta. Un algoritmo siffatto non esegue la ricerca nello spazio degli stati del mondo (mani di carte), ma in quello degli stati-credenza (credenze su chi ha quali carte, con le relative probabilità). Dopo aver presentato le necessarie nozioni probabilistiche saremo in grado di presentare l'algoritmo nel modo appropriato nel Capitolo 17, in cui elaboreremo anche un altro punto molto importante: nei giochi a informazione imperfetta è meglio fornire all'avversario la minor quantità di informazione possibile, e spesso il modo migliore di farlo è di agire *in modo impredicibile*. Ecco perché, nei ristoranti, le verifiche degli ispettori dell'igiene si verificano senza preavviso.

6.6 Lo stato dell'arte dei programmi di gioco

Si potrebbe dire che i giochi stanno all'IA come la Formula 1 all'industria automobilistica: i programmi allo stato dell'arte sono macchine velocissime e incredibilmente curate, che incorporano tecniche di ingegneria molto avanzate, ma non vanno bene per uscire a fare la spesa. Benché alcuni ricercatori li ritengano abbastanza irrilevanti per il resto dell'IA, i giochi continuano a generare non solo molta eccitazione ma anche un flusso costante di innovazioni che vengono poi adottate dall'intera comunità.

Scacchi: nel 1957, Herbert Simon predisse che entro 10 anni un computer avrebbe battuto il campione del mondo in carica. Quarant'anni dopo, il programma Deep Blue sconfisse Garry Kasparov in un incontro-esibizione della durata di sei partite: Simon si sbagliò, ma solo di un fattore 4. Kasparov scrisse:

scacchi

La partita decisiva del match fu la seconda, che lasciò una cicatrice nella mia memoria... abbiamo visto qualcosa che andava ben oltre le nostre aspettative circa la capacità del computer di prevedere le conseguenze posizionali a lungo termine delle sue decisioni. La macchina rifiutò di muoversi in una posizione che comportava un decisivo vantaggio a breve termine, mostrando un senso del pericolo davvero umano. (Kasparov, 1997)

Deep Blue fu sviluppato da Murray Campbell, Feng-Hsiung Hsu e Joseph Hoane all'IBM (v. Campbell et al., 2002), elaborando il progetto Deep Thought già sviluppato da Campbell e Hsu alla Carnegie Mellon. La macchina vittoriosa era un computer parallelo con 30 processori IBM RS/6000 che eseguivano la "ricerca software" e 480 processori VLSI appositamente specializzati per generare (e ordinare) le mosse, eseguire la "ricerca hardware" negli ultimi livelli dell'albero e valutare i nodi foglia. Deep Blue considerava in media 126 milioni di nodi al secondo, con una velocità massima di 330 milioni di nodi/sec. Per ogni mossa poteva generare 30 miliardi di posizioni, raggiungendo normalmente una profondità di 14. Il cuore del programma era costituito da una procedura di ricerca alfa-beta standard ad approfondimento iterativo dotata di una tabella di trasposizione, ma la chiave del suo successo sembra essere stata la capacità di generare estensioni oltre il limite di profondità per sequenze di mosse forzate sufficientemente interessanti. In alcuni casi, la ricerca poteva raggiungere una profondità di 40 livelli. La funzione di valutazione considerava oltre 8000 caratteristiche, molte delle quali rivolte a specifiche configurazioni di pezzi. Tra le altre cose veniva usato un "libro di aperture" di circa 4000 posizioni e un database di 700.000 partite di grandi maestri da cui si potevano estrarre indicazioni. Il sistema sfruttava anche un grande database di finali risolti, comprendente tutte le posizioni con cinque pezzi e molte con sei pezzi. Quest'ultimo database aveva l'effetto di estendere sensibilmente la profondità effettiva della ricerca, permettendo in certi casi a Deep Blue di giocare in modo perfetto anche quando lo scacco matto era molto distante.

Il successo di Deep Blue rinforzò la credenza molto diffusa che il progresso nell'esecuzione di giochi da parte dei computer scaturisse principalmente dall'uso di hardware sempre più potente: un punto di vista che naturalmente IBM voleva incoraggiare. I creatori di Deep Blue, d'altra parte, hanno affermato che le estensioni alla ricerca e la funzione di valutazione furono aspetti ugualmente cruciali (Campbell et al., 2002). Inoltre, sappiamo che diversi recenti miglioramenti algoritmicamente hanno permesso a programmi in esecuzione su PC standard di vincere ogni campionato del mondo di scacchi per computer dal 1992 a oggi, spesso sconfiggendo avversari massivamente paralleli che potevano considerare un numero di nodi 1000 volte superiore. Sono state usate varie euristiche di potatura per ridurre il fattore di ramificazione effettivo a meno di 3 (quello reale è di circa 35). Tra queste la più importante è l'euristica della **mossa nulla**, che genera un buon limite inferiore del valore di una posizione usando una ricerca poco profonda in cui l'avversario comincia muovendo due volte. Questo limite inferiore permette spesso di applicare la potatura alfa–beta senza la spesa di una ricerca completa. Un'altra tecnica importante è la **potatura di futilità**, che aiuta a decidere in anticipo quali mosse causeranno un taglio beta nei nodi successori.

Gli sviluppatori di Deep Blue rifiutarono di concedere una rivincita a Kasparov. La più recente competizione di alto livello vide, nel 2002, il programma FRITZ contrapposto al campione del mondo Vladimir Kramnik. L'incontro, della durata di otto partite, terminò con un pareggio. Le condizioni dell'incontro erano molto favorevoli al contendente umano e l'hardware era costituito da un normale PC, non un supercomputer. Kramnik commentò l'incontro dicendo che “è ormai chiaro che il miglior programma e il campione del mondo giocano allo stesso livello”.

Dama: a partire dal 1952 Arthur Samuel, nel tempo libero lasciatogli dal suo lavoro all'IBM, sviluppò un programma per giocare a dama capace di apprendere la propria funzione di valutazione giocando migliaia di volte contro se stesso (descriveremo quest'idea dettagliatamente nel Capitolo 21, nel 2° volume). Il programma di Samuel inizialmente giocava come un principiante, ma dopo solo pochi giorni di addestramento contro se stesso aveva migliorato il proprio livello fino a superare quello dello stesso Samuel (che, a dire il vero, non era un giocatore molto forte). Nel 1962 sconfisse Robert Nealy, un campione della cosiddetta “dama cieca”, grazie a un errore di quest'ultimo. Molti credettero che questo significasse che i computer avevano superato gli esseri umani nel gioco della dama, ma non era così. In ogni caso, considerando che il computer di Samuel (un IBM 704) aveva solo 10.000 parole di memoria, nastro magnetico come memoria di massa, e un processore da 0,000001 GHz, quella vittoria rimane un grande risultato.

Pochi altri cercarono di fare meglio finché Jonathan Schaeffer e i suoi colleghi svilupparono Chinook, un programma che gira su normali PC e utilizza la ricerca alfa–beta. Chinook usa un database precalcolato di 444 miliardi di posizioni con otto pezzi sulla scacchiera o meno, in modo da rendere impeccabile il suo gioco nei finali. Chinook si classificò secondo agli U.S. Open del 1990, guadagnan-

mossa nulla

potatura di futilità

dama

dosi così il diritto di sfidare il campione del mondo. A questo punto si dovette scontrare con un grosso problema, che aveva il nome di Marion Tinsley. Il dr. Tinsley era stato campione del mondo di dama per più di 40 anni, perdendo in tutto quel tempo soltanto tre partite. Nel primo incontro con Chinook, Tinsley dovette subire la quarta e la quinta sconfitta, ma alla fine vinse 20.5 a 18.5. La rivincita dell'agosto 1994 finì prematuramente quando Tinsley si dovette ritirare per ragioni di salute: Chinook divenne così campione del mondo ufficiale.

Schaeffer ritiene che, con una sufficiente potenza di calcolo, il database di finali potrà essere allargato fino al punto che una ricerca in avanti dalla posizione iniziale potrà sempre raggiungere una configurazione nota della scacchiera: in altre parole, il gioco della dama risulterebbe completamente risolto (già oggi, è capitato che Chinook annunciasse una vittoria dopo solo 5 mosse). Questo tipo di analisi esaustiva può essere fatto a mano per il tic-tac-toe 3×3 ed è stato effettuato, con l'aiuto del computer, anche per Cubic (tic-tac-toe $4 \times 4 \times 4$), Go-Moku (5 pedine in fila su una tavola da Go) e il "Nine-Men's Morris", noto anche come filetto italiano o "Mulino" (Gasser, 1998). Con un notevole lavoro, Ken Thompson e Lewis Stiller (1992) hanno risolto tutti i finali degli scacchi con cinque pezzi e alcuni con sei, rendendoli disponibili su Internet. Stiller ha anche scoperto una configurazione di pezzi in cui un giocatore può forzare il matto, ma solo in 262 mosse: purtroppo, le regole degli scacchi impongono che un giocatore possa ottenere la patta se "non succede nulla" per 50 mosse.

Othello: chiamato anche Reversi, è probabilmente più popolare come gioco per computer che nella sua versione da tavolo. Lo spazio di ricerca è più piccolo di quello degli scacchi, dato che solitamente il numero delle mosse legali va da 5 a 15, ma è stato necessario sviluppare da zero la teoria alla base della valutazione delle posizioni. Nel 1997, il programma Logistello (Buro, 2002) ha sconfitto sei a zero il campione del mondo umano, Takeshi Murakami. Oggi è generalmente riconosciuto che, per quanto riguarda Othello, gli esseri umani non possono competere con i computer.

Othello

Backgammon: nel Paragrafo 6.5 abbiamo visto come la presenza dell'incertezza legata ai tiri di dado renda impossibile eseguire una ricerca profonda. La maggior parte dei lavori dedicati al backgammon si sono dedicati al miglioramento della funzione di valutazione. Gerry Tesauro (1992) ha unito il metodo di apprendimento per rinforzo di Samuel con tecniche di reti neurali (v. Capitolo 20, nel 2º vol.) per sviluppare un valutatore di posizioni straordinariamente accurato, che può essere applicato a una ricerca di profondità 2 o 3. Dopo aver giocato più di un milione di partite di addestramento contro se stesso il programma di Tesauro, TD-GAMMON, viene oggi considerato uno dei tre giocatori più forti del mondo. Le opinioni del programma sulle aperture, in certi casi, hanno modificato radicalmente la teoria moderna del backgammon.

backgammon

Go: è il gioco da tavolo più popolare dell'Asia, e richiede ai professionisti una disciplina almeno pari a quella necessaria per gli scacchi. Dato che la plancia ha una misura 19×19 , il fattore di ramificazione iniziale è 361, decisamente troppo per i

Go

metodi di ricerca tradizionali. Fino al 1997 non esisteva neppure un programma competente, ma i software più recenti giocano spesso mosse rispettabili. La maggior parte dei programmi uniscono tecniche di riconoscimento di pattern (del tipo: “quando compare la seguente configurazione di pezzi, considera questa mossa”) con una forma limitata di ricerca per decidere quali pezzi possono essere catturati, rimanendo in un’area circoscritta del tavoliere. Mentre scriviamo, i programmi più forti sono probabilmente Goemate di Chen Zhixing e Go4++ di Michael Reiss, valutati entrambi intorno ai 10 kyu (dilettanti un po’ scarsi). Il gioco del Go è un’area di ricerca che potrebbe trarre benefici da un’approfondita investigazione che utilizzi i metodi di ragionamento più sofisticati. Il successo potrebbe scaturire da nuove modalità di integrazione tra le diverse linee di ragionamento riguardanti i molti “sottogiochi” debolmente interconnessi in cui si può scomporre una partita. Tecniche simili sarebbero di grandissima utilità per i sistemi intelligenti in generale.

bridge

Bridge: è un gioco a informazione imperfetta; le carte di ogni giocatore sono nascoste a tutti gli altri. Il bridge è anche *multigiocatore* e richiede quattro partecipanti anziché due, benché questi debbano formare due squadre. Come abbiamo visto nel Paragrafo 6.5, per giocare a bridge in modo ottimo si devono considerare elementi come la raccolta di informazioni, la comunicazione, il bluff e la valutazione accurata delle probabilità. Molte di queste tecniche sono incluse nel programma Bridge Baron™ (Smith et al., 1998), che vinse il campionato mondiale per computer nel 1997. Benché non giochi in modo ottimo, Bridge Baron è uno dei pochi sistemi di successo in grado di formulare piani complessi e gerarchici (v. Capitolo 12) che prendono in considerazione idee familiari ai giocatori di bridge come il *finessing* e lo *squeezing*.

Il programma GIB (Ginsberg, 1999) ha vinto il campionato 2000 in modo netto. GIB usa il metodo della “media sulla chiaroveggenza” con due modifiche fondamentali. Innanzitutto, invece di valutare ogni scelta per tutte le possibili configurazioni delle carte nascoste (che possono arrivare a 10 milioni) prende in esame un campione casuale di 100 configurazioni. In secondo luogo, GIB utilizza una tecnica chiamata **generalizzazione basata sulla spiegazione** per calcolare e tenere in memoria regole generali di gioco ottimo in varie categorie standard di situazioni. Questo gli permette di risolvere ogni mano *in modo esatto*. L’accuratezza tattica di GIB controbilancia la sua incapacità di ragionare sull’informazione. GIB si è classificato 12° su 35 in una gara individuale nella quale è stato impegnato nella sola fase di gioco delle carte (il cosiddetto “par contest”). Questa sfida ha avuto luogo in occasione del campionato mondiale del 1998 per esseri umani, e il risultato ottenuto da GIB è andato ben oltre le aspettative degli esperti.

6.7 Discussione

Dal momento che il calcolo delle decisioni ottime ha una complessità intrattabile per la gran parte dei giochi, gli algoritmi devono adottare ipotesi restrittive e accontentarsi di risultati approssimati. L'approccio standard, basato su minimax, funzioni di valutazione e potatura alfa–beta, è solo uno dei modi di farlo. Probabilmente perché fu proposto per primo, l'approccio standard è stato sviluppato a fondo e domina gli altri metodi nell'applicazione ai tornei. Alcuni ritengono che questo abbia causato un distacco tra la ricerca sui giochi e quella dedicata al resto dell'IA, perché il metodo standard non offre più molto spazio per nuove soluzioni applicabili al processo decisionale in generale. In questo paragrafo esamineremo le alternative.

Consideriamo prima di tutto minimax. Questa tecnica seleziona una mossa ottima in un dato albero di ricerca *a patto che le valutazioni dei nodi foglia siano esattamente corrette*. Nella realtà, le valutazioni sono spesso delle stime approssimate del valore di una posizione e possono includere un errore anche grande. La Figura 6.13 mostra un albero di gioco a due livelli per cui minimax sembra inappropriato: infatti suggerisce di prendere il ramo di destra, mentre è probabile che il valore reale del ramo sinistro sia molto superiore. La scelta si basa sulla certezza che *tutti* i nodi etichettati con i valori 100, 101, 102 e 100 siano *effettivamente* preferibili al nodo che vale 99. Tuttavia, il fatto che quest'ultimo abbia dei nodi fratelli che valgono 1000 suggerisce che in realtà potrebbe avere un valore molto più alto. Un modo di gestire questo problema è usare una valutazione che restituiscce una *distribuzione di probabilità* dei valori possibili. A questo punto si potrebbe calcolare la distribuzione di probabilità del nodo padre usando tecniche consolidate di calcolo statistico. Sfortunatamente, i valori di un gruppo di nodi fratelli hanno solitamente un alto grado di correlazione, cosicché questo calcolo può risultare molto costoso, dal momento che richiede informazione difficile da ottenere.

Ora prendiamo in esame l'algoritmo di ricerca che genera l'albero: lo scopo del progettista di un algoritmo è ottenere una computazione veloce che ha come risultato una buona mossa. Il problema principale di alfa–beta sta nel fatto che è

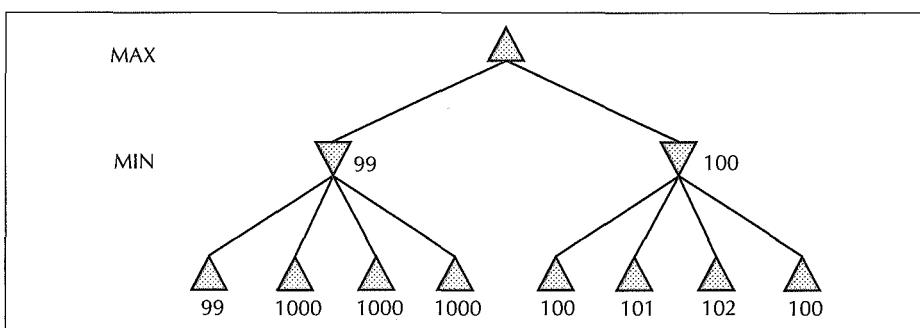


Figura 6.13
Un albero di gioco a due livelli per cui minimax potrebbe essere inappropriato.

progettato non solo per selezionare una buona mossa, ma anche per calcolare i limiti sui valori di tutte le mosse legali. Per vedere perché quest'informazione aggiuntiva è inutile, considerate una posizione in cui la mossa legale è una sola: anche in questo caso la ricerca alfa–beta continuerà a generare e a valutare un grande albero totalmente inutile. Naturalmente possiamo inserire nell'algoritmo un apposito test, ma questo non fa che nascondere il problema di base, e cioè che gran parte dei calcoli eseguiti da alfa–beta sono, nella pratica, irrilevanti. Il caso in cui si ha una sola mossa legale non è molto diverso da quello in cui una mossa va bene e tutte le altre sono paleamente disastrose. In una situazione come questa, che presenta una mossa "chiaramente preferibile", sarebbe meglio arrivare a una decisione rapida dopo una piccola quantità di ricerca e non sprecare tempo che potrebbe essere usato in modo produttivo più tardi, su una posizione problematica. Questo porta al concetto di *utilità dell'espansione di un nodo*. Un buon algoritmo di ricerca dovrebbe selezionare le espansioni di maggiore utilità, ovvero quelle che porteranno più probabilmente alla scoperta di una mossa sensibilmente migliore. Se non ci sono più espansioni di nodi la cui utilità è superiore al loro costo (in termini di tempo impiegato), l'algoritmo dovrebbe interrompere la ricerca e muovere. Notate che questo funziona non solo nel caso in cui ci siano mosse chiaramente preferibili ma anche quando le mosse sono *simmetriche*, tali che non si possa in nessun caso stabilire quale sia la migliore.

Questo tipo di ragionamento riguardante la computazione stessa prende il nome di **metaragionamento**, e non si applica solo ai giochi. Tutti i calcoli sono eseguiti al fine di raggiungere decisioni valide, tutti hanno un costo, e ognuno ha una certa probabilità di avere come risultato un miglioramento della qualità della decisione. Alfa–beta incorpora il tipo più semplice di metaragionamento: un teorema che afferma che alcuni rami dell'albero possono essere ignorati senza perdere qualità della ricerca.

Infine, riesaminiamo la stessa natura della ricerca. Gli algoritmi per la ricerca euristica e per i giochi funzionano generando sequenze di stati concreti, partendo da uno stato iniziale e applicando una funzione di valutazione. Chiaramente, non è così che giocano gli esseri umani. Negli scacchi spesso il giocatore sceglie un particolare obiettivo, come ad esempio intrappolare la regina avversaria, e partendo da quest'obiettivo genera *selettivamente* piani plausibili per ottenerlo. Questo tipo di **ragionamento guidato dagli obiettivi**, o **pianificazione**, talvolta permette di eliminare del tutto la ricerca combinatoria (v. Parte IV). Il programma PARADISE, di David Wilkins (1980), è l'unico ad aver usato con successo il ragionamento guidato dagli obiettivi applicato agli scacchi: il software è stato capace di risolvere problemi che richiedevano combinazioni di 18 mosse. A tutt'oggi non è chiaro come si possono *combinare* i due tipi di algoritmi in un sistema robusto ed efficiente, anche se Bridge Baron sembra essere un passo nella giusta direzione. Un sistema pienamente integrato rappresenterebbe un risultato importante non solo per la ricerca dedicata ai giochi ma anche per l'intera IA, perché sarebbe una buona base per un agente intelligente in senso generale.

6.8 Riepilogo

Abbiamo esaminato una varietà di giochi per comprendere cosa significa “giocare in modo ottimo” e per capire come si possono realizzare in pratica programmi che giocano bene. I concetti più importanti sono i seguenti.

- ◆ Un gioco può essere definito specificando lo **stato iniziale** (configurazione del tavoliere all'inizio della partita), le **azioni legali** in ogni stato, un **test di terminazione** (che determina quando la partita è finita) e una **funzione di utilità** che si applica agli stati terminali.
- ◆ Nei giochi a due partecipanti, a somma zero e con **informazione perfetta**, l'algoritmo **minimax** può scegliere le mosse ottime eseguendo un'enumerazione in profondità dell'albero di gioco.
- ◆ L'algoritmo di ricerca **alfa–beta** calcola le stesse mosse ottime di minimax, ma ha un'efficienza migliore perché elimina sottoalberi che sono con tutta probabilità irrilevanti.
- ◆ Normalmente, è impossibile (anche per la ricerca alfa–beta) prendere in considerazione l'intero albero di gioco: di conseguenza, diventa necessario tagliare la ricerca a un certo punto e applicare una **funzione di valutazione** che fornisce la stima dell'utilità di uno stato.
- ◆ I giochi che includono componenti casuali possono essere gestiti con un'estensione dell'algoritmo minimax che valuta un **nodo di possibilità** prendendo l'utilità media di tutti i nodi figli, pesata in base alla probabilità di ognuno di essi.
- ◆ Nei giochi a **informazione imperfetta**, come il bridge, per giocare in modo ottimo bisogna ragionare sugli **stati-credenza** presenti e futuri di ogni giocatore. Un'approssimazione si può ottenere semplicemente facendo la media di ogni azione per ogni possibile configurazione dell'informazione mancante.
- ◆ I migliori programmi oggi disponibili possono giocare alla pari o battere i campioni umani di dama, Othello e backgammon, e sono molto vicini al loro livello nel bridge. In un incontro speciale, un programma ha battuto il campione mondiale di scacchi. Per quanto riguarda il Go, sono ancora fermi a un livello amatoriale.

Note storiche e bibliografiche

La storia degli sistemi automatici di gioco è segnata da numerose frodi. La più nota è quella del barone Wolfgang von Kempelen (1734-1804), creatore del “Turco”, un automa giocatore di scacchi che sconfisse anche Napoleone prima che venisse svelato il trucco: una scatola magica da prestigiatore celava uno scacchista umano (v. Levitt, 2000). In ogni caso, l'automa “funzionò” dal 1769 al 1854. Sembra che

nel 1846 Charles Babbage (che era stato affascinato dal Turco) sia stato uno dei primi autori a contribuire a una discussione seria sulla possibilità che delle macchine possano giocare a scacchi e dama (Morrison e Morrison, 1961). Babbage inoltre progettò, ma non costruì mai, una macchina speciale per giocare a tic-tac-toe. La prima vera macchina in grado di giocare autonomamente fu costruita intorno al 1890 dall'ingegnere spagnolo Leonardo Torres y Quevedo. Era specializzata nel finale scacchistico di re e torre contro re, e garantiva la vittoria del giocatore con la torre partendo da qualsiasi posizione.

L'algoritmo minimax si fa spesso risalire a un articolo pubblicato nel 1912 da Ernst Zermelo, il creatore della moderna teoria degli insiemi. Sfortunatamente l'articolo conteneva diversi errori e non spiegava correttamente il funzionamento di minimax. L'importantissimo lavoro *Theory of Games and Economic Behavior* (von Neumann e Morgenstern, 1944) pose una solida base per lo sviluppo della teoria dei giochi: tra l'altro includeva un'analisi del fatto che alcuni giochi richiedono strategie casuali (o comunque impredicibili).

Negli anni in cui si stavano sviluppando i primi computer, molte figure di spicco erano interessate agli scacchi. Konrad Zuse (1945), la prima persona a progettare un computer programmabile, elaborò idee abbastanza dettagliate sull'argomento. L'influente libro *Cybernetics* di Norbert Wiener (1948) discuteva un possibile progetto di programma scacchistico che comprendeva i concetti di ricerca minimax, taglio della ricerca a una certa profondità e funzione di valutazione. Claude Shannon (1950) espresse i principî alla base dei programmi moderni con un dettaglio molto maggiore di Wiener, introducendo la ricerca di quiescenza e descrivendo alcune idee per una ricerca dell'albero di gioco selettiva e non esaustiva. Slater (1950) e i commentatori del suo articolo fornirono un ulteriore contributo all'esplorazione delle possibilità scacchistiche dei computer. In particolare, I. J. Good (1950) sviluppò la nozione di quiescenza indipendentemente da Shannon.

Nel 1951, Alan Turing scrisse il primo programma per computer capace di giocare una partita intera di scacchi (v. Turing et al., 1953). Ma il programma di Turing non venne mai effettivamente fatto girare su un computer; fu eseguito con una simulazione manuale contro un avversario umano molto scarso, che lo sconfisse. Nel frattempo D. G. Prinz (1952) aveva scritto (ed effettivamente mandato in esecuzione) un programma che era in grado di risolvere problemi scacchistici, ma non una partita intera. Fu Alex Bernstein a scrivere il primo programma che giocava una partita intera di scacchi con tutte le regole (Bernstein e Roberts, 1958; Bernstein et al., 1958).³

³ Newell et al. (1958) fanno menzione di un programma russo, BESM, che potrebbe aver preceduto quello di Bernstein.

John McCarthy ideò la ricerca alfa–beta nel 1956, anche se non pubblicò nulla in riguardo. Il programma per giocare a scacchi NSS (Newell et al., 1958) fu il primo a utilizzare alfa–beta, in una versione semplificata. Secondo Nilsson (1971), anche il programma di Arthur Samuel per giocare a dama (Samuel, 1959, 1967) usava alfa–beta, benché Samuel non ne facesse menzione nelle pubblicazioni. Articoli su alfa–beta cominciarono a essere pubblicati nei primi anni '60 (Hart e Edwards, 1961; Brudno, 1963; Slagle, 1963b). Un'implementazione completa di alfa–beta è descritta da Slagle e Dixon (1969) in un programma per il gioco *Kalah* (noto anche come *Owari* o *Awalé*). La tecnica alfa–beta fu usata anche dal programma scacchistico “*Kotok–McCarthy*”, scritto da uno studente di John McCarthy (Kotok, 1962). Knuth e Moore (1975) offrono una storia di alfa–beta, con una prova della sua correttezza e una discussione della sua complessità temporale. La loro analisi di alfa–beta con ordinamento casuale dei successori mostrò che la complessità asintotica era $O((b/\log b)^d)$, un risultato piuttosto deludente dato che un fattore effettivo di ramificazione di $b/\log b$ non è molto inferiore allo stesso b . In seguito capirono che la formula asintotica è accurata solo per $b > 1000$ o giù di lì, laddove la spesso citata $O(b^{3d/4})$ si applica ai valori dei fattori di ramificazione che si incontrano effettivamente nei giochi. Pearl (1982b) mostra che la ricerca alfa–beta è asintoticamente ottima tra tutti gli algoritmi di ricerca con alberi di lunghezza prefissata.

Il primo incontro scacchistico tra computer vide contrapposti il programma *Kotok–McCarthy* e “*ITEP*”, scritto a metà degli anni '60 all'Istituto di Fisica Teorica e Sperimentale di Mosca (Adelson-Velsky et al., 1970). Le partite intercontinentali furono svolte con l'ausilio del telegrafo: l'incontro finì nel 1967 con la vittoria per 3–1 di *ITEP*. Il primo programma a competere con successo con scacchisti umani fu *MacHack 6* (Greenblatt et al., 1967). Il suo punteggio di circa 1400 era ben superiore a quello di un principiante, che è 1000, ma era di molto inferiore al 2800 o più che sarebbe stato necessario per realizzare la predizione di Herb Simon, che nel 1957 aveva asserito che un programma per computer sarebbe stato campione del mondo di scacchi entro 10 anni (Simon e Newell, 1958).

La competizione tra programmi scacchistici cominciò a farsi seria a partire dal primo campionato nordamericano dell'ACM, nel 1970. Nei primi anni '70 i programmi cominciarono a diventare estremamente complicati, includendo molti trucchi per eliminare rami della ricerca, generare mosse plausibili e così via. Nel 1974, il primo campionato mondiale di scacchi per computer si tenne a Stoccolma e fu vinto da *Kaissa* (Adelson-Velsky et al., 1975), un altro programma sviluppato all'*ITEP* di Mosca. *Kaissa* usava un approccio molto più semplice e diretto: una ricerca alfa–beta esaustiva unita alla ricerca di quiescenza. La preminenza di quest'approccio fu confermata dalla convincente vittoria di *CHESS 4.6* al campionato mondiale del 1977. *CHESS 4.6* esaminava fino a 400.000 posizioni per mossa e aveva un punteggio di 1900.

Una versione successiva del *MacHack 6* di Greenblatt fu il primo programma scacchistico a girare su hardware speciale appositamente sviluppato (Moussouris et

al., 1979), ma il primo programma a ottenere un certo successo grazie all'uso di hardware dedicato fu Belle (Condon e Thompson, 1982). Belle era in grado di eseguire la generazione di mosse e la valutazione delle posizioni via hardware, cosa che gli permetteva di considerare diversi milioni di posizioni per mossa. Belle raggiunse un punteggio di 2250, diventando il primo programma a fregiarsi del titolo di maestro. Il sistema HITECH, anch'esso basato su un computer appositamente configurato, fu progettato dall'ex campione mondiale di scacchi per corrispondenza Hans Berliner e dal suo studente Carl Ebeling alla CMU in modo da calcolare rapidamente le funzioni di valutazione (Ebeling, 1987; Berliner e Ebeling, 1989). Con i suoi 10 milioni di posizioni generate per mossa, HITECH divenne campione del Nord America tra i computer nel 1985 e fu il primo programma a sconfiggere un grande maestro umano, nel 1987. Deep Thought, il predecessore di Deep Blue, fu anch'esso sviluppato alla CMU e incrementò ulteriormente la pura velocità di ricerca (Hsu et al., 1990) raggiungendo un punteggio di 2551. Il premio Fredkin, stabilito nel 1980, offriva 5000 dollari al primo programma che avesse raggiunto il grado di maestro, 10.000 al primo che avesse ottenuto un punteggio USCF (United States Chess Federation) di 2500 (molto vicino al livello di un grande maestro) e 100.000 al primo che avesse battuto il campione del mondo umano. Il premio da 5000 dollari fu vinto da Belle nel 1983, quello da 10.000 da Deep Thought nel 1989, e quello da 100.000 da Deep Blue per la sua vittoria su Garry Kasparov nel 1997. È importante ricordare che il successo di Deep Blue fu dovuto a migliorie algoritmiche oltre che alla semplice potenza dell'hardware (Hsu, 1999; Campbell et al., 2002). Tecniche come l'euristica della mossa nulla (Beal, 1990) avevano permesso ai programmi di essere molto selettivi nella ricerca. Gli ultimi tre campionati del mondo per computer nel 1992, 1995 e 1999 sono stati vinti da programmi in esecuzione su PC standard. La più completa descrizione di un programma moderno per gli scacchi è probabilmente quella di Ernst Heinz (2000), il cui programma DARKTHOUGHT si è classificato primo nel campionato del 1999 per programmi non commerciali su PC.

Per superare i problemi che affliggono il cosiddetto "approccio standard", a cui abbiamo fatto accenno nel Paragrafo 6.7, sono stati fatti molti tentativi. Il primo algoritmo di ricerca selettivo con una base teorica è stato probabilmente B^* (Berliner, 1979), che invece di assegnare una stima univoca a ogni nodo nell'albero di gioco lavora sugli estremi dell'intervallo dei suoi possibili valori. L'espansione seleziona i nodi foglia nel tentativo di raffinare gli estremi dei valori del livello superiore, finché risulta che una mossa è "palesemente preferibile". Palay (1985) estende l'idea di B^* usando distribuzioni di probabilità al posto degli intervalli. La tecnica di David McAllester (1988) denominata "conspiracy number search" espande i nodi foglia che, cambiando valore, potrebbero causare un cambiamento della mossa selezionata. MGSS* (Russell e Wefald, 1989) sfrutta le tecniche di teoria delle decisioni presentate nel Capitolo 16 (nel 2° vol.) per stimare il valore dell'espansione di ogni foglia in termini di miglioramento atteso nella qualità della decisione alla radice dell'albero. Il suo algoritmo si dimostrò migliore di alfa-beta

nel gioco dell'Othello, benché prendesse in considerazione un numero di nodi un ordine di grandezza inferiore. L'approccio MGSS*, in via di principio, è applicabile al controllo di ogni forma di deliberazione.

La ricerca alfa-beta, sotto molti aspetti, si può considerare come la versione a due giocatori del "branch-and-bound" in profondità, che nel caso di agenti singoli è dominato da A*. L'algoritmo SSS* (Stockman, 1979) può essere considerato la versione a due giocatori di A* e non espande mai un numero di nodi maggiore di alfa-beta per raggiungere la stessa decisione. I requisiti di memoria e il costo computazionale del mantenimento della coda rendono SSS* inapplicabile nella sua forma originale, ma ne è già stata sviluppata una versione a partire dall'algoritmo RBFS (Korf e Chickering, 1996) che richiede uno spazio lineare. Plaat et al. (1996) formularono nuovamente SSS* sotto forma di combinazione di ricerca alfa-beta e di tabelle di trasposizione, mostrando come superare i limiti dell'algoritmo originale e sviluppando una nuova variante chiamata MTD(f) che è stata adottata da molti dei migliori programmi.

D. F. Beal (1980) e Dana Nau (1980, 1983) hanno studiato le debolezze di minimax applicato alle valutazioni approssimate. Gli autori dimostrarono che, date alcune ipotesi sull'indipendenza della distribuzione dei valori delle foglie nell'albero, minimax può assegnare alla radice un valore che è di fatto *meno* affidabile dell'uso diretto della funzione di valutazione stessa. Il libro di Pearl *Heuristics* (1984) spiega parzialmente quest'apparente paradosso e analizza molti algoritmi di gioco. Baum e Smith (1997) propongono di rimpiazzare minimax con una tecnica basata sulla probabilità, dimostrando che per certe categorie di giochi questo porta a risultati migliori. Sugli effetti del taglio della ricerca a diversi livelli e l'applicazione delle funzioni di valutazione la teoria consolidata è ancora poca.

L'algoritmo expectiminimax fu proposto da Donald Michie (1966), sebbene sia ovviamente basato sui principî di valutazione degli alberi di gioco enunciati da von Neumann e Morgenstern. Bruce Ballard (1983) estese la potatura alfa-beta per trattare anche gli alberi che comprendono nodi di possibilità. Il primo programma per backgammon di successo fu BKG (Berliner, 1977, 1980b); usava una complessa funzione di valutazione costruita a mano e la sua ricerca si fermava alla profondità 1. Fu il primo programma a sconfiggere il campione del mondo umano in un gioco da tavolo classico (Berliner, 1980a). Berliner ammise subito che l'incontro era stato una semplice esibizione (non una sfida per il titolo) e che BKG era stato davvero fortunato con i dadi. Il lavoro di Gerry Tesauro, prima su NEUROGAMMON (Tesauro, 1989) e più tardi su TD-GAMMON (Tesauro, 1995), dimostrò che si potevano ottenere risultati molto migliori grazie all'apprendimento per rinforzo, che tratteremo nel Capitolo 21 del 2° volume.

Tra tutti i giochi classici, il primo di cui i computer arrivarono a poter svolgere un'intera partita non furono gli scacchi, ma la dama. Christopher Strachey (1952) scrisse il primo programma funzionante. Schaeffer (1997) offre un resoconto molto leggibile dello sviluppo del suo programma Chinook campione del mondo, che non trascura i difetti e gli aspetti problematici.

I primi programmi per giocare a Go furono sviluppati più tardi di quelli dedicati alla dama e agli scacchi (Lefkovitz, 1960; Remus, 1962), e il loro progresso è stato più lento. Ryder (1971) usò un approccio basato sulla ricerca pura, adottando una serie di metodi di potatura selettiva per gestire l'enorme fattore di ramificazione. Zobrist (1970) utilizzò regole condizione-azione per suggerire mosse plausibili quando venivano rilevate particolari configurazioni di pezzi. Reitman e Wilcox (1979) combinarono con buoni risultati le regole e la ricerca, e la maggior parte dei programmi moderni hanno seguito quest'approccio ibrido. Müller (2002) riassume lo stato dell'arte del Go per computer e fornisce una grande quantità di riferimenti. Anshelevich (2000) ha applicato tecniche simili per giocare a Hex. Gli sviluppi presenti sono documentati dalla *Computer Go Newsletter*, pubblicata dalla Computer Go Association.

Articoli sull'uso dei computer applicato ai giochi appaiono in una quantità di riviste. Gli atti del congresso chiamato (alquanto impropriamente) *Heuristic Programming in Artificial Intelligence* offrono un resoconto delle Olimpiadi per Computer, che includono una varietà di giochi. Esistono anche molte raccolte commentate di articoli fondamentali per la ricerca sui giochi (Levy, 1988a, 1988b; Marsland e Schaeffer, 1990). L'International Computer Chess Association (ICCA), fondata nel 1977, pubblica ogni quadriennale l'*ICCA Journal* (precedentemente chiamato *ICCA Journal*). Lavori importanti sono stati pubblicati nella serie antologica *Advances in Computer Chess*, a partire da Clarke (1977). Il Volume 134 della rivista *Artificial Intelligence* (2002) contiene una descrizione dello stato dell'arte dei programmi per giocare a scacchi, Othello, Hex, Shogi, Go, backgammon, poker, ScrabbleTM e altri.

Esercizi

- 6.1 Questo problema permette di sperimentare i concetti base dei giochi usando come esempio il tic-tac-toe (gioco del tris). Definiamo X_n come il numero di righe, colonne o diagonali con esattamente n X e nessuna O . In modo analogo, O_n sarà il numero di righe, colonne o diagonali che contengono solamente n O . La funzione di utilità assegna +1 a ogni posizione con $X_3 = 1$ e -1 a ogni posizione con $O_3 = 1$. Tutte le altre posizioni terminali hanno utilità 0. Per le posizioni non terminali, usiamo una funzione di valutazione lineare definita come $\text{Eval}(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$.
- Approssimativamente, quante possibili partite di tic-tac-toe esistono?
 - Mostrate l'intero albero di gioco partendo da una plancia vuota fino a una profondità 2 (cioè, con solo una X e una O), prendendo in considerazione la simmetria.

- c. Segnate sul vostro albero le valutazioni di tutte le posizioni a profondità 2.
- d. Usando l'algoritmo minimax, segnate sull'albero i valori “tirati su” (*backed-up*) dai nodi figli per le posizioni a profondità 1 e 0, e usate quei valori per scegliere la miglior mossa iniziale.
- e. Indicate con un cerchio i nodi a profondità 2 che *non* sarebbero valutati se si applicasse una potatura alfa–beta, presumendo che i nodi siano generati *nell'ordine ottimo per la potatura alfa–beta*.
- 6.2 Dimostrate la seguente asserzione: per ogni albero di gioco, l'utilità ottenuta da MAX usando decisioni minimax contro un avversario MIN subottimo non sarà mai inferiore all'utilità ottenuta giocando contro un MIN ottimo. Potete disegnare un albero di gioco in cui MAX può fare ancora meglio usando una strategia *subottima* contro un MIN subottimo?
- 6.3 Considerate il gioco a due descritto nella Figura 6.14.
- Disegnate l'albero di gioco completo, usando le seguenti convenzioni:
 - ◆ scrivete ogni stato nella forma (s_A, s_B) dove s_A e s_B indicano la posizione dei gettoni;
 - ◆ indicate ogni stato terminale con un quadrato e scrivete il suo valore all'interno di un cerchietto;
 - ◆ indicate gli *stati ciclici* (quelli che appaiono già nel cammino dalla radice) con dei quadrati doppi. Dato che non è chiaro come si può assegnare loro un valore, annotate ognuno con un punto interrogativo “?” racchiuso in un cerchio.
 - Ora indicate accanto a ogni nodo, sempre in un cerchietto, il valore minimax “tirato su” (*backed-up*). Spiegate come avete gestito i valori “?” e perché.

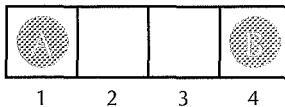


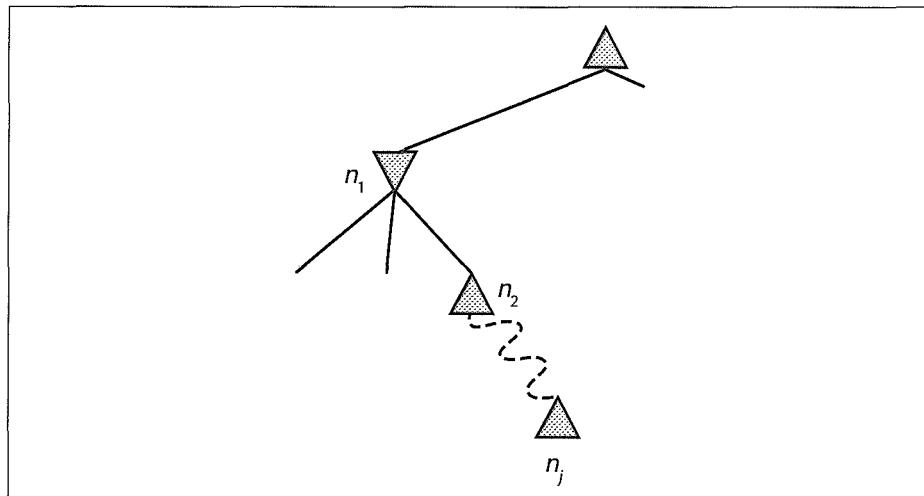
Figura 6.14 La posizione iniziale di un semplice gioco. Il giocatore A muove per primo. I due giocatori si alternano, ognuno muovendo il proprio segnalino in una casella adiacente libera *in qualsiasi direzione*. Se l'avversario occupa una casella adiacente il giocatore può “saltarlo” e muovere nello spazio libero retrostante, se esiste (ad esempio, se A si trova in 3 e B in 2, A può saltare indietro nella casella 1). Il gioco termina quando un giocatore raggiunge l'altro lato della plancia. Se il giocatore A raggiunge per primo lo spazio 4, il valore della partita per lui è +1; se è il giocatore B a raggiungere per primo lo spazio 1, allora la partita per A vale –1.

- c. Spiegate perché l'algoritmo minimax standard fallirebbe se applicato a quest'albero di gioco e descrivete brevemente come potreste modificarlo per farlo funzionare, ispirandovi alla vostra risposta al punto (b). Il vostro algoritmo modificato fornisce decisioni ottime in tutti i giochi che presentano cicli?
- d. Questo gioco a quattro caselle può essere generalizzato a n caselle, con $n > 2$. Dimostrate che A vince se n è pari e perde se n è dispari.
- 6.4 Implementate generatori di mosse e funzioni di valutazione per uno o più dei seguenti giochi: Kalah (Awalè), Othello, dama e scacchi. Costruire un agente generico alfa-beta per i giochi sfruttando la vostra implementazione. Confrontate l'effetto dell'incremento della profondità di ricerca, del miglioramento dell'ordine delle mosse e di quello della funzione di valutazione. Quanto si avvicina il vostro fattore di ramificazione effettivo al caso ideale, corrispondente a un ordinamento delle mosse perfetto?
- 6.5 Sviluppate una dimostrazione formale di correttezza per la potatura alfa-beta. Per far ciò, considerate la situazione illustrata nella Figura 6.15. La questione è se potare o no il nodo n_j , che è un nodo max discendente di n_1 . L'idea base è di potare se e solo se si può dimostrare che il valore minimax di n_1 è indipendente dal valore di n_j .
- a. Il valore di n_1 è dato da
- $$n_1 = \min(n_2, n_{21}, \dots, n_{2b_2}).$$

Trovate un'espressione simile per n_2 e di conseguenza un modo per esprimere n_1 in funzione di n_j .

Figura 6.15

Una situazione in cui si deve considerare se potare il nodo n_j .



- b. Sia l_i (ℓ) il valore minimo (o massimo) dei nodi alla *sinistra* del nodo n_i alla profondità i , il cui valore minimax è già noto. In modo analogo, sia r_i il valore minimo (o massimo) dei nodi non ancora esplorati alla destra di n_i alla profondità i . Riscrivete la vostra espressione per n_1 in termini di valori l_i e r_i .
- c. Ora riformulate l'espressione per mostrare che per avere effetto su n_1 , n_j non deve superare un certo limite derivato dai valori l_i .
- d. Ripetete il processo nel caso in cui n_j sia un nodo min.

6.6 Implementate l'algoritmo expectiminimax e quello *-alfa–beta, descritto da Ballard (1983), per potare gli alberi di gioco che presentano nodi di possibilità. Metteteli alla prova con un gioco come backgammon e misurate l'efficienza nella potatura *-alfa–beta.

6.7 Dimostrate che se si applica una trasformazione lineare positiva dei valori delle foglie (cioè, se si trasforma ogni x in $ax + b$ con $a > 0$), la scelta finale della mossa rimane invariata anche quando sono presenti nodi di possibilità.

6.8 Considerate la seguente procedura per scegliere le mosse nei giochi che hanno elementi casuali:

- ◆ generate alcune sequenze di tiri di dado (ad esempio 50) che arrivino tutte fino a una profondità adeguata (ad esempio 8);
- ◆ conoscendo in anticipo i tiri di dado, l'albero di gioco diventa deterministico. Per ogni sequenza, risolvete l'albero di gioco risultante usando alfa–beta;
- ◆ in base ai risultati ottenuti, stimate il valore di ogni mossa e scegliete quella migliore.

Secondo voi questa procedura funzionerà bene? Perché, o perché no?

6.9 Descrivete e implementate un ambiente di gioco *multigiocatore in tempo reale*, in cui il tempo fa parte dello stato dell'ambiente. Ogni giocatore ne avrà a disposizione una quantità prefissata.

6.10 Descrivete o implementate descrizioni di stato, generatori di mosse, test terminali, funzioni di utilità e funzioni di valutazione per uno o più dei seguenti giochi: Monopoli, Scarabeo, bridge (presumendo che il contratto sia dato) e poker (scegliendo la vostra variante favorita).

6.11 Considerate attentamente l'interazione tra eventi casuali e informazione parziale in ognuno dei giochi citati nell'Esercizio 6.10.

- a. Per quale gioco è appropriato il modello expectiminimax standard? Implementate l'algoritmo ed eseguitelo mediante il vostro agente giocatore, con le modifiche appropriate all'ambiente specifico.



- b. Per quale gioco sarebbe appropriato lo schema descritto nell'Esercizio 6.8?
- c. Discutete come si può gestire il fatto che, in alcuni giochi, i partecipanti non hanno la stessa conoscenza dello stato corrente.
- 6.12 L'algoritmo minimax presume che i giocatori muovano sempre nello stesso ordine, ma nei giochi di carte come whist e bridge, il vincitore di una presa è il primo a giocare nella mano successiva.
- Modificate l'algoritmo in modo che funzioni correttamente per questi giochi. Potete presumere che esista una funzione $VINCITORE(s)$ che restituisce il giocatore che ha vinto la presa appena completata, se questo esiste.
 - Disegnate l'albero di gioco per le prime delle coppie di mani mostrate a pag. 231.
- 6.13 Il programma per la dama Chinook fa un uso estensivo di un database di finali, che fornisce valori esatti per ogni posizione con otto pezzi o meno. Come pensate si possa generare in modo efficiente un simile database?
- 6.14 Discutete se sarebbe possibile applicare l'approccio standard a giochi come tennis, biliardo o croquet, che si svolgono in uno spazio degli stati fisici continuo.
- 6.15 Descrivete come cambiano gli algoritmi minimax e alfa-beta nel caso di giochi a due, a somma zero in cui ogni giocatore ha la propria funzione di utilità. Potete presumere che ogni giocatore conosca la funzione di utilità dell'avversario. Se non ci sono vincoli sulle utilità terminali dei due, è possibile potare nodi con la tecnica alfa-beta?
- 6.16 Supponete di avere un programma scacchistico capace di valutare 1 milione di nodi al secondo. Decidete quale rappresentazione compatta dello stato del gioco adottare per la memorizzazione in una tabella di trasposizione. Quante configurazioni riuscite a registrare in una tabella da 500MB tenuta in memoria? Saranno sufficienti per i tre minuti di ricerca disponibili per ogni mossa? Quante volte potete accedere alla tabella nel tempo richiesto da una valutazione? Ora supponete che la tabella sia troppo grande per stare tutta in RAM. Approssimativamente, quante valutazioni potrete eseguire nel tempo necessario per accedere al disco, usando hardware standard?

quanto sei idiota!
ignorante, stupido, scemo!
x

Conoscenza e ragionamento

Capitolo 7

Agenti logici

Nel quale progettiamo agenti che possono costruire rappresentazioni del mondo, applicare processi di inferenza per derivare nuove rappresentazioni e usarle per dedurre cosa fare.

In questo capitolo introduciamo gli agenti basati sulla conoscenza. Il concetto di rappresentazione della conoscenza e i processi di ragionamento che permettono di manipolarla sono un argomento centrale dell'intelligenza artificiale.

Gli esseri umani apparentemente sanno delle cose, e sono capaci di ragionarci sopra. La conoscenza e il ragionamento sono importanti anche per gli agenti artificiali, perché permettono di ottenere comportamenti di successo che sarebbero dificili da raggiungere in altro modo. Abbiamo già visto che la conoscenza delle conseguenze di un'azione permette agli agenti risolutori di problemi di comportarsi bene in ambienti complessi. Un agente reattivo, per trovare la strada da Arad a Bucarest, non potrebbe che affidarsi alla fortuna. Tuttavia, la conoscenza degli agenti risolutori di problemi è molto specifica e poco flessibile: un programma che gioca a scacchi può calcolare le mosse legali del re, ma in senso generale non "sa" che nessun pezzo può essere presente contemporaneamente in due caselle. Gli agenti basati sulla conoscenza possono trarre beneficio da conoscenze espresse in una forma molto generale, combinando e ricombinando le informazioni per adattarle a una varietà di scopi. Spesso questo processo può avere ben poco a che fare con le necessità immediate, come succede quando un matematico dimostra un teorema o un astronomo calcola l'aspettativa di vita futura del pianeta Terra.

La conoscenza e il ragionamento hanno anche un ruolo cruciale nella gestione degli ambienti parzialmente osservabili. Gli agenti che progetteremo, prima di scegliere l'azione da intraprendere, possono combinare la loro conoscenza generale con le percezioni correnti per dedurre aspetti nascosti dello stato del mondo. In modo analogo, prima di prescrivere una cura un medico deve diagnosticare la malattia del paziente, ovvero dedurre uno stato patologico non direttamente osservabile. Una parte della conoscenza utilizzata dal medico ha la forma di regole appre-

se dai libri e dai docenti, un'altra parte è costituita da schemi di associazione che egli stesso potrebbe trovare difficile descrivere in modo esplicito. In ogni caso, tutto quello che risiede nella testa del medico conta come conoscenza.

Anche la comprensione del linguaggio naturale richiede la deduzione di uno stato nascosto, e precisamente dell'intenzione di chi sta parlando. Quando udiamo la frase "John vide il diamante attraverso il vetro e lo desiderò", sappiamo che "lo" si riferisce al diamante e non al vetro: ragioniamo, forse inconsciamente, in base alla nostra conoscenza del valore relativo dei materiali. In modo analogo, quando sentiamo "John scagliò il mattone contro il vetro e lo ruppe", sappiamo che "lo" si riferisce al vetro. Il ragionamento ci permette di trattare un numero virtualmente infinito di frasi utilizzando un deposito finito di conoscenza comune. Gli agenti risolutori di problemi trovano difficile gestire questo tipo di ambiguità, perché la loro rappresentazione delle contingenze è intrinsecamente esponenziale.

Un'altra ragione per studiare gli agenti basati sulla conoscenza è la loro flessibilità: possono intraprendere nuove attività espresse sotto forma di obiettivi descritti esplicitamente, possono ottenere rapidamente nuove competenze ricevendo o apprendendo conoscenze aggiuntive e possono adattarsi ai cambiamenti dell'ambiente modificando e aggiornando la relativa conoscenza.

Cominceremo con il presentare il progetto generale dell'agente nel Paragrafo 7.1. Il Paragrafo 7.2 introduce un nuovo, semplice ambiente, il mondo del wumpus, e descrive il funzionamento di un agente basato sulla conoscenza senza alcun dettaglio tecnico. Nel Paragrafo 7.3 presenteremo i principi generali della logica, che in tutta la Parte III del libro sarà il veicolo principale per la rappresentazione della conoscenza. Negli agenti logici, la conoscenza è sempre definita: ogni proposizione è sempre vera o falsa nel mondo, benché un agente possa non avere un'idea precisa riguardo alcune proposizioni.

La logica ha il vantaggio pedagogico di essere un esempio semplice di rappresentazione per agenti basati sulla conoscenza, ma presenta anche delle notevoli limitazioni. È chiaro che una gran parte del ragionamento effettuato dagli esseri umani e da altri agenti in ambienti parzialmente osservabili dipende dalla gestione di conoscenza incerta. La logica fatica a rappresentare efficacemente l'incertezza, così nella Parte V, nel 2° volume, introdurremo la probabilità che invece lo fa benissimo. Sempre nel 2° volume, nelle parti VI e VII tratteremo molte rappresentazioni, tra cui alcune basate sulla matematica del continuo come le combinazioni di gaussiane, reti neurali e altre rappresentazioni.

Il Paragrafo 7.4 definisce una semplice logica, chiamata logica o calcolo proposizionale. Benché sia molto meno espressiva della logica del primo ordine (Capitolo 8), quella proposizionale va più che bene per presentare i concetti fondamentali. Inoltre, il suo utilizzo per il ragionamento è supportato da una tecnologia ben sviluppata, che presentiamo nei Paragrafi 7.5. e 7.6. Infine, il Paragrafo 7.7 unisce il concetto di agente logico con la tecnologia della logica proposizionale per costruire alcuni semplici agenti per il mondo del wumpus. Identificheremo anche alcune limitazioni del calcolo proposizionale: questo ci darà motivo di sviluppare logiche più potenti nei capitoli successivi.

7.1 Agenti basati sulla conoscenza

Il componente più importante degli agenti basati sulla conoscenza è appunto la base di conoscenza, o KB (dall'inglese *knowledge base*). Informalmente, la base di conoscenza è costituita da un insieme di formule, espresse mediante un linguaggio di rappresentazione della conoscenza. Ogni formula rappresenta un'asserzione sul mondo.

La base di conoscenza deve prevedere meccanismi per aggiungere nuove formule e per le interrogazioni. I nomi standard per queste due azioni sono rispettivamente TELL (asserisci) e ASK (chiedi): entrambe possono comportare un processo di inferenza, ovvero la derivazione di nuove formule a partire da quelle conosciute. Negli agenti logici, che sono il principale oggetto di studio di questo capitolo, l'inferenza deve soddisfare il requisito fondamentale che la risposta a ogni richiesta (ASK) posta alla base di conoscenza sia una conseguenza di quello che le è stato detto (attraverso TELL) in precedenza. Più avanti nel capitolo saremo più precisi riguardo alla fondamentale parola "conseguenza". Per adesso, accontentiamoci di dire che il processo di inferenza non può semplicemente inventarsi i fatti.

La Figura 7.1 mostra lo schema di un programma agente basato sulla conoscenza: come tutti i nostri agenti, prende in input una percezione e restituisce un'azione. L'agente mantiene in memoria una base di conoscenza, KB, che può contenere conoscenza iniziale (*background knowledge*). Ogni volta che viene invocato, il programma agente fa due cose: prima di tutto comunica le sue percezioni (attraverso TELL) alla base di conoscenza. Quindi le chiede (ASK) quale azione eseguire. Rispondere a questa domanda può comportare un esteso processo di ragionamento sullo stato corrente del mondo, le conseguenze delle possibili azioni e così via. Una volta che è stata scelta un'azione l'agente la registra con TELL prima di eseguirla. Il secondo TELL è necessario per dire alla base di conoscenza che l'ipotetica azione è stata effettivamente eseguita.

base di conoscenza
formule
linguaggio di
rappresentazione
della conoscenza

inferenza

conoscenza iniziale

```

function KB-Agente(percezione) returns un'azione
    static: KB, una base di conoscenza
        t, un contatore, inizializzato a 0, che indica il tempo

    TELL(KB, COSTRUISCI-FORMULA-PERCEZIONE(percezione, t))
    azione ← ASK(KB, COSTRUISCI-INTERROGAZIONE-AZIONE(t))
    TELL(KB, COSTRUISCI-FORMULA-AZIONE(azione, t))
    t ← t + 1
    ...
    return azione

```

Figura 7.1 Un generico agente basato sulla conoscenza.



livello di conoscenza

I dettagli del linguaggio di rappresentazione sono racchiusi nelle tre funzioni che implementano l'interfaccia tra i sensori e attuatori da una parte e il sistema interno di rappresentazione e ragionamento dall'altra. COSTRUISCI-FORMULA-PERCEZIONE prende una percezione e un dato istante temporale e restituisce la formula che assicura che in quel momento l'agente ha ricevuto quella percezione. COSTRUISCI-INTERROGAZIONE-AZIONE prende in input un istante temporale e restituisce la formula che chiede quale azione intraprendere in quel momento. I dettagli del meccanismo di inferenza sono nascosti dentro TELL e ASK: li riveliamo nei prossimi paragrafi.

L'agente della Figura 7.1 ha un aspetto molto simile a quelli dotati di uno stato interno che abbiamo descritto nel Capitolo 2. Tuttavia le limitazioni intrinseche di TELL e ASK fanno sì che l'agente basato sulla conoscenza non sia un programma arbitrario per il calcolo di azioni. Un agente siffatto si presta bene a essere descritto a livello di conoscenza, in cui per fissare il comportamento basta specificare solamente ciò che l'agente conosce e i suoi obiettivi. Ad esempio, un taxi automatico potrebbe avere l'obiettivo di portare un passeggero in Marin County, e potrebbe sapere di trovarsi a San Francisco e che il Golden Gate Bridge è l'unico collegamento tra le due località. Allora ci aspetteremo che il taxi attraversi il ponte, perché sa che così facendo raggiungerà il suo obiettivo. Notate che quest'analisi è del tutto indipendente dal funzionamento del taxi a livello di implementazione: non importa se la sua conoscenza della geografia è realizzata con liste dinamiche o mappe di pixel, o se per ragionare manipola stringhe di simboli memorizzate in registri o propaga segnali in una rete neurale.



livello di implementazione

Come abbiamo menzionato nell'introduzione a questo capitolo, si possono costruire sistemi basati sulla conoscenza semplicemente dicendo loro quello che devono sapere. Il programma agente iniziale, prima di cominciare a ricevere percezioni, è costruito aggiungendo una a una le formule che rappresentano la conoscenza dell'ambiente da parte del progettista. Un linguaggio di rappresentazione che rende facile esprimere la conoscenza per mezzo di formule facilita enormemente la costruzione. Questo procedimento prende il nome di approccio dichiarativo alla realizzazione di sistemi. In contrapposizione a questo, l'approccio procedurale codifica direttamente nel programma i comportamenti desiderati sotto forma di codice; il sistema risultante minimizza l'importanza della rappresentazione esplicita e del ragionamento e può essere molto più efficiente. Vedremo esempi di agenti di entrambi i tipi nel Paragrafo 7.7. Negli anni '70 e '80 ci sono stati dibattiti molto accesi tra i difensori dei due approcci: oggi l'opinione comune è che un agente di successo debba combinare sia elementi dichiarativi che procedurali.

Oltre a dire a un agente basato sulla conoscenza quello che deve sapere attraverso TELL, possiamo fornirgli dei meccanismi che gli permettano di apprendere. Questi meccanismi, che discuteremo nel Capitolo 18 del 2^o volume, partono da una serie di percezioni per dare origine a nuova conoscenza generale riguardante l'ambiente; questa può essere poi incorporata nella base di conoscenza dell'agente e utilizzata nel processo decisionale. In questo modo l'agente può essere completamente autonomo.

dichiarativo

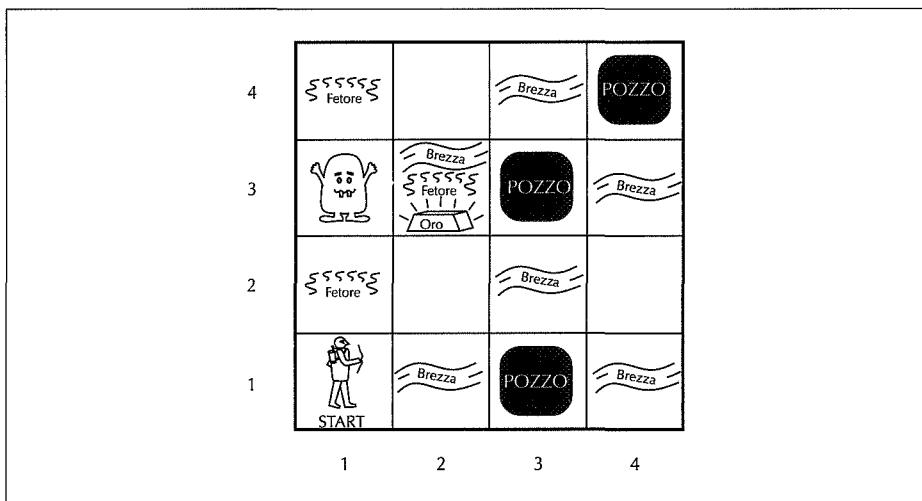
Tutte queste capacità – rappresentazione, ragionamento e apprendimento – si appoggiano allo sviluppo, durato secoli, della teoria e tecnologia della logica. Prima di presentarla, comunque, introdurremo un semplice mondo che ci aiuterà nell'esposizione.

7.2 Il mondo del wumpus

Il **mondo del wumpus** è una caverna formata da stanze collegate da passaggi. Acciattato da qualche parte nell'ombra c'è il wumpus, una bestia pronta a divorare chiunque entri. Il wumpus può essere colpito da lontano da un agente, che però ha una sola freccia. Alcune stanze contengono pozzi senza fondo che intrappoleranno chiunque entri (tranne il wumpus, che è troppo grande per caderci dentro). L'unica caratteristica che mitiga la durezza di quest'ambiente è la possibilità di trovare un mucchio d'oro. Benché risulti piuttosto blando se confrontato con i videogiochi più recenti, il mondo del wumpus (come ha suggerito per primo Michael Genesereth) costituisce un ottimo ambiente di test per gli agenti intelligenti.

Un esempio di **mondo del wumpus** è mostrato nella Figura 7.2. La definizione precisa dell'ambiente è fornita, come suggerito nel Capitolo 2, sotto forma di descrizione PEAS.

- ♦ **Misura di prestazione:** +1000 se si raccoglie l'oro, -1000 se si cade in un pozzo o si viene divorati dal wumpus, -1 per ogni azione eseguita e -10 per l'uso della freccia.



mondo del wumpus

Figura 7.2
Un tipico mondo del wumpus.
L'agente si trova nell'angolo in basso a sinistra.

- ◆ **Ambiente:** una griglia 4×4 di stanze. L'agente comincia sempre nella posizione etichettata [1,1], rivolto verso destra. Le posizioni dell'oro e del wumpus sono scelte casualmente, con una distribuzione uniforme, tra tutti i riquadri tranne quello iniziale. Inoltre, tutti i riquadri tranne quello iniziale hanno una probabilità 0,2 di contenere un pozzo senza fondo.
- ◆ **Attuatori:** l'agente può muoversi in avanti e girare a destra o a sinistra di 90°. L'agente muore miserevolmente se entra in un riquadro che contiene un pozzo o il wumpus vivo: entrare in una stanza con un wumpus morto non è rischioso, anche se il fetore è quasi insopportabile. Muoversi in avanti non ha alcun effetto se davanti all'agente c'è un muro. L'azione *Afferra* può essere usata per prendere un oggetto che si trova nella stessa stanza. L'azione *Scocca scaglia* una freccia in linea retta nella direzione verso cui l'agente è rivolto; la freccia continua la sua corsa finché non colpisce il wumpus uccidendolo o sbatte contro un muro. L'agente possiede una sola freccia, per cui l'unica azione *Scocca* ad avere effetto è la prima.
- ◆ **Sensori:** l'agente ha cinque sensori, ognuno dei quali fornisce un singolo bit di informazione:
 - nel riquadro che contiene il wumpus e in quelli direttamente adiacenti (non in diagonale) l'agente percepisce un pungente fetore;
 - nei riquadri direttamente adiacenti a un pozzo l'agente percepisce uno spostamento d'aria;
 - nel riquadro che contiene l'oro l'agente percepisce uno scintillio;
 - qualora l'agente dovesse sbattere contro un muro, percepisce l'urto;
 - quando il wumpus viene ucciso emette un ululato straziante, percepibile nell'intera caverna.

Le percezioni saranno fornite all'agente sotto forma di una lista di cinque simboli: ad esempio, se c'è puzza e movimento d'aria ma nessun scintillio, urto o ululato, l'agente riceverà la percezione [*Fetore, Brezza, None, None, None*].

L'Esercizio 7.1 vi chiederà di definire l'ambiente del wumpus sulla base delle varie caratteristiche discusse nel Capitolo 2. La difficoltà principale dell'agente è legata alla sua iniziale ignoranza della configurazione dell'ambiente; per superarla sembra essere necessario un ragionamento logico. Nella maggior parte delle istanze del mondo del wumpus, l'agente può recuperare l'oro in sicurezza. Occasionalmente, dovrà scegliere se tornare a casa a mani vuote o rischiare la vita per ottenerlo. Il 21% circa degli ambienti è decisamente antisportivo, perché l'oro si trova in fondo a un pozzo o in una stanza circondata da pozzi.

Esaminiamo il comportamento di un agente basato sulla conoscenza che esplora l'ambiente mostrato nella Figura 7.2. La sua base di conoscenza iniziale contiene le regole del mondo, come le abbiamo descritte qui sopra: in particolare,

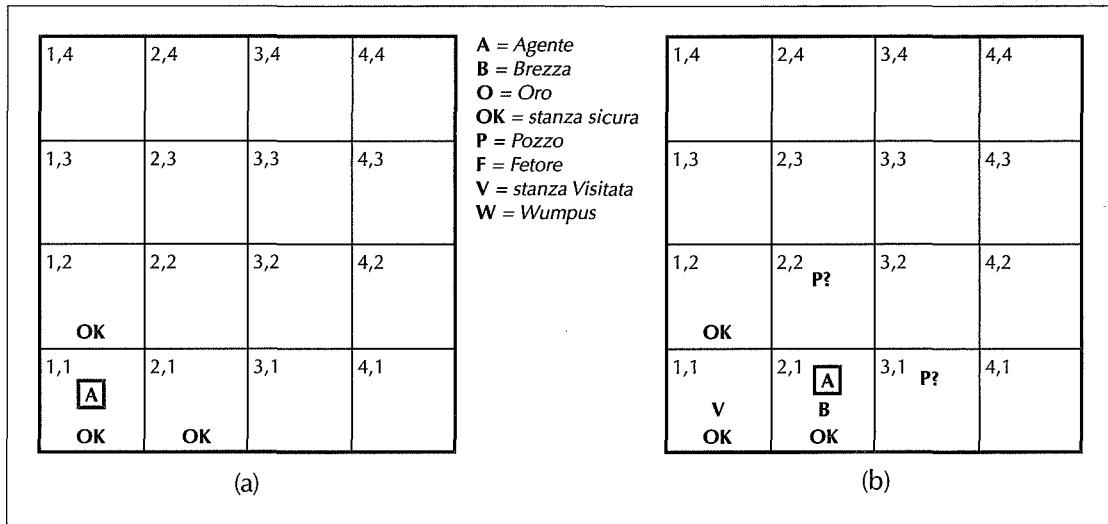


Figura 7.3 Il primo passo dell'agente nel mondo del wumpus. (a) La situazione iniziale, dopo la percezione *[None, None, None, None, None]*. (b) Dopo una mossa, con la percezione *[None, Brezza, None, None, None]*.

l'agente sa di essere nella posizione [1,1] e che la stanza è sicura. Vedremo come evolve la sua conoscenza man mano che riceve nuove percezioni e intraprende azioni.

La prima percezione è *[None, None, None, None, None]*, da cui l'agente può dedurre che le stanze adiacenti sono sicure. La Figura 7.3(a) mostra lo stato della conoscenza dell'agente a questo punto. Indichiamo alcune delle formule presenti nella base di conoscenza mediante lettere come *B* (brezza) e *OK* (stanza sicura, senza pozzi né wumpus) scritte nei riquadri appropriati. La Figura 7.2 rappresenta il mondo stesso.

Dato che in [1,1] non c'è puzza né movimento d'aria, l'agente può dedurre che [1,2] e [2,1] sono privi di pericoli: per indicare questo fatto, i riquadri sono marcati *OK*. Un agente cauto entrerà solo nelle stanze così marcate. Supponiamo che l'agente decida di avanzare in [2,1], trovandosi così nella situazione della Figura 7.3(b).

L'agente percepisce un movimento d'aria in [2,1], perciò in una stanza adiacente dev'esserci un pozzo. Questo non può essere in [1,1], per cui deve trovarsi in [2,2] o [3,1], o anche in entrambe le posizioni. La notazione *P?* nella Figura 7.3(b) indica questa possibilità. A questo punto, c'è un solo riquadro marcato con *OK* e non ancora visitato; il nostro prudente agente tornerà quindi indietro in [1,1] e da lì procederà in [1,2].

La percezione in [1,2] è *[Fetore, None, None, None, None]*, che dà come risultato lo stato di conoscenza della Figura 7.4(a). L'odore nella stanza [1,2] significa che il wumpus è vicino. Ma il mostro non può trovarsi in [1,1] per definizione, e

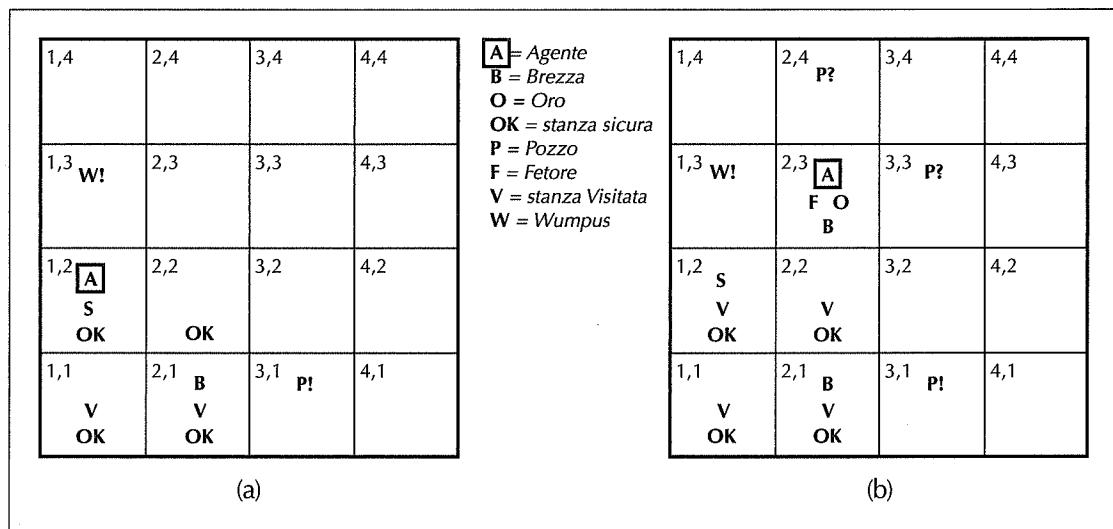


Figura 7.4 Due fasi successive dell'esplorazione dell'agente. (a) Dopo la terza mossa, con la percezione [Fetore, None, None, None, None]. (b) Dopo la quinta mossa, con percezione [Fetore, Brezza, Scintillio, None, None].

neppure in [2,2], altrimenti l'agente avrebbe percepito la puzza anche nella stanza [2,1]. L'agente può quindi dedurre che il wumpus si trova in [1,3], ciò che indichiamo con la notazione $W!$. Inoltre, la mancanza di *Brezza* in [1,2] implica che non ci siano pozzi in [2,2], e a questo punto si può inferire con certezza che ce ne dev'essere uno nella posizione [3,1]. Quest'ultima deduzione è abbastanza difficile, perché si fonda su conoscenze apprese in diversi momenti e in posti differenti; inoltre si basa sull'assenza di una particolare percezione per compiere un passo logico fondamentale. Questo tipo di inferenza va oltre la capacità della maggior parte degli animali, ma è tipica del genere di ragionamento svolto da un agente logico.

Ora l'agente si è convinto oltre ogni dubbio che in [2,2] non ci sono pozzi né mostri, per cui è *OK* andarci. Non mostreremo lo stato della conoscenza dell'agente in [2,2]; ci limiteremo a supporre che da lì si giri verso sinistra ed entri nella stanza [2,3], raggiungendo la situazione della Figura 7.4(b). In [2,3] l'agente percepisce lo scintillio dell'oro, così può raccoglierlo e terminare felicemente il gioco.

Ogni volta che l'agente trae una conclusione dalle informazioni disponibili, tale conclusione sarà sempre corretta a patto che lo sia l'informazione di partenza. Questa è una proprietà fondamentale del ragionamento logico. Nel resto del capitolo spiegheremo come costruire agenti logici che possono rappresentare l'informazione necessaria e trarre le conclusioni che abbiamo appena descritto.



7.3 Logica

Questo paragrafo fornisce una panoramica dei concetti fondamentali della rappresentazione e del ragionamento logico. Rimanderemo al prossimo paragrafo i dettagli tecnici legati a una qualsiasi particolare forma di logica, utilizzando invece esempi informali dal mondo del wumpus e dalla semplice aritmetica. Quest'approccio è piuttosto inusual, ma ci piace adottarlo per mostrare che le idee della logica sono molto più generali e belle di quanto si ritenga comunemente.

Nel Paragrafo 7.1 abbiamo detto che le basi di conoscenza sono costituite da formule. Queste formule sono espresse secondo le regole della sintassi del linguaggio di rappresentazione, che specifica quali di esse sono “ben formate”. La nozione di sintassi è abbastanza intuitiva quando si parla dell’aritmetica a noi familiare: “ $x + y = 4$ ” è una formula ben formata, mentre “ $x2y+ =$ ” non lo è. La sintassi dei linguaggi logici (e dell’aritmetica, se è per questo) è solitamente progettata per scrivere articoli e libri. Esistono letteralmente dozzine di sintassi differenti, alcune pieno di lettere greche e simboli matematici esotici, altre basate su diagrammi di bell’aspetto fatti di frecce e bolle. In ogni caso, comunque, le formule contenute nella base di conoscenza di un agente rappresentano configurazioni fisiche di una parte dell’agente stesso; il ragionamento coinvolgerà la generazione e manipolazione di tali configurazioni.

sintassi

Una logica deve anche definire la semantica del linguaggio. Informalmente, potremmo dire che questo termine fa riferimento al “significato” delle formule. Nella logica, comunque, la definizione è più precisa. La semantica di un linguaggio definisce la verità delle formule rispetto a ogni mondo possibile. Per esempio, la semantica normalmente adottata per l’aritmetica specifica che la formula “ $x + y = 4$ ” è vera in un mondo in cui x vale 2 e così y , ma falsa in un mondo in cui entrambe le variabili valgono 1.¹ Nelle logiche standard, ogni formula dev’essere o vera o falsa in ogni possibile mondo: non esistono possibilità intermedie.²

semantica

Quando dovremo essere precisi, al posto di “mondo possibile” useremo il termine modello. Utilizzeremo anche la frase “ m è un modello di α ” per indicare che la formula α è vera nel modello m . Laddove i mondi possibili possono essere considerati ambienti (potenzialmente) reali in cui un agente potrebbe o non potrebbe trovarsi, i modelli sono astrazioni matematiche, e ognuno di essi semplicemente fissa il valore di verità di ogni formula. Informalmente, potremmo ad esempio pensare che x e y sia rispettivamente il numero di uomini e donne seduti intorno a un tavolo per giocare a bridge, e la formula $x + y = 4$ sarebbe vera qualora il nume-

verità

mondo possibile

modello

¹ Avrete sicuramente notato la somiglianza tra la nozione di verità delle formule e quella di soddisfacimento dei vincoli introdotta nel Capitolo 5. Questo non è un caso; i linguaggi per esprimere vincoli sono effettivamente logiche e la risoluzione dei relativi problemi è una forma di ragionamento logico.

² La logica fuzzy o sfumata, che presenteremo nel Capitolo 14 (nel 2° vol.), permette di esprimere gradi di verità.

implicazione

ro totale corrispondesse appunto a quattro. Formalmente, i modelli possibili non sono altro che tutti i modi in cui si possono assegnare i valori alle variabili x e y . Ogni assegnamento fissa il valore di verità di tutte le formule aritmetiche le cui variabili sono x e y .

Ora che abbiamo definito la nozione di verità possiamo parlare del ragionamento logico. Per questo dobbiamo introdurre la relazione di implicazione logica tra formule, che significa che una segue logicamente dall'altra. In notazione matematica, si scrive

$$\alpha \models \beta \quad \alpha \Rightarrow \beta$$

per indicare che α implica β . La definizione formale di implicazione è la seguente: $\alpha \models \beta$ se e solo se, in ogni modello in cui α è vera, anche β lo è. Un altro modo di esprimere lo stesso concetto è dire che, se α è vera, allora anche β deve essere vera. Informalmente, si può dire che la verità di β sia "contenuta" in quella di α . La relazione di implicazione è nota a chiunque conosca l'aritmetica; sappiamo tutti che la formula $x + y = 4$ implica la formula $4 = x + y$. È infatti palese che in ogni modello in cui $x + y = 4$ (come quello in cui entrambe le variabili valgono 2) è anche vero che $4 = x + y$. Vedremo tra poco che una base di conoscenza può essere considerata un'asserzione, e spesso si dice che una base di conoscenza implica una determinata formula.

Possiamo applicare lo stesso tipo di analisi all'esempio di ragionamento nel mondo del wumpus che abbiamo presentato nel paragrafo precedente. Considerate la situazione nella Figura 7.3(b): l'agente non ha percepito nulla in [1,1] e un movimento d'aria in [2,1]. Queste percezioni, unite alla conoscenza da parte dell'agente delle regole del mondo (cioè la descrizione PEAS fornita a pag. 255) costituiscono la KB. L'agente, tra le altre cose, è interessato a sapere se le stanze adiacenti [1,2], [2,2] e [3,1] contengono pozzi. Ognuno dei riquadri potrebbe contenere un pozzo oppure no, per cui (per quanto riguarda quest'esempio) ci sono $2^3 = 8$ modelli possibili, che abbiamo riportato nella Figura 7.5.³

La KB è falsa nei modelli che contraddicono le conoscenze dell'agente: ad esempio in qualsiasi modello in cui [1,2] contiene un pozzo, dato che l'agente non ha percepito alcun movimento d'aria in [1,1]. In effetti, i modelli in cui la KB è vera sono solo tre, indicati come un sottoinsieme nella Figura 7.5. Ora consideriamo le possibili conclusioni:

$$\alpha_1 = \text{"Non c'è pozzo in [1,2]”}$$

$$\alpha_2 = \text{"Non c'è pozzo in [2,2]”}.$$

I modelli corrispondenti alle formule α_1 e α_2 sono indicati rispettivamente nelle Figure 7.5(a) e 7.5(b). Si può verificare quanto segue:

in ogni modello in cui KB è vero, lo è anche α_1 .

³ Benché la figura mostri i modelli come mondi del wumpus parziali, in effetti essi non sono altro che assegnamenti dei valori *true* e *false* alle formule “c'è un pozzo in [1,2]” etc. I modelli, in senso matematico, non hanno bisogno di contenere orribili mostri pelosi.

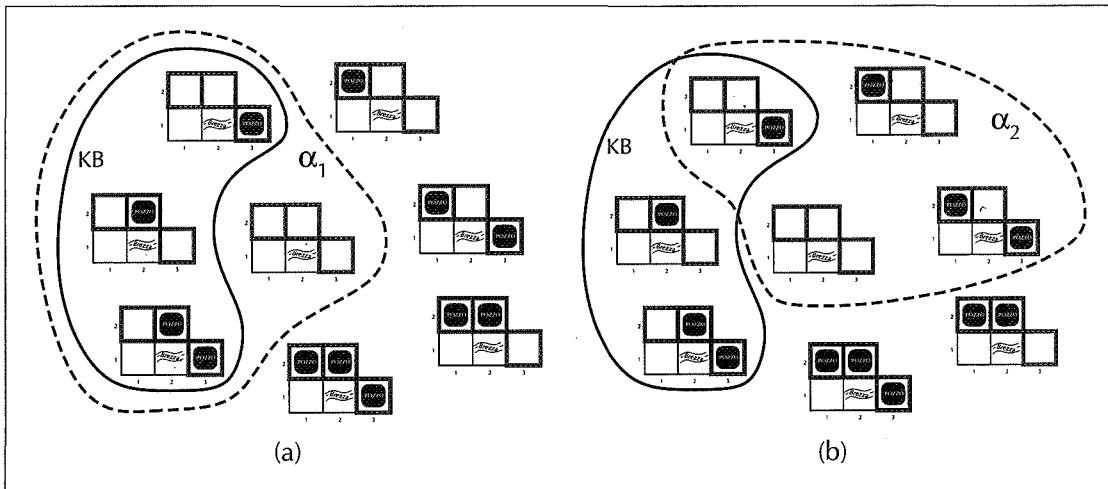


Figura 7.5 I possibili modelli per la presenza di pozzi nelle stanze [1,2], [2,2] e [3,1], non avendo rilevato nulla in [1,1] e una brezza in [2,1]. (a) Modelli della base di conoscenza e α_1 (nessun pozzo in [1,2]). (b) Modelli della base di conoscenza e α_2 (nessun pozzo in [2,2]).

Quindi $KB \models \alpha_1$: non c'è pozzo in [1,2]. Possiamo anche verificare che

in alcuni modelli in cui KB è vero, α_2 è falso.

Quindi $KB \not\models \alpha_2$: l'agente non può concludere che non ci sia un pozzo in [2,2] (e neppure che vi sia, peraltro).⁴

Quest'esempio non solo illustra l'implicazione logica, ma mostra anche come la sua definizione può essere applicata per derivare conclusioni, ovvero per eseguire inferenze logiche. L'algoritmo di inferenza riportato nella Figura 7.5 è chiamato model checking, perché enumera tutti i possibili modelli per verificare che α sia vero in tutti quelli in cui è vera la KB .

Per comprendere meglio implicazione e inferenza, potreste provare a pensare che l'insieme di tutte le conseguenze di KB sia come un pagliaio e α sia un ago. L'implicazione è come dire che l'ago si trova nel pagliaio; l'inferenza equivale a trovarlo. Questa distinzione viene rappresentata nella notazione formale: se un algoritmo di inferenza i può derivare α da KB , scriviamo

$$KB \vdash_i \alpha,$$

e si dice che " α è derivato da KB attraverso i " o anche " i deriva α da KB ".

inferenze logiche
model checking

⁴ L'agente può però calcolare la probabilità che ci sia un pozzo in [2,2]; vedremo come nel Capitolo 13.

corretto
preserva la verità

completezza



grounding



Un algoritmo di inferenza che deriva solo formule implicate viene chiamato corretto; si dice anche che preserva la verità. La correttezza è una proprietà decisamente auspicabile. Una procedura di inferenza non corretta, essenzialmente, genera formule a suo piacimento: è come dire che annuncia la scoperta di aghi inesistenti. È facile verificare che il model checking, quando è applicabile,⁵ è una procedura corretta.

Un'altra proprietà desiderabile è la completezza: un algoritmo di inferenza è completo se può derivare ogni formula implicata. Nei pagliai reali, che hanno un'estensione finita, è abbastanza ovvio che un esame sistematico può sempre decidere se l'ago vi è contenuto oppure no. In molte basi di conoscenza, però, il pagliaio delle conseguenze è infinito, e la completezza diventa una questione importante.⁶ Fortunatamente, esistono procedure di inferenza complete per logiche abbastanza espressive da gestire molte basi di conoscenza.

Abbiamo descritto un processo di ragionamento le cui conclusioni sono garantite vere in qualsiasi mondo in cui sono vere le premesse; in particolare, se KB è vera nel mondo reale, allora ogni formula α derivata da KB con un procedimento di inferenza corretto è anch'essa vera nel mondo reale. Così mentre un processo di inferenza lavora a livello di "sintassi", ovvero di configurazioni fisiche interne come i bit nei registri o i pattern elettrici nel cervello, il processo corrisponde alla relazione esistente laddove qualche aspetto del mondo reale accade⁷ in virtù del fatto che accade qualche altro aspetto del mondo reale. Questa corrispondenza tra mondo e rappresentazione è illustrata nella Figura 7.6.

L'ultimo aspetto da considerare, occupandosi di agenti logici, è quello del grounding (che potremmo tradurre radicamento o ancoramento): il legame, se esiste, tra i processi di ragionamento logico e l'ambiente reale in cui si trova l'agente. In particolare, come facciamo a sapere che la KB è vera nel mondo reale? Dopo tutto, si tratta solo di "sintassi" nella mente dell'agente. Questa è una questione filosofica su cui sono stati scritti molti libri: la riprenderemo nel Capitolo 26. Una risposta semplice è che il legame è creato dai sensori dell'agente. Ad esempio, il nostro agente nel mondo del wumpus ha un rilevatore di odori. Il programma agente crea una formula adeguata ogni volta che viene percepito un odore. Di conseguenza, ogni volta che quella formula è nella base di conoscenza, è vera nel mondo reale. Il significato e la verità delle formule percettive sono così definite dai processi di per-

⁵ Il model checking funziona se lo spazio dei modelli è finito: ad esempio, nei mondi del wumpus di estensione limitata. Nell'aritmetica, d'altra parte, lo spazio dei modelli è infinito: anche se ci limitiamo a considerare numeri interi, ci sono sempre infinite coppie di valori per x e y tali che $x + y = 4$.

⁶ Confrontate questo caso con quello degli spazi di ricerca infiniti che abbiamo visto nel Capitolo 3, in cui la ricerca in profondità non è completa.

⁷ Come scrisse Wittgenstein (1922) nel suo famoso *Tractatus*: "il mondo è tutto ciò che accade".

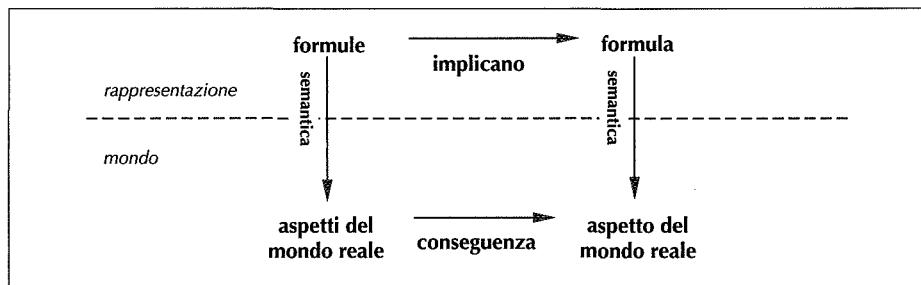


Figura 7.6 Le formule sono configurazioni fisiche dell'agente, e il ragionamento è il processo di costruzione di nuove configurazioni partendo dalle vecchie. Il ragionamento logico dovrebbe assicurare che le nuove configurazioni rappresentino aspetti del mondo che sono effettive conseguenze degli aspetti del mondo rappresentati dalle vecchie configurazioni.

cezione e di costruzione sintattica che le producono. E per quanto riguarda il resto della conoscenza dell'agente, ad esempio circa il fatto che il wumpus causa la presenza di un pungente fetore nelle stanze adiacenti? Questa non è una rappresentazione diretta di una singola percezione ma una regola generale, forse derivata dall'esperienza percettiva ma nient'affatto identica alla semplice affermazione di tale esperienza. Regole generali come questa sono prodotte da un processo di costruzione di formule chiamato apprendimento, che sarà l'argomento della Parte VI nel 2° volume. L'apprendimento è soggetto a errori: potrebbe darsi che i wumpus puzzino sempre tranne il 29 febbraio degli anni bisestili, giorno in cui fanno il bagno. Di conseguenza KB potrebbe non essere vera nel mondo reale, ma questo non toglie che, se si adottano buone procedure di apprendimento, ci siano buone ragioni di essere ottimisti. /

7.4 Calcolo proposizionale: una logica molto semplice

Presentiamo ora una logica molto semplice, chiamata calcolo proposizionale.⁸ Ci occuperemo della sua sintassi e della semantica, cioè di come si può determinare il valore di verità delle formule. Passeremo poi a considerare l'implicazione, cioè la relazione tra due formule di cui una è la conseguenza logica dell'altra, e questo ci porterà a formulare un semplice algoritmo per l'inferenza logica. Tutto questo si svolgerà, naturalmente, nel mondo del wumpus.

calcolo proposizionale

⁸ Il calcolo proposizionale prende anche il nome di logica di Boole, dal matematico George Boole (1815-1864).

formule atomiche

simbolo proposizionale

formule complesse
connettivi logicinegazione
letterale

congiunzione

disgiunzione

implicazione
premessa
conclusione

bicondizionale

Sintassi

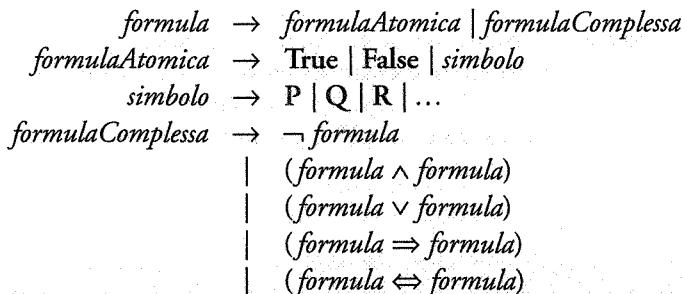
La sintassi del calcolo proposizionale definisce le formule accettabili. Le **formule atomiche** rappresentano gli elementi sintattici indivisibili e consistono di un **singolo simbolo proposizionale**. Ogni simbolo rappresenta una proposizione che può essere vera o falsa. Per i simboli useremo delle lettere maiuscole: P , Q , R e così via. I nomi sono arbitrari, ma spesso li sceglieremo in modo da essere significativi per il lettore: ad esempio, potremo scrivere $W_{1,3}$ per indicare la proposizione che il wumpus si trova in [1,3] (ricordate sempre che i simboli sono *atomici*, questo significa che $W_{1,3}$ non si può scomporre nei suoi componenti W , 1 e 3). Due simboli proposizionali hanno sempre un valore prefissato: *True* è la proposizione sempre vera, *False* quella sempre falsa.

Si possono costruire **formule complesse** partendo da formule più semplici grazie all'uso dei **connettivi logici**. Ce ne sono cinque di uso comune.

- (not): una formula come $\neg W_{1,3}$ è chiamata la **negazione** di $W_{1,3}$. Si può usare il termine **letterale (literal)** per indicare una formula atomica (letterale positivo) o una formula atomica negata (letterale negativo).
- ∧ (and): una formula il cui connettivo principale è \wedge , come $W_{1,3} \wedge P_{3,1}$, prende il nome di **congiunzione**; le sue due parti si chiamano **congiunti**. Il simbolo \wedge ricorda la "A" di "And".
- ∨ (or): una formula che usa \vee , come $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, è una **disgiunzione** composta dai due **disgiunti** ($W_{1,3} \wedge P_{3,1}$) e $W_{2,2}$. Storicamente, il simbolo \vee deriva dal latino "vel", che significa "oppure"; per alcuni sarà più facile ricordarlo semplicemente come un \wedge rovesciato.
- ⇒ (implicazione): una formula come $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ si chiama **implicazione** (o condizionale). La sua **premessa** o **antecedente** è $(W_{1,3} \wedge P_{3,1})$ e la sua **conclusione** o **conseguente** è $\neg W_{2,2}$. Le implicazioni sono anche dette **regole** o **asserzioni if–then**. In altri testi per indicare l'implicazione si usa il simbolo insiemistico \supset o la freccia sottile \rightarrow .
- ↔ (se e solo se): la formula $W_{1,3} \leftrightarrow \neg W_{2,2}$ è una **equivalenza**, chiamata anche **bicondizionale**.

La Figura 7.7 fornisce una grammatica formale del calcolo proposizionale; leggete pag. 631 se non avete familiarità con la notazione BNF.

Noteate che la grammatica è molto precisa circa le parentesi: ogni formula che usa i connettivi deve esserne racchiusa. Questo assicura che la grammatica sia totalmente priva di ambiguità; significa anche che saremo obbligati a scrivere, ad esempio, $((A \wedge B) \Rightarrow C)$ invece di $A \wedge B \Rightarrow C$. Per migliorare la leggibilità spesso ometteremo le parentesi, affidandoci a una definizione precisa dell'ordine di esecuzione dei connettivi. Questo è quanto si fa in aritmetica; $ab + c$ si legge $((ab) + c)$

**Figura 7.7** Una grammatica delle formule del calcolo proposizionale in forma BNF (Backus–Naur Form).

e non $a(b + c)$ proprio perché la precedenza della moltiplicazione è superiore a quella dell'addizione. Dalla più alta alla più bassa, l'ordine di precedenza dei connettivi del calcolo proposizionale è: \neg , \wedge , \vee , \Rightarrow e \Leftrightarrow . Ne consegue che la formula

$$\neg P \vee Q \wedge R \Rightarrow S$$

è equivalente a

$$((\neg P) \vee (Q \wedge R)) \Rightarrow S.$$

La precedenza non elimina l'ambiguità in formule come $A \wedge B \wedge C$, che possono essere lette $((A \wedge B) \wedge C)$ oppure $(A \wedge (B \wedge C))$. Dato che queste due letture hanno lo stesso significato in base alla semantica che vedremo tra poco, formule come $A \wedge B \wedge C$ sono ammesse; così come $A \vee B \vee C$ e $A \Leftrightarrow B \Leftrightarrow C$. Al contrario, formule come $A \Rightarrow B \Rightarrow C$ non sono legali, perché le due letture hanno significati diversi; in questo caso è indispensabile usare le parentesi. Infine, talvolta useremo parentesi quadre anziché tonde nel caso questo aiuti a rendere più chiara la formula.

Semantica

Dopo aver specificato la sintassi del calcolo proposizionale, passiamo ora alla semantica. Con questo termine si intendono le regole usate per determinare il valore di verità di una formula nei confronti di un particolare modello. Nel calcolo proposizionale, un modello fissa semplicemente il valore di verità *true* o *false* di ogni simbolo proposizionale. Per esempio, se le formule nella base di conoscenza fanno uso dei simboli $P_{1,2}$, $P_{2,2}$, e $P_{3,1}$, un modello possibile è

$$\{m_1 = \{P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{true}\}\}.$$

Con tre simboli ci sono $2^3 = 8$ possibili modelli, esattamente quelli mostrati nella Figura 7.5. Notate, comunque, che una volta fissata la sintassi i modelli diventano oggetti puramente matematici senza nessuna relazione particolare con i mondi e i wumpus. $P_{1,2}$ è semplicemente un simbolo: potrebbe significare “c'è un pozzo nella posizione [1,2]” o “sarò a Parigi oggi e domani”.

La semantica del calcolo proposizionale deve specificare come si fa, dato un modello, a calcolare il valore di verità di qualsiasi formula. Questo viene fatto ricorsivamente. Tutte le formule sono costruite partendo da formule atomiche e applicando i cinque connettivi; di conseguenza dobbiamo specificare come calcolare la verità delle formule atomiche e di quelle costruite con ognuno dei connettivi. Per le formule atomiche è facile:

- ◆ *True* è vero in ogni modello e *False* è falso in ogni modello;
- ◆ il valore di verità di ogni altro simbolo proposizionale dev'essere specificato direttamente nel modello. Ad esempio, nel modello m_1 che abbiamo definito qui sopra, $P_{1,2}$ è falso.

Per le formule complesse, abbiamo regole come

- ◆ per ogni formula s e ogni modello m , la formula $\neg s$ è vera in m se è solo se s è falsa in m .

tabella di verità

Queste regole riducono il calcolo della verità delle formule complesse a quello di formule più semplici. Le regole per ogni connettivo possono essere riassunte in una tabella di verità che specifica i valori di verità di una formula complessa per ogni possibile configurazione dei suoi componenti. Le tabelle di verità dei cinque connettivi logici sono fornite nella Figura 7.8. Usando queste tabelle, il valore di verità di ogni formula s può essere calcolato per ogni modello m con un semplice processo di valutazione ricorsiva. Per esempio, la formula $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, valutata in m_1 , corrisponde a $true \wedge (false \vee true) = true \wedge true = true$. Nell'Esercizio 7.3 vi sarà richiesto di scrivere l'algoritmo $CP\text{-VERO?}(s, m)$, che calcola il valore di verità di una formula di calcolo proposizionale s in un modello m .

Abbiamo già detto che una base di conoscenza consiste in un insieme di formule. Ora possiamo constatare che una base di conoscenza logica è precisamente la congiunzione di tutte quelle formule: in altre parole, se prendiamo una KB vuota ed eseguiamo $TELL(KB, S_1) \dots TELL(KB, S_n)$ alla fine avremo $KB = S_1 \wedge \dots \wedge S_n$. Questo significa che possiamo considerare basi di conoscenza e formule logiche in modo intercambiabile.

Le tabelle di verità per “and”, “or” e “not” corrispondono al significato intuitivo delle corrispondenti parole inglesi “e”, “oppure” e “non”. La principale causa di confusione sta nel fatto che la formula $P \vee Q$ è vera quando P è vera, o lo è Q , o anche quando lo sono entrambe. C'è un diverso connettivo chiamato “or esclusivo” (o “xor” in breve) che restituisce falso quando entrambi i disgiunti sono veri.⁹ Non c'è consenso generale sul simbolo da utilizzare per indicare l'or esclusivo; due scelte possibili sono \vee e \oplus .

⁹ Il latino ha una parola distinta, *aut*, per indicare l'or esclusivo.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Figura 7.8 Tabelle di verità dei cinque connettivi logici. Per usare la tabella nel calcolare, ad esempio, il valore di $P \vee Q$ quando P è vero e Q falso, per prima cosa cercate nelle due colonne a sinistra la riga in cui P vale true e Q false (in questo caso, la riga è la terza). Una volta trovata la riga giusta incrociate con la colonna $P \vee Q$ per vedere il risultato: true. Si può anche dire che ogni riga della tabella corrisponde a un modello, e i valori di verità in ogni colonna indicano se la formula corrispondente è vera in tale modello.

La tabella di verità per \Rightarrow all'inizio potrebbe sembrare strana, perché non corrisponde al significato intuitivo di “ P implica Q ” o “se P , allora Q ”. Prima di tutto, occorre dire che il calcolo proposizionale non richiede alcuna relazione di causa-effetto, e a dire il vero nessuna relazione in assoluto tra P e Q . La frase “5 è dispari implica che Tokyo è la capitale del Giappone”, per quanto sia strana da pronunciare, è una formula vera del calcolo proposizionale. Un altro punto che può generare confusione è che ogni implicazione è vera se il suo antecedente è falso. Ad esempio, “5 è pari implica che Sam è intelligente” è una formula vera, indipendentemente dall'effettivo quoziente intellettuale di Sam. Questa può sembrare una scelta bizzarra, ma si deve pensare che la formula “ $P \Rightarrow Q$ ” in effetti significa “se P è vero, allora sostengo che lo è anche Q . In caso contrario non faccio alcuna asserzione”. L'unico modo perché questa formula risulti falsa è che P sia vera ma Q falsa.

La tabella di verità mostra che un bicondizionale, $P \Leftrightarrow Q$, è vero quando lo sono sia $P \Rightarrow Q$ che $Q \Rightarrow P$. Spesso in questo caso si usa l'espressione “ P se e solo se Q ” o, usando una notazione di derivazione anglosassone, “ P iff Q ”. Le regole del mondo del wumpus si esprimono bene usando \Leftrightarrow . Ad esempio, affinché in una stanza ci sia movimento d'aria è necessario che in una delle stanze adiacenti ci sia un pozzo, ma d'altronde la brezza si può rilevare solamente in tal caso. Di conseguenza possiamo scrivere un bicondizionale come

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

dove $B_{1,1}$ significa che si percepisce una brezza in [1,1]. Notate che l'implicazione monodirezionale

$$B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})$$

è vera nel mondo del wumpus, ma incompleta: infatti non esclude i modelli in cui $B_{1,1}$ è falsa e $P_{1,2}$ è vera, ciò che viola le regole. Si può dire che l'implicazione richiede la presenza di pozzi se c'è spostamento d'aria, mentre il bicondizionale impone anche la loro assenza se la brezza non c'è.

Una semplice base di conoscenza

Ora che abbiamo definito la semantica del calcolo proposizionale, possiamo costruire una base di conoscenza per il mondo del wumpus. Per semplicità ci occuperemo solo dei pozzi, lasciando il wumpus stesso come esercizio per i lettori. Forniremo abbastanza conoscenza da svolgere l'inferenza che abbiamo descritto informalmente nel Paragrafo 7.3.

Prima di tutto dobbiamo scegliere un vocabolario per i simboli proposizionali. Per ogni i, j :

- ♦ $P_{i,j}$ è vero se c'è un pozzo in $[i, j]$
- ♦ $B_{i,j}$ è vero se si percepisce una brezza in $[i, j]$.

La base di conoscenza include le seguenti formule, che abbiamo etichettato per comodità.

- ♦ In $[1,1]$ non ci sono pozzi:

$$R_1 : \neg P_{1,1}.$$

- ♦ In una stanza si percepisce brezza se e solo se c'è un pozzo in una stanza adiacente. Questo dev'essere specificato per ogni locazione; per ora indichiamo solo quelle rilevanti:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}).$$

- ♦ Le formule qui sopra sono vere in tutti i mondi del wumpus. Ora aggiungiamo le percezioni relative alla brezza raccolte nelle prime due stanze visitate dall'agente nello specifico mondo in cui si trova, ponendoci così nella situazione rappresentata nella Figura 7.3(b):

$$R_4 : \neg B_{1,1} \quad (\text{non c'è brezza in } 1,1)$$

$$R_5 : B_{2,1}. \quad (\text{c'è brezza in } 2,1)$$

La base di conoscenza ora consiste nelle formule da R_1 a R_5 . Potremmo anche considerarla composta da una singola formula, la congiunzione $R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$, dato che quest'ultima asserisce che tutte le formule che la compongono sono vere.

Inferenza

Ricorderete che lo scopo dell'inferenza logica è decidere se, data una formula α , $KB \models \alpha$. Ad esempio, $P_{2,2}$ è implicata? Il nostro primo algoritmo per l'inferenza sarà un'implementazione diretta della definizione di implicazione: dovremo enumerare esplicitamente i modelli e verificare che α sia vera in ogni modello in cui la KB lo è. Nel calcolo proposizionale i modelli sono rappresentati da un assegnazione

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	true	true	true	true	false	false						
false	false	false	false	false	false	true	true	true	false	false	false	false
:	:	:	:	:	:	:	:	:	:	:	:	:
false	true	false	false	false	false	false	true	true	false	true	true	false
false	true	false	false	false	false	true	false	true	true	true	true	true
false	true	false	false	false	false	true	true	true	true	true	true	true
false	true	false	false	true	false	false	true	false	false	true	true	false
false	true	false	false	true	false	false	true	false	false	true	true	false
:	:	:	:	:	:	:	:	:	:	:	:	:
true	false	true	true	false	true	false						

Figura 7.9 Una tabella di verità per la base di conoscenza fornita nel testo. KB è vera se sono vere tutte le formule da R_1 a R_5 , il che si verifica solo in 3 delle 128 righe. In tutte e tre $P_{1,2}$ è falsa, per cui si può dire che non c'è alcun pozzo in [1,2]. D'altra parte, in [2,2] potrebbe esserci un pozzo o no.

mento dei valori *true* o *false* a ogni simbolo proposizionale. Ritornando al mondo del wumpus, i simboli sono $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$ e $P_{3,1}$. Con sette simboli, ci sono $2^7 = 128$ possibili modelli; in tre di essi KB è vera (Figura 7.9). In quei tre modelli $\neg P_{1,2}$ è vera, per cui possiamo star certi che non c'è pozzo in [1,2]. D'altra parte, $P_{2,2}$ è vera in due dei tre modelli e falsa nel terzo, quindi non possiamo ancora dire se in [2,2] c'è un pozzo oppure no.

La Figura 7.9 riproduce in una forma più precisa il ragionamento illustrato nella Figura 7.5. La Figura 7.10 mostra un algoritmo generale per calcolare le implicazioni nel calcolo proposizionale. Come la RICERCA-BACKTRACKING citata a pag. 185, TV-IMPLICA? esegue l'enumerazione ricorsiva di uno spazio finito di assegnamenti alle variabili. L'algoritmo è corretto, dato che implementa direttamente la definizione di implicazione, ed è completo, perché funziona con ogni base KB e formula α e termina sempre l'esecuzione: infatti il numero dei modelli da esaminare è sempre finito.

Naturalmente, dire che un numero è finito non significa che sia piccolo: se il numero totale di simboli contenuti in KB e α è n , i modelli saranno 2^n . La complessità temporale dell'algoritmo è quindi $O(2^n)$ (quella spaziale è solamente $O(n)$, perché l'enumerazione viene svolta in profondità). Più avanti vedremo che esistono algoritmi molto più efficienti: sfortunatamente, *ogni algoritmo di inferenza conosciuto per il calcolo proposizionale ha una complessità, nel caso pessimo, esponenziale nelle dimensioni dell'input*. Non ci aspettiamo di ottenere risultati migliori perché l'implicazione proposizionale è co-NP-completa (v. Appendice A).



```

function TV-IMPLICA?(KB,  $\alpha$ ) returns true oppure false
  inputs: KB, la base di conoscenza, una formula di logica proposizionale
   $\alpha$ , la query, una formula di logica proposizionale

  simboli  $\leftarrow$  una lista dei simboli proposizionali contenuti in KB e  $\alpha$ 
  return TV-VERIFICA-TUTTO(KB,  $\alpha$ , simboli, [ ])

function TV-VERIFICA-TUTTO(KB,  $\alpha$ , simboli, modello) returns true oppure false
  if VUOTO?(simboli) then
    if CP-VERO?(KB, modello) then return CP-VERO?( $\alpha$ , modello)
    else return true
  else do
     $P \leftarrow$  PRIMO(simboli); resto  $\leftarrow$  RESTO(simboli)
    return TV-VERIFICA-TUTTO(KB,  $\alpha$ , resto, ESTENDI( $P$ , true, modello) and
      TV-VERIFICA-TUTTO(KB,  $\alpha$ , resto, ESTENDI( $P$ , false, modello))

```

Figura 7.10 Un algoritmo che enumera una tabella di verità per determinare l'implicazione nel calcolo proposizionale: "TV" sta appunto per "tabella di verità". CP-VERO? restituisce true se la formula è vera nel modello. La variabile *modello* rappresenta un modello parziale, in cui sono stati assegnati valori solo ad alcune variabili. La chiamata alla funzione ESTENDI(P , true, *modello*) restituisce un nuovo modello parziale in cui P ha valore true.

Equivalenza, validità e soddisfacibilità

Prima di addentrarci nei dettagli degli algoritmi di inferenza logica, definiamo qualche altro concetto relativo all'implicazione. Come quest'ultima, anche le nozioni che presenteremo si applicano a tutte le forme di logica, ma il modo migliore di introdurle è nel contesto di una logica particolare, come il calcolo proposizionale.

Il primo concetto è quello di **equivalenza logica**: due formule α e β sono logicamente equivalenti se sono vere nello stesso insieme di modelli. Questo si scrive $\alpha \Leftrightarrow \beta$. Ad esempio possiamo mostrare facilmente con le tabelle di verità che $P \wedge Q$ e $Q \wedge P$ sono logicamente equivalenti; altre equivalenze sono riportate nella Figura 7.11. Il loro ruolo è analogo a quello delle identità aritmetiche in matematica. Una definizione alternativa dell'equivalenza è la seguente: date due formule α e β ,

$$\alpha \equiv \beta \quad \text{se e solo se} \quad \alpha \vDash \beta \text{ e } \beta \vDash \alpha$$

(ricordate che \vDash indica l'implicazione).

Il secondo concetto è quello di **validità**. Una formula è valida se è vera in *tutti* i modelli. Ad esempio, la formula $P \vee \neg P$ è valida. Le formule valide sono note anche come **tautologie** e sono necessariamente vere. Dato che la formula costante *True* è vera in tutti i modelli, ogni formula valida è logicamente equivalente a *True*.

$(\alpha \wedge \beta)$	\equiv	$(\beta \wedge \alpha)$ commutatività di \wedge
$(\alpha \vee \beta)$	\equiv	$(\beta \vee \alpha)$ commutatività di \vee
$((\alpha \wedge \beta) \wedge \gamma)$	\equiv	$(\alpha \wedge (\beta \wedge \gamma))$ associatività di \wedge
$((\alpha \vee \beta) \vee \gamma)$	\equiv	$(\alpha \vee (\beta \vee \gamma))$ associatività di \vee
$\neg(\neg \alpha)$	\equiv	α eliminazione della doppia negazione
$(\alpha \Rightarrow \beta)$	\equiv	$(\neg \beta \Rightarrow \neg \alpha)$ contrapposizione
$(\alpha \Rightarrow \beta)$	\equiv	$(\neg \alpha \vee \beta)$ eliminazione dell'implicazione
$(\alpha \Leftrightarrow \beta)$	\equiv	$((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$ eliminazione del bicondizionale
$\neg(\alpha \wedge \beta)$	\equiv	$(\neg \alpha \vee \neg \beta)$ De Morgan
$\neg(\alpha \vee \beta)$	\equiv	$(\neg \alpha \wedge \neg \beta)$ De Morgan
$(\alpha \wedge (\beta \vee \gamma))$	\equiv	$((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$ distributività di \wedge su \vee
$(\alpha \vee (\beta \wedge \gamma))$	\equiv	$((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$ distributività di \vee su \wedge

Figura 7.11 Alcune equivalenze logiche standard. I simboli α , β e γ rappresentano formule arbitrarie del calcolo proposizionale.

A cosa servono le formule valide? Dalla nostra definizione dell'implicazione possiamo derivare il teorema di deduzione, che era già noto agli antichi greci (l'Esercizio 7.4 vi chiederà di dimostrarlo):

date due formule qualsiasi α e β , $\alpha \models \beta$ se e solo se la formula $(\alpha \Rightarrow \beta)$ è valida.

Possiamo pensare che l'algoritmo di inferenza della Figura 7.10 controlli la validità di $(KB \Rightarrow \alpha)$. Parimenti, ogni formula di implicazione valida descrive un'inferenza legittima.

L'ultimo concetto è la soddisfacibilità. Una formula è soddisfacibile se è vera in qualche modello. Per esempio la base di conoscenza che abbiamo scritto poco fa, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, è soddisfacibile perché, come si vede nella Figura 7.9, è vera in tre modelli. Se una formula α è vera in un modello m diciamo che m soddisfa α , o anche che m è un modello di α . La soddisfacibilità può essere verificata enumerando i possibili modelli finché non se ne trova uno che soddisfa la formula. La determinazione della soddisfacibilità delle formule del calcolo proposizionale è stato il primo problema di cui è stata dimostrata la NP-complettezza.

Nell'informatica, in effetti, molti problemi sono in realtà problemi di soddisfacibilità. Tutti i problemi di soddisfacimento di vincoli che abbiamo trattato nel Capitolo 5, per esempio, richiedono in sostanza di verificare se i vincoli sono soddisfacibili da qualche assegnamento delle variabili. Con adeguate trasformazioni, anche i problemi di ricerca possono essere risolti verificandone la soddisfacibilità. Naturalmente, validità e soddisfacibilità sono strettamente connesse: α è valida se e solo se $\neg \alpha$ è insoddisfacibile; di contro, α è soddisfacibile se e solo se $\neg \alpha$ non è valida. Un altro risultato utile è il seguente:

$\alpha \models \beta$ se e solo se la formula $(\alpha \wedge \neg \beta)$ è insoddisfacibile.

teorema di deduzione

soddisfacibilità

soddisfa

SAT



dimostrazione per assurdo
refutazione

Dimostrare β partendo da α e verificando l'insoddisfacibilità di $(\alpha \wedge \neg\beta)$ corrisponde esattamente alla tecnica matematica standard chiamata dimostrazione per assurdo, detta anche dimostrazione per refutazione o per contraddizione. Si dà per scontato che la formula β sia falsa e si dimostra che questo porta a una contraddizione con i noti assiomi α . Questa contraddizione è esattamente ciò che si intende quando si dice che la formula $(\alpha \wedge \neg\beta)$ è insoddisfacibile.

7.5 Schemi di ragionamento nel calcolo proposizionale

regole di inferenza
Modus Ponens

Questo paragrafo presenta degli schemi standard di inferenza che possono essere applicati per derivare catene di deduzioni che portano all'obiettivo desiderato. Questi schemi prendono il nome di regole di inferenza; la più nota è il Modus Ponens, che si scrive come segue:

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta} \text{ Vole} \quad \text{una} \quad \text{vola} \quad \text{in} \quad \text{un} \quad \text{infere} \quad \text{re}$$

~~che~~

La notazione significa che, ogni volta che sono date le formule $\alpha \Rightarrow \beta$ e α , allora si può inferire la formula β . Ad esempio, se sono date le formule $(WumpusDavanti \wedge WumpusVivo) \Rightarrow ScagliaFreccia$ e vale $(WumpusDavanti \wedge WumpusVivo)$, allora si può inferire $ScagliaFreccia$.

Un'altra utile regola di inferenza è l'eliminazione degli and in base alla quale, data una congiunzione, si può inferire uno qualsiasi dei congiunti:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

Per esempio, da $(WumpusDavanti \wedge WumpusVivo)$ si può inferire $WumpusVivo$.

Considerando i possibili valori di verità di α e β , si può facilmente mostrare una volta per tutte che Modus Ponens ed eliminazione degli and sono corrette: di conseguenza queste regole possono essere utilizzate in tutte le istanze a cui si applicano, producendo inferenze corrette senza bisogno di enumerare i modelli.

Tutte le equivalenze logiche della Figura 7.11 possono essere usate come regole di inferenza. Ad esempio, l'equivalenza per l'eliminazione del bicondizionale dà origine alle due regole di inferenza

$$\frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta} \quad \text{e} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Non tutte le regole di inferenza funzionano, come questa, in entrambe le direzioni: ad esempio, non è possibile eseguire Modus Ponens all'indietro per dedurre $\alpha \Rightarrow \beta$ e α da β .

Vediamo ora come si possono utilizzare queste regole di inferenza ed equivalenze nel mondo del wumpus. La base di conoscenza iniziale contiene le formule da R_1 a R_5 , e vogliamo dimostrare $\neg P_{1,2}$, ovvero che non c'è alcun pozzo in [1,2]. Per prima cosa, applichiamo l'eliminazione del bicondizionale a R_2 per ottenere

$$R_6 : (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

Quindi applichiamo l'eliminazione degli and a R_6 per ottenere

$$R_7 : ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

L'equivalenza logica della contrapposizione fornisce

$$R_8 : (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})).$$

Ora possiamo applicare Modus Ponens con R_8 e la percezione R_4 (cioè $\neg B_{1,1}$), ottenendo

$$R_9 : \neg(P_{1,2} \vee P_{2,1}).$$

Infine, applicando la regola di De Morgan, arriviamo alla conclusione

$$R_{10} : \neg P_{1,2} \wedge \neg P_{2,1}.$$

Ovvero, non ci sono pozzi né in [1,2] né in [2,1].

La derivazione qui sopra, composta da una sequenza di applicazioni di regole di inferenza, prende il nome di dimostrazione. Cercare dimostrazioni è esattamente come cercare soluzioni a problemi di ricerca. In effetti, se la funzione successore è definita in modo da generare tutte le possibili applicazioni di regole di inferenza, si possono applicare alla ricerca di dimostrazioni tutti gli algoritmi che abbiamo visto nei Capitoli 3 e 4. La ricerca di una dimostrazione è quindi un'alternativa all'enumerazione dei modelli. La ricerca può procedere in avanti dalla base di conoscenza iniziale, applicando regole di inferenza per derivare la formula obiettivo, oppure può risalire all'indietro da quella, cercando di trovare una catena di regole di inferenza che parta dalla base di conoscenza iniziale. In questo paragrafo presenteremo due famiglie di algoritmi che utilizzano queste tecniche.

Il fatto che l'inferenza nella logica proposizionale sia un problema NP-completo suggerisce che, nel caso pessimo, la ricerca di dimostrazioni non sarà più efficiente dell'enumerazione dei modelli. In molti casi pratici, comunque, trovare una dimostrazione può essere molto efficiente semplicemente perché si possono ignorare le proposizioni irrilevanti, indipendentemente dal loro numero. Ad esempio, la dimostrazione che abbiamo appena fornito per $\neg P_{1,2} \wedge \neg P_{2,1}$ non fa alcuna menzione delle proposizioni $B_{2,1}$, $P_{1,1}$, $P_{2,2}$ o $P_{3,1}$, che possono essere ignorate in tutta sicurezza. Infatti la proposizione obiettivo $P_{1,2}$ compare solo nella formula R_4 , e le altre proposizioni in R_4

dimostrazione



$$\begin{array}{c} \alpha \rightarrow c \rightarrow \text{ra} \vee b \\ (\alpha \wedge b) \rightarrow c \rightarrow \text{ra} \vee b \vee c \end{array}$$

monotonicità

compaiono solo in R_4 e R_2 ; ne consegue che R_1 , R_3 e R_5 non hanno alcun effetto sulla dimostrazione. Lo stesso ragionamento varrebbe anche se dovessimo aggiungere un milione di altre formule alla base di conoscenza: il semplice algoritmo della tabella di verità, al contrario, sarebbe presto sopraffatto dalla crescita esponenziale del numero di modelli.

Questa proprietà dei sistemi logici, in effetti, deriva da una proprietà più fondamentale, che prende il nome di monotonicità. La monotonicità afferma che un insieme di formule collegate dall'implicazione può solo aumentare man mano che si aggiunge informazione alla base di conoscenza.¹⁰ Per qualsiasi coppia di formule α e β ,

$$\text{se } KB \models \alpha \text{ allora } KB \models \beta \models \alpha.$$

Supponiamo ad esempio che la base di conoscenza contenga l'asserzione aggiuntiva β che nel mondo esistono esattamente otto pozzi. Questa conoscenza potrebbe aiutare l'agente a trarre conclusioni aggiuntive, ma non potrà mai invalidare una conclusione α già dedotta, come il fatto che non c'è alcun pozzo nella posizione [1,2]. La monotonicità significa che le regole di inferenza possono essere applicate non appena si trovano nella base di conoscenza le premesse necessarie; le conclusioni di tali regole dovranno essere vere indipendentemente dal resto delle formule contenute nella base di conoscenza.

Risoluzione

Abbiamo detto che le regole di inferenza viste fin qui sono corrette, ma non abbiamo discusso la questione della completezza degli algoritmi di inferenza che le utilizzano. Algoritmi come la ricerca ad approfondimento iterativo (v. pag. 105) sono completi nel senso che arriveranno a qualsiasi obiettivo raggiungibile, ma se le regole di inferenza disponibili sono inadeguate, l'obiettivo non sarà raggiungibile affatto: in altre parole, non esisterà alcuna dimostrazione che usi solo quelle regole. Per esempio, senza la regola di eliminazione del bicondizionale non sarebbe più possibile scrivere la dimostrazione qui sopra. In questo paragrafo introdurremo una singola regola di inferenza, la risoluzione, che unita a qualsiasi algoritmo di ricerca completo dà luogo a un algoritmo di inferenza completo.

Cominceremo ad applicare una versione semplificata della regola nel mondo del wumpus. Consideriamo i passi che hanno portato alla Figura 7.4(a): l'agente ritorna da [2,1] a [1,1] e da lì passa in [1,2] dove percepisce una forte puzza, ma nessun movimento d'aria. Aggiungiamo alla base di conoscenza i seguenti fatti:

$$R_{11}: \neg B_{1,2}.$$

$$R_{12}: B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3}).$$

¹⁰ Le logiche non monotone, che violano la proprietà della monotonicità, esprimono una proprietà comune del ragionamento umano: la possibilità di cambiare idea. Le discuteremo nel Paragrafo 10.7.

Applicando lo stesso processo che in precedenza ci ha portato a R_{10} , possiamo derivare l'assenza di pozzi in [2,2] e [1,3] (ricordate che sappiamo già che non c'è alcun pozzo in [1,1]):

$$R_{13} : \neg P_{2,2}.$$

$$R_{14} : \neg P_{1,3}.$$

Possiamo anche applicare l'eliminazione del bicondizionale a R_3 , seguita dal modus ponens con R_5 , per dedurre il fatto che ci dev'essere un pozzo in [1,1], [2,2] o [3,1]:

$$R_{15} : P_{1,1} \vee P_{2,2} \vee P_{3,1}.$$

Ora giungiamo alla prima applicazione della regola di risoluzione: la formula atomica $\neg P_{2,2}$ in R_{13} risolve con la formula atomica $P_{2,2}$ in R_{15} per fornire

$$R_{16} : P_{1,1} \vee P_{3,1}.$$

In linguaggio naturale possiamo dire che se c'è un pozzo in [1,1], [2,2] o [3,1], e non si trova in [2,2], allora dev'essere in [1,1] o [3,1]. In modo analogo, la formula atomica $\neg P_{1,1}$ in R_1 risolve con $P_{1,1}$ in R_{16} per dare

$$R_{17} : P_{3,1}.$$

Ovvero: se c'è un pozzo in [1,1] o [3,1] e non si trova in [1,1], allora dev'essere in [3,1]. Questi ultimi due passi inferenziali sono esempi della regola di inferenza di risoluzione unitaria,

$$\frac{\ell_1 \vee \dots \vee \ell_k \quad m}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k}, \quad m \Leftarrow \ell_i$$

dove ogni ℓ è una formula atomica (o letterale) e ℓ_i e m sono letterali complementari (cioè, uno è la negazione dell'altro). La regola di risoluzione unitaria, quindi, prende una clausola – cioè una disgiunzione di letterali – e un letterale e produce una nuova clausola. Notate che una singola formula atomica può essere considerata come una disgiunzione formata da un solo letterale, che in questo caso dà origine a una clausola unitaria.

La regola di risoluzione unitaria può essere generalizzata nella regola di risoluzione:

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad \overline{m_1} \vee \dots \vee \overline{m_n}}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee \underbrace{m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}, \quad \begin{array}{c} \ell \\ m \end{array}}$$

risoluzione unitaria

letterali complementari

clausola

clausola unitaria

risoluzione

dove ℓ_i e m_j sono letterali complementari. Se tutte le clausole fossero di lunghezza due potremmo scrivere più semplicemente

$$\frac{\ell_1 \vee \ell_2, \quad \neg \ell_2 \vee \ell_3}{\ell_1 \vee \ell_3}.$$

In definitiva, la risoluzione prende due clausole e ne produce una nuova che contiene tutti i letterali delle due clausole originali tranne i due complementari. Per esempio, nel nostro caso

$$\frac{P_{1,1} \vee P_{3,1}, \quad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

C'è un altro aspetto tecnico che riguarda la regola di risoluzione: la clausola risultante dovrebbe contenere solo una copia di ogni letterale.¹¹ La rimozione delle eventuali copie è chiamata **fattorizzazione**. Ad esempio, se risolviamo $(A \vee B)$ con $(A \vee \neg B)$ otteniamo $(A \vee A)$, che viene ridotto ad A .

La *correttezza* della regola di risoluzione può essere facilmente dimostrata considerando il letterale ℓ_i . Se ℓ_i è vero, allora m_j è falso, e quindi $m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n$ dev'essere vero, perché $m_1 \vee \dots \vee m_n$ è dato. Se ℓ_i è falso, allora $\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k$ dev'essere vero, perché $\ell_1 \vee \dots \vee \ell_k$ è dato. È chiaro che ℓ_i è vero oppure falso, per cui deve verificarsi l'una o l'altra di queste conclusioni, esattamente come afferma la regola di risoluzione.

La cosa più sorprendente della regola di risoluzione è che rappresenta la base di una famiglia di procedure di inferenza *complete*. *Qualsiasi algoritmo di ricerca completo, che applichi solo la regola di risoluzione, può derivare tutte le conclusioni implicite da una base di conoscenza nel calcolo proposizionale*. La risoluzione, tuttavia, è completa in un'accezione particolare. Dato un letterale A vero, non possiamo usare la risoluzione per generare automaticamente la conseguenza $A \vee B$. Possiamo comunque usarla per rispondere alla domanda se la formula $A \vee B$ è vera. Questa viene chiamata completezza per la refutazione, e significa che la risoluzione può sempre essere usata per confermare o confutare una formula, ma non può essere usata per enumerare le formule vere. I prossimi due sottoparagrafi mostreranno come.

fattorizzazione

completezza per la refutazione

¹¹ Se una clausola viene considerata come un *insieme* di letterali, la restrizione è automaticamente rispettata. Usare la notazione degli insiemi per le clausole rende la regola di risoluzione molto più pulita, anche se richiede l'introduzione di notazione aggiuntiva.

Forma normale congiuntiva

La regola di risoluzione si applica solo alle disgiunzioni di formule atomiche, ragion per cui sembrerebbe utilizzabile solo su basi di conoscenza e query composte da simili disgiunzioni. Com'è possibile allora che possa portare a una procedura di inferenza completa per l'intero calcolo proposizionale? La risposta è che ogni formula del calcolo proposizionale è logicamente equivalente a una congiunzione di disgiunzioni di letterali. Una formula così espressa viene detta in **forma normale congiuntiva** o, con l'acronimo inglese, CNF. Più avanti considereremo anche la famiglia ristretta delle formule k -CNF, che hanno esattamente k letterali per clausola:

$$(\ell_{1,1} \vee \dots \vee \ell_{1,k}) \wedge \dots \wedge (\ell_{n,1} \vee \dots \vee \ell_{n,k}).$$

Si può dimostrare che ogni formula può essere trasformata in una formula 3-CNF con un insieme di modelli equivalente.

Invece di fornire la dimostrazione (v. Esercizio 7.10), descriveremo una semplice procedura di conversione. Per illustrarla, convertiremo in CNF la formula $R_2: B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$. Ecco la sequenza dei passi:

1. eliminiamo \Leftrightarrow , rimpiazzando $\alpha \Leftrightarrow \beta$ con $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$:

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1});$$

2. eliminiamo \Rightarrow , rimpiazzando $\alpha \Rightarrow \beta$ con $\neg \alpha \vee \beta$:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg (P_{1,2} \vee P_{2,1}) \vee B_{1,1});$$

3. CNF richiede che \neg si applichi solo ai letterali, per cui lo "muoviamo all'interno" utilizzando ripetutamente le seguenti equivalenze dalla Figura 7.11:

$$\neg(\neg \alpha) \equiv \alpha \text{ (eliminazione della doppia negazione)}$$

$$\neg(\alpha \wedge \beta) \equiv (\neg \alpha \wedge \neg \beta) \text{ (De Morgan)}$$

$$\neg(\alpha \vee \beta) \equiv (\neg \alpha \vee \neg \beta) \text{ (De Morgan)}$$

In quest'esempio ci basta applicare una sola volta la terza regola:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1});$$

4. ora abbiamo una formula che contiene operatori nidificati \wedge e \vee applicati a letterali. Sfruttiamo la legge di distributività della Figura 7.11, distribuendo \vee su \wedge ovunque possibile:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}).$$

La formula originale è ora in CNF, come congiunzione di tre clausole. La sua lettura è molto più ardua, ma può essere usata come input di una procedura di risoluzione.



forma normale
congiuntiva

k-CNF

Un algoritmo di risoluzione

Le procedure di inferenza basate sulla risoluzione sfruttano il principio di dimostrazione per assurdo che abbiamo menzionato alla fine del Paragrafo 7.4. Questo significa che, per dimostrare che $KB \models \alpha$, dimostriamo che $(KB \wedge \neg\alpha)$ è insoddisfacibile. Per far questo dobbiamo giungere a una contraddizione.

Un algoritmo di risoluzione è mostrato nella Figura 7.12. Per prima cosa si converte $(KB \wedge \neg\alpha)$ in CNF; quindi si applica la regola di risoluzione alle clausole risultanti. Ogni coppia che contiene letterali complementari è risolta per produrre una nuova clausola che viene aggiunta all'insieme (se non vi è già presente). Il processo continua finché non si verifica una delle due seguenti possibilità:

- ♦ non è più possibile aggiungere alcuna clausola, nel qual caso KB non implica α ;
- ♦ la risoluzione applicata a due clausole dà come risultato la clausola *vuota*, nel qual caso KB implica α .

La clausola vuota, una disgiunzione senza alcun disgiunto, è equivalente a *False* perché una disgiunzione è vera solo se è vero almeno uno dei disgiunti. Per vedere che la clausola vuota rappresenta una contraddizione, del resto, basta osservare che può avere origine solo dalla risoluzione di due clausole unitarie complementari come P e $\neg P$.

Possiamo applicare la procedura di risoluzione a un'inferenza molto semplice nel mondo del wumpus. Quando l'agente si trova in [1,1] non percepisce alcun spostamento d'aria, per cui non ci possono essere pozzi nelle stanze adiacenti. La base di conoscenza relativa è

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

e noi vogliamo dimostrare α che corrisponde a, poniamo, $\neg P_{1,2}$. Convertendo $(KB \wedge \neg\alpha)$ in CNF, otteniamo le clausole riportate nella parte superiore della Figura 7.13. La parte inferiore della figura mostra le clausole ottenute risolvendo le coppie nella parte superiore. Quando risolviamo $P_{1,2}$ con $\neg P_{1,2}$ otteniamo la clausola vuota, indicata con un piccolo quadratino. L'analisi della Figura 7.13 rivela che molti passi di risoluzione non hanno alcuna utilità: ad esempio, la clausola $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ è equivalente a *True* $\vee P_{1,2}$ che a sua volta è equivalente a *True*. Dedurre che *True* è vero non rappresenta un grande risultato; di conseguenza, ogni clausola in cui compaiono due letterali complementari può essere scartata.

Completezza della risoluzione

Per concludere la nostra discussione sulla risoluzione, dimostriamo ora che CP-RISOLUZIONE è completo. Per fare ciò è utile introdurre la chiusura della risoluzione $RC(S)$ di un insieme di clausole S , che è l'insieme di tutte le clausole derivabili dall'applicazione ripetuta della regola di risoluzione alle clausole in S o a quelle da loro derivate. La chiusura della risoluzione è proprio quello che viene cal-

```

function CP-RISOLUZIONE( $KB, \alpha$ ) returns true oppure false
  inputs:  $KB$ , la base di conoscenza, una formula di calcolo proposizionale
   $\alpha$ , la query, una formula di calcolo proposizionale

   $clausole \leftarrow$  l'insieme di clausole nella rappresentazione CNF di  $KB \wedge \neg\alpha$ 
   $new \leftarrow \{ \}$ 
  loop do
    for each  $C_i, C_j$  in  $clausole$  do
       $resolvents \leftarrow$  CP-RISOLUZIONE( $C_i, C_j$ )
      if  $resolvents$  contiene la clausola vuota then return true
       $new \leftarrow new \cup resolvents$ 
    if  $new \subseteq clausole$  then return false
     $clausole \leftarrow clausole \cup new$ 
  
```

Figura 7.12 Un semplice algoritmo di risoluzione per il calcolo proposizionale. La funzione CP-RISOLUZIONE restituisce l'insieme di tutte le clausole possibili ottenute risolvendo i due input.

colato da CP-RISOLUZIONE come valore finale della variabile clausole. È facile vedere che $RC(S)$ dev'essere finita, perché è finito il numero di clausole distinte che possono essere costruite con i simboli P_1, \dots, P_k di S (ma notate che questo non sarebbe vero senza il passo di fattorizzazione, che elimina le copie dei letterali). Ne consegue che CP-RISOLUZIONE termina sempre l'esecuzione.

Il teorema di completezza per la risoluzione nella logica proposizionale è chiamato teorema di risoluzione ground:

teorema di risoluzione
ground

Se un insieme di clausole è insoddisfacibile, la sua chiusura della risoluzione contiene la clausola vuota.

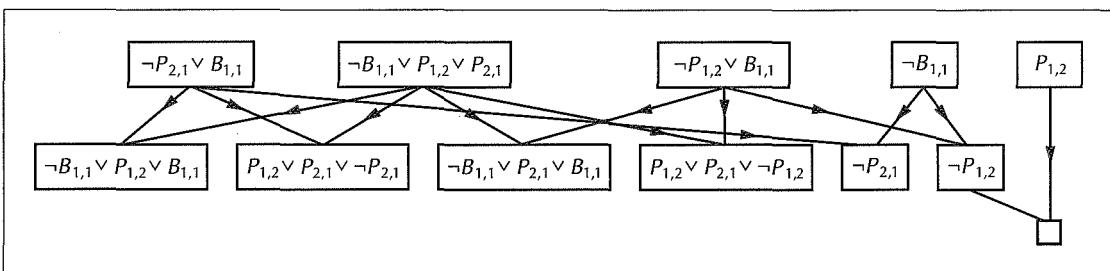


Figura 7.13 Applicazione parziale di CP-RISOLUZIONE a una semplice inferenza nel mondo del wumpus. Come si vede, $\neg P_{1,2}$ è una conseguenza delle prime quattro clausole della riga superiore.

Dimostriamo il teorema per assurdo: se la chiusura $RC(S)$ non contiene la clausola vuota, S è soddisfacibile. In effetti, possiamo costruire un modello di S assegnando adeguati valori di verità per P_1, \dots, P_k . La procedura di costruzione è la seguente.

Per i da 1 a k ,

- se in $RC(S)$ c'è una clausola che contiene il letterale $\neg P_i$ e tutti i suoi altri letterali sono falsi nell'assegnamento scelto per P_1, \dots, P_{i-1} , assegna *false* a P_i ;
- altrimenti, assegna *true* a P_i .

Rimane da dimostrare che quest'assegnamento di P_1, \dots, P_k è un modello di S , posto che $RC(S)$ è chiusa rispetto alla risoluzione e non contiene la clausola vuota. Questo è lasciato come esercizio al lettore.

Concatenazione in avanti e all'indietro

La completezza della risoluzione la rende un metodo di inferenza molto importante. In molte applicazioni pratiche, tuttavia, non è necessario utilizzarne tutta la potenza. Le basi di conoscenza comunemente utilizzate spesso contengono solo clausole di un tipo ristretto, dette **clausole di Horn**. Una clausola di Horn è una disunione di letterali in cui al massimo uno dei letterali è positivo. Per esempio, la clausola $(\neg L_{1,1} \vee \neg Brezza \vee B_{1,1})$, dove $L_{1,1}$ indica che la posizione dell'agente è [1,1], è una clausola di Horn, mentre $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ non lo è.

La restrizione che ci sia al più un solo letterale positivo può sembrare arbitraria e poco interessante, ma in effetti è molto importante per tre ragioni.

1. Ogni clausola di Horn può essere scritta come un'implicazione la cui premessa è una congiunzione di letterali positivi e la cui conclusione è un singolo letterale positivo (v. Esercizio 7.12). Ad esempio, la clausola di Horn $(\neg L_{1,1} \vee \neg Brezza \vee B_{1,1})$ può essere scritta come $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. In quest'ultima forma è molto più facile da leggere: dice che se l'agente si trova in [1,1] e percepisce uno spostamento d'aria, allora la stanza [1,1] è ventosa. Le persone trovano facile leggere e scrivere formule come questa in molti domini della conoscenza.

Le clausole di Horn come questa, che hanno esattamente un letterale positivo, sono chiamate **clausole definite**. Il letterale positivo prende il nome di **testa** e quelli negativi formano il **corpo** della clausola. Una clausola definita senza letterali negativi si limita ad asserire una determinata proposizione, e talvolta viene chiamata **fatto**. Le clausole definite formano la base della **programmazione logica**, che discuteremo nel Capitolo 9. Una clausola di Horn senza letterali positivi può essere scritta in forma di implicazione la cui conclusione vale *False*. Ad esempio, la clausola $(\neg W_{1,1} \vee \neg W_{1,2})$, che specifica che il wumpus non può essere contemporaneamente in [1,1] e [1,2], è equivalente a $W_{1,1} \wedge W_{1,2} \Rightarrow False$. Formule siffatte sono chiamate **vincoli di integrità** nel campo dei database, dove servono a segnalare errori nei dati.

clausole di Horn

clausole definite
testa
corpo
fatto

vincoli di integrità

Negli algoritmi che seguono, daremo per scontato per semplicità che la base di conoscenza contenga solo clausole definite e nessun vincolo di integrità. In questo caso si dice che le basi di conoscenza sono nella forma di Horn.

2. L'inferenza sulle clausole di Horn può essere svolta mediante gli algoritmi di **concatenazione in avanti** e **concatenazione all'indietro**, che vedremo tra poco. Entrambi gli algoritmi sono molto chiari per gli esseri umani, che trovano i passi di inferenza naturali e facili da seguire.
3. Con le clausole di Horn è possibile determinare l'implicazione logica in un tempo che cresce *linearmente* con la dimensione della base di conoscenza.

concatenazione in
avanti
concatenazione
all'indietro

Quest'ultimo fatto è una piacevole sorpresa: significa che l'inferenza logica è computazionalmente economica per molte basi di conoscenza proposizionali usate in pratica.

L'algoritmo di concatenazione in avanti CP-CA-IMPLICA?(KB, q) determina se un singolo simbolo proposizionale q – la query – è implicato da una base di conoscenza composta da clausole di Horn. L'algoritmo comincia dai fatti conosciuti (letterali positivi) nella base di conoscenza. Se tutte le premesse di un'implicazione sono verificate, la sua conclusione è aggiunta all'insieme dei fatti noti. Ad esempio, se $L_{1,1}$ e *Brezza* sono fatti conosciuti e nella base di conoscenza è presente la formula $(L \wedge Brezza) \Rightarrow B_{1,1}$, allora gli si può aggiungere $B_{1,1}$. Questo processo continua finché non viene aggiunta la stessa query q o non è più possibile effettuare alcuna inferenza. L'algoritmo completo è mostrato nella Figura 7.14; la sua caratteristica fondamentale è la complessità temporale lineare.

Il modo migliore di comprendere l'algoritmo è attraverso un esempio e una figura. La Figura 7.15(a) mostra una semplice base di conoscenza composta da clausole di Horn i cui fatti conosciuti sono A e B . La Figura 7.15(b) rappresenta la stessa base di conoscenza disegnata in forma di **grafo AND-OR**. Nei grafi AND-OR, più collegamenti uniti da un arco indicano una congiunzione (ogni collegamento dev'essere verificato), mentre collegamenti multipli senza arco indicano una disgiunzione (basta verificare uno qualsiasi dei collegamenti). È facile capire come funziona la concatenazione in avanti guardando il grafo. Partendo dalle foglie conosciute (qui, A e B) l'inferenza si propaga nel grafo il più lontano possibile. Ogni volta che si incontra una congiunzione, la propagazione attende finché non sono noti tutti i congiunti. Incoraggiamo i lettori a seguire passo passo l'esempio sul grafo.

grafo AND-OR

È facile vedere che la concatenazione in avanti è **corretta**: ogni inferenza sostanzialmente è un'applicazione del **Modus Ponens**. L'algoritmo è anche **completo**: ogni formula atomica implicata sarà derivata. Il modo più facile di verificare quest'ultima asserzione è considerare lo stato finale della tabella *inferenze* dopo che l'algoritmo ha raggiunto un **punto fisso** oltre il quale non è più possibile inferire nulla. La tabella contiene il valore *true* per ogni simbolo inferito durante il processo e *false* per tutti gli altri simboli. Possiamo così considerare la tabella alla stregua

punto fisso

```

function CP-CA-IMPLICA?(KB, q) returns true oppure false
  inputs: KB, la base di conoscenza, un insieme di clausole proposizionali di Horn
            q, la query, un simbolo proposizionale
  variabili locali: conto, una tabella, indicizzata per clausola,
                      che contiene inizialmente il numero di premesse
            inferiti, una tabella, indicizzata per simbolo,
                      in cui ogni elemento è inizialmente false
            agenda, una lista di simboli, che contiene inizialmente
                      quelli noti come veri nella KB

  while agenda non è vuota do
    p  $\leftarrow$  POP(agenda)
    unless inferiti[p] do
      inferiti[p]  $\leftarrow$  true
    for each clausola di Horn c in cui appare la premessa p do
      decrementa conto[c]
      if conto[c] = 0 then do
        if TESTA[c] = q then return true
        PUSH(TESTA[c], agenda)
    return false

```

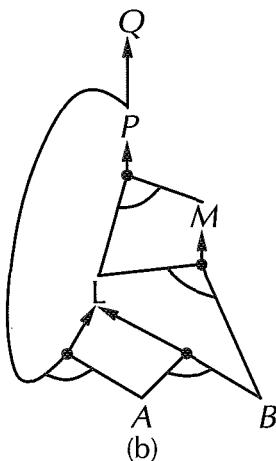
Figura 7.14 L'algoritmo di concatenazione in avanti per il calcolo proposizionale. L'agenda tiene traccia dei simboli che sono noti come veri ma non ancora "processati." La tabella *conto* tiene traccia di quante premesse di ogni implicazione sono ancora sconosciute. Ogni volta che un nuovo simbolo *p* dell'agenda è processato, il conto viene ridotto di uno per ogni implicazione in cui appare la premessa *p* (questo può essere fatto in tempo costante, se KB è stata indicizzata in modo appropriato). Se un conto arriva a zero, significa che tutte le premesse di un'implicazione sono note, dimodoché la sua conclusione può essere aggiunta all'agenda. Infine, dobbiamo tener traccia dei simboli processati; un simbolo non dev'essere aggiunto all'agenda se è già stato considerato in precedenza. Questo permette di evitare di svolgere elaborazioni inutili e previene cicli infiniti nel caso esistano coppie di implicazioni come $P \Rightarrow Q$ e $Q \Rightarrow P$.



di un modello logico; inoltre, *ogni clausola definita nella KB originale è vera in questo modello*. Per verificare questo punto basta assumere l'ipotesi opposta, e cioè che qualche clausola $a_1 \wedge \dots \wedge a_k \Rightarrow b$ sia falsa nel modello. Allora la premessa $a_1 \wedge \dots \wedge a_k$ dev'essere vera, e b falsa: ma questo contraddice l'assunto che l'algoritmo abbia raggiunto un punto fisso! Possiamo quindi concludere che l'insieme di formule atomiche inferite in corrispondenza del punto fisso definisce un modello della KB originale. Ma ogni formula atomica *q* implicata dalla KB dev'essere vera in tutti i suoi modelli, e in particolare in questo: Ne consegue che l'algoritmo avrà necessariamente inferito ogni formula *q* implicata.

$P \Rightarrow Q$
 $L \wedge M \Rightarrow P$
 $B \wedge L \Rightarrow M$
 $A \wedge P \Rightarrow L$
 $A \wedge B \Rightarrow \perp$
 A
 B

(a)



(b)

Figura 7.15 (a) Una semplice base di conoscenza composta da clausole di Horn. (b) Il corrispondente grafo AND-OR.

La concatenazione in avanti è un esempio del concetto generale di **ragionamento guidato dai dati**, un tipo di ragionamento in cui l'attenzione parte dei fatti conosciuti. All'interno di un agente quest'approccio può essere usato per derivare conclusioni partendo dalle percezioni in input, spesso avendo in mente un quesito specifico. Ad esempio, l'agente del mondo del wumpus potrebbe riferire (TELL) le sue percezioni alla base di conoscenza usando un algoritmo di concatenazione in avanti incrementale in cui nuovi fatti possono essere aggiunti alla sua "agenda" per dare inizio a nuove inferenze. Negli esseri umani, man mano che giunge nuova informazione, avviene una certa quantità di ragionamento guidato dai dati. Ad esempio, se sono a casa e sento che è cominciato a piovere, potrei pensare che probabilmente il picnic sarà cancellato. Tuttavia con ogni probabilità non mi verrà in mente che il diciassettesimo petalo della rosa più grande nel giardino del mio vicino si bagnerà; gli esseri umani infatti tengono la concatenazione in avanti sotto stretto controllo, per non essere sopraffatti da un numero enorme di conseguenze irrilevanti.

ragionamento guidato
dai dati

L'algoritmo di concatenazione all'indietro, come suggerisce il nome, parte dalla query e lavora a ritroso. Se è già noto che la query è vera, non occorre fare nulla. In caso contrario, l'algoritmo trova tutte le implicazioni nella base di conoscenza che hanno q come conclusione: se tutte le premesse di una di quelle implicazioni possono essere verificate mediante la concatenazione all'indietro, allora q è vera. Applicato alla query Q della Figura 7.15, l'algoritmo percorre il grafo verso il basso finché non raggiunge un insieme di fatti noti che formano la base della dimostrazione. L'algoritmo dettagliato è lasciato come esercizio; come nel caso della concatenazione, in avanti un'implementazione efficiente avrà complessità lineare.

ragionamento basato sugli obiettivi

La concatenazione all'indietro è una forma di **ragionamento basato sugli obiettivi**. È utile per rispondere a questioni specifiche come “cosa devo fare adesso?” oppure “dove avrò lasciato le chiavi?”. Spesso il costo della concatenazione all'indietro è molto inferiore a quello lineare nelle dimensioni della base di conoscenza, perché il processo coinvolge solo i fatti rilevanti. In generale, un agente dovrebbe dividere il suo lavoro tra i due tipi di ragionamento, limitando la concatenazione in avanti alla generazione dei fatti che saranno probabilmente di interesse per le query che dovranno essere risolte con la concatenazione all'indietro.

7.6 Inferenza proposizionale efficiente

In questo paragrafo descriveremo due famiglie di algoritmi efficienti per l'inferenza proposizionale che si rifanno al model checking: il primo approccio è basato sulla ricerca con backtracking, il secondo sulla ricerca hill-climbing. Questi algoritmi fanno parte della “tecnologia” del calcolo proposizionale. Se è la prima volta che leggete questo capitolo, potete scorrere velocemente questo paragrafo per ritornarvi in seguito.

Gli algoritmi che descriveremo servono a verificare la soddisfabilità. Abbiamo già notato l'analogia tra la ricerca di un modello che soddisfa una formula logica e quella di una soluzione per un problema di soddisfacimento di vincoli, per cui non è molto sorprendente che le due famiglie di algoritmi ricordino da vicino quelli con backtracking visti nel Paragrafo 5.2 e quelli di ricerca locale del Paragrafo 5.3. Questo non toglie che siano molto importanti, dal momento che molti problemi combinatori comuni possono essere ricondotti alla verifica di soddisfabilità di una formula proposizionale. Ogni passo avanti nel campo degli algoritmi di soddisfabilità ha enormi ripercussioni sulla nostra capacità di gestire la complessità in generale.

Un algoritmo con backtracking completo

Il primo algoritmo che presenteremo viene spesso chiamato **algoritmo Davis–Putnam**, a causa del fondamentale articolo di Martin Davis e Hilary Putnam (1960). La nostra versione in effetti è quella descritta da Davis, Logemann e Loveland (1962), per cui useremo l'acronimo DPLL delle iniziali dei quattro autori. DPLL prende come input una formula in forma normale congiuntiva, ovvero un insieme di clausole. Come RICERCA-BACKTRACKING e TV-IMPLICA?, si tratta essenzialmente di un'enumerazione ricorsiva in profondità dei modelli possibili. Rispetto al semplice schema di TV-IMPLICA?, DPLL apporta tre migliorie.

- ◆ *Terminazione anticipata*: l'algoritmo è in grado di determinare se la formula è vera o falsa anche con un modello parzialmente incompleto. Una clausola è vera se *qualsiasi* suo letterale è vero, anche se agli altri non è ancora stato assegnato un valore di verità; di conseguenza l'intera formula può essere giudi-

cata vera ancor prima di completare il modello. Ad esempio, la formula $(A \vee B) \wedge (A \vee C)$ è vera se A è vero, indipendentemente dai valori di B e C . In modo analogo, una formula è falsa se *una qualsiasi* delle sue clausole è falsa, ciò che accade quando sono falsi tutti i letterali che la compongono. Ancora una volta, questo può accadere molto tempo prima che il modello sia completo. La terminazione anticipata fa risparmiare all'algoritmo l'esame di interi sottoalberi nello spazio di ricerca.

- ◆ *Euristica del simbolo puro:* un **simbolo puro** è un simbolo che compare sempre con lo stesso “segno” in tutte le clausole. Ad esempio, nelle tre clausole $(A \vee \neg B)$, $(\neg B \vee \neg C)$ e $(C \vee A)$, il simbolo A è puro perché compare solo come letterale positivo, B è puro perché compare solo il letterale negativo, e C è impuro. È facile vedere che se una formula ha un modello, i valori di verità dei simboli puri saranno assegnati in modo che i corrispondenti letterali valgano *true*, perché questo non potrà mai rendere falsa una clausola. Notate che, determinando la purezza di un simbolo, l'algoritmo può ignorare le clausole che sono già note come vere nel modello costruito fino a quel momento. Per esempio, se il modello contiene $B = \text{false}$, allora la clausola $(\neg B \vee \neg C)$ è già verificata, e C diventa un simbolo puro perché compare solamente in $(C \vee A)$.
- ◆ *Euristica della clausola unitaria:* abbiamo già definito una **clausola unitaria** come una clausola che contiene un solo letterale. Nel contesto di DPLL, il termine indica anche clausole in cui è già stato assegnato dal modello il valore *false* a tutti i letterali tranne uno. Per esempio, se il modello contiene $B = \text{false}$, allora $(B \vee \neg C)$ diventa una clausola unitaria perché è equivalente a $(\text{False} \vee \neg C)$, o solamente $\neg C$. Palesemente, affinché questa clausola sia vera, a C dev'essere assegnato il valore *false*. L'euristica della clausola unitaria esegue tutti questi assegnamenti prima di seguire le restanti diramazioni. Una conseguenza importante di questa euristica è che ogni tentativo di dimostrare (per assurdo) la verità di un letterale che è già presente nella base di conoscenza avrà immediatamente successo (v. Esercizio 7.16). Notate anche che assegnare un valore a una clausola unitaria può creare un'altra: ad esempio, quando si assegna a C il valore *false*, $(C \vee A)$ diventa una clausola unitaria, e questo fa sì che A riceva il valore *true*. Questa “cascata” di assegnamenti forzati è chiamata **propagazione delle unità** e ricorda il processo di concatenazione in avanti con le clausole di Horn. In effetti, se l'espressione CNF contiene solo clausole di Horn, il DPLL essenzialmente non fa che replicare la concatenazione in avanti (v. Esercizio 7.17).

simbolo puro

propagazione delle unità

L'algoritmo DPLL è mostrato nella Figura 7.16: ne abbiamo riportato solo lo schema essenziale, che descrive il processo di ricerca. Non abbiamo fornito dettagli sulle strutture dati che devono essere mantenute in memoria per rendere efficiente ogni passo della ricerca, né le tecniche aggiuntive con cui si può migliorarne l'efficienza: apprendimento delle clausole, euristiche di selezione delle variabili e riavvii casuali. Con la loro inclusione DPLL risulta essere, nonostante la sua età, uno dei più veloci algoritmi di soddisfacibilità disponibili. L'implementazione CHAFF è utilizzata per risolvere problemi di verifica hardware con un milione di variabili.

function DPLL-SODDISFACIBILE?(*s*) **returns** true oppure false

inputs: *s*, una formula del calcolo proposizionale

clausole \leftarrow l'insieme di clausole nella rappresentazione CNF di *s*

simboli \leftarrow una lista di tutti i simboli proposizionali in *s*

return DPLL(*clausole*, *simboli*, [])

function DPLL(*clausole*, *simboli*, *modello*) **returns** true oppure false

if ogni clausola in *clausole* è vera in *modello* **then return** true

if qualche clausola in *clausole* è falsa in *modello* **then return** false

P, *valore* \leftarrow TROVA-SIMBOLO-PURO(*simboli*, *clausole*, *modello*)

if *P* è diverso da null **then return** DPLL(*clausole*, *simboli* - *P*, ESTENDI(*P*, *valore*, *modello*))

P, *valore* \leftarrow TROVA-CLAUSOLA-UNITARIA(*clausole*, *modello*)

if *P* è diverso da null **then return** DPLL(*clausole*, *simboli* - *P*, ESTENDI(*P*, *valore*, *modello*))

P \leftarrow PRIMO(*simboli*); *resto* \leftarrow RESTO(*simboli*)

return DPLL(*clausole*, *resto*, ESTENDI(*P*, true, *modello*)) **or**

 DPLL(*clausole*, *resto*, ESTENDI(*P*, false, *modello*))

Figura 7.16 L'algoritmo DPLL per la verifica della soddisfacibilità di una formula del calcolo proposizionale. TROVA-SIMBOLO-PURO e TROVA-CLAUSOLA-UNITARIA sono descritti nel testo; entrambi restituiscono un simbolo (o null) e il valore di verità da assegnare a tale simbolo. Come TV-IMPLICA?, l'algoritmo lavora su modelli parziali.

Algoritmi di ricerca locale

Abbiamo già visto diversi algoritmi di ricerca locale, tra cui HILL-CLIMBING (pag. 147) e SIMULATED-ANNEALING (pag. 151). Questi algoritmi possono essere applicati direttamente ai problemi di soddisfacibilità, a patto di scegliere la giusta funzione di valutazione. Dato che l'obiettivo è trovare un assegnamento che soddisfa ogni clausola, sarà sufficiente una funzione di valutazione che conta il numero di clausole non soddisfatte. In effetti questa è esattamente la misura utilizzata dall'algoritmo MIN-CONFLICTS, utilizzato per i CSP (pag. 196). Tutti questi algoritmi si muovono nello spazio degli assegnamenti completi, invertendo il valore di verità di un simbolo per volta. Lo spazio normalmente contiene molti minimi locali, per sfuggire ai quali si possono introdurre varie forme di perturbazione casuale. In anni recenti sono stati effettuate molte sperimentazioni per trovare un compromesso soddisfacente tra "avidità" (greediness) e casualità.

Uno dei più semplici ed efficaci algoritmi scaturiti da questa ricerca è WALKSAT (v. Figura 7.17). A ogni iterazione, l'algoritmo seleziona una clausola non soddisfatta e cambia il valore di verità di uno dei suoi simboli. Ci sono due

```

function WALKSAT(clausole, p, max_flips) returns un modello soddisfacente o il fallimento
inputs: clausole, un insieme di clausole del calcolo proposizionale
          p, la probabilità di effettuare un passo di “camminata casuale”,  
tipicamente intorno a 0,5
          max_flips, numero massimo di inversioni di valore prima di abbandonare

modello  $\leftarrow$  un assegnamento casuale di valori di verità ai simboli in clausole
for i = 1 to max_flips do
    if modello soddisfa clausole then return modello
    clausola  $\leftarrow$  una clausola, falsa in modello, scelta casualmente nell’insieme clausole
    con probabilità p inverti il valore in modello di un simbolo scelto casualmente in clausola
    else inverti il valore di verità del simbolo in clausole che  
massimizza il numero di clausole soddisfatte
return fallimento

```

Figura 7.17 L’algoritmo WALKSAT verifica la soddisficiabilità invertendo casualmente i valori delle variabili. Ne esistono diverse versioni.

modi per scegliere tale simbolo, e l’algoritmo utilizza l’uno o l’altro casualmente: (1) un passo “a conflitti minimi” che minimizza il numero di clausole non soddisfatte nel nuovo stato; (2) una “camminata casuale” che pesca il simbolo del tutto casualmente.

Ma WALKSAT, in pratica, funziona? Chiaramente, se restituisce un modello, la formula in input dev’essere necessariamente soddisfacibile. Ma che dire quando il risultato è un *fallimento*? Sfortunatamente, in tal caso non si può sapere se la formula è insoddisfacibile o se si deve concedere all’algoritmo altro tempo. Potremmo provare a impostare il valore di *max_flips* a infinito. In tal caso, è facile dimostrare che WALKSAT prima o poi restituirà un modello, a patto che uno esista e che la probabilità *p* sia > 0 . La ragione è che esiste sempre una sequenza di commutazioni che porta a un assegnamento soddisfacente, e prima o poi la sequenza casuale la dovrà generare. Ma ahinòi, se *max_flips* è infinito e la formula è insoddisfacibile, l’algoritmo non terminerà mai l’esecuzione!

Tutto ciò suggerisce che gli algoritmi di ricerca locale come WALKSAT siano più utili quando ci aspettiamo che una soluzione esista davvero: i problemi discussi nei Capitoli 3 e 5, ad esempio, tipicamente hanno soluzioni. D’altra parte, la ricerca locale non può mai determinare l’*insoddisficiabilità*, ciò che è un requisito per determinare l’implicazione. Per esempio, un agente non può usare la ricerca locale in modo *affidabile* per dimostrare che, in un mondo del wumpus, una particolare stanza è sicura. Al massimo sarebbe in grado di affermare “ci ho pensato per un’ora e non sono riuscito a trovare neppure un mondo possibile in cui quella stanza

non è sicura”. Se l’algoritmo di ricerca locale normalmente è molto veloce nel trovare un modello, quando questo esiste, l’agente potrebbe essere giustificato nel presumere che il fallimento della ricerca indichi l’effettiva insoddisfacibilità. Naturalmente questa congettura non è affatto la stessa cosa di una dimostrazione, e l’agente dovrebbe pensarci bene prima di affidare a essa la sua vita.

Problemi di soddisfacibilità difficili

Esaminiamo ora le prestazioni di DPLL e WALKSAT nella pratica. In particolare siamo interessati ai problemi *difficili*, perché quelli *facili* possono essere risolti da qualsiasi algoritmo. Nel Capitolo 5 abbiamo fatto delle scoperte sorprendenti riguardo certe categorie di problemi: ad esempio quello a n regine, che per lungo tempo è stato ritenuto particolarmente difficoltoso per gli algoritmi di ricerca con backtracking, si è rivelato banalmente semplice per i metodi di ricerca locale come min-conflicts. Questo è dovuto al fatto che le soluzioni sono distribuite densamente nello spazio degli assegnamenti, ed è garantito che ogni assegnamento iniziale abbia una soluzione vicina. Il problema a n regine è quindi facile perché è **poco vincolato**. Quando consideriamo i problemi di soddisfacibilità in forma normale congiuntiva, diciamo che un problema è poco vincolato quando ci sono relativamente poche clausole che, appunto, vincolano le variabili. Ad esempio, ecco una formula 3-CNF generata casualmente¹² con cinque simboli e cinque clausole:

$$\begin{aligned} & (\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \\ & \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C). \end{aligned}$$

16 dei 32 assegnamenti possibili sono modelli della formula, per cui in media basterranno solo due tentativi casuali per trovare un modello.

Che caratteristiche avranno, quindi, i problemi difficili? Presumibilmente, *aumentando* il numero di clausole e tenendo fermo quello dei simboli rendiamo il problema più vincolato, cosicché diventerà più difficile trovare soluzioni. Sia m il numero di clausole e n quello dei simboli. La Figura 7.18(a) mostra la probabilità che una formula 3-CNF casuale sia soddisfacibile in funzione del rapporto clausole/simboli m/n , con n prefissato di valore 50. Com’è prevedibile, per m/n piccoli la probabilità si avvicina a 1, mentre al crescere di m/n tende a zero. Notate che la probabilità diminuisce in modo abbastanza brusco intorno a $m/n = 4,3$. Le formule CNF vicine a questo **punto critico** potrebbero essere descritte come “quasi soddisfacibili” o “quasi insoddisfacibili”. È qui che si trovano i problemi difficili?

La Figura 7.18(b) riporta il tempo d’esecuzione di DPLL e WALKSAT intorno al punto critico, restringendo l’attenzione ai soli problemi *soddisfacibili*. Sono chiare tre cose: prima di tutto, i problemi vicini al punto critico sono *molto* più difficili degli altri problemi casuali; secondo, DPLL è piuttosto efficiente anche in

poco vincolato

punto critico

¹² Ogni clausola contiene tre simboli *distinti* scelti a caso, ognuno dei quali ha il 50% di probabilità di essere negato.

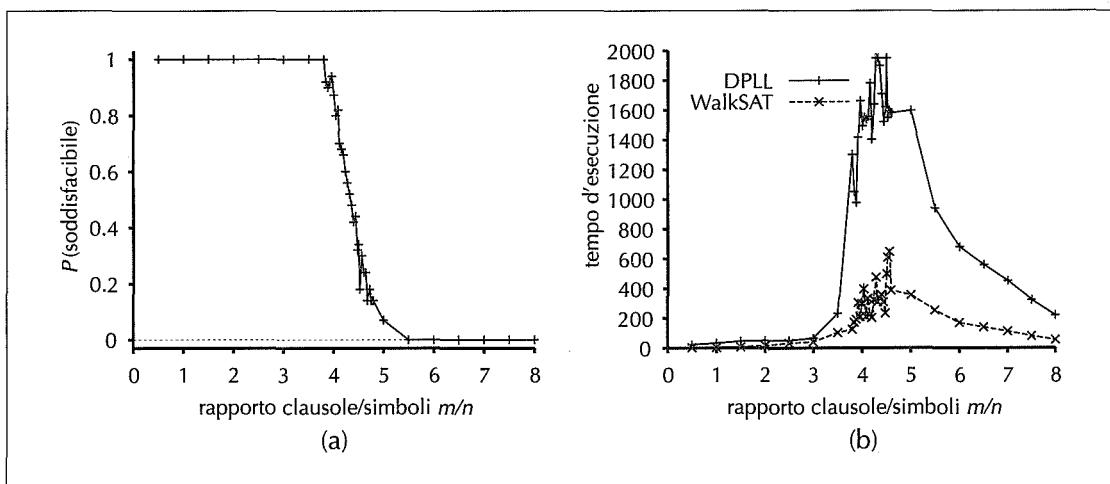


Figura 7.18 (a) Un grafo che mostra la probabilità che una formula 3-CNF casuale con $n = 50$ simboli sia soddisfacibile in funzione del rapporto clausole/simboli m/n . (b) Grafo del tempo medio di esecuzione di DPLL e WALKSAT su 100 formule casuali 3-CNF *soddisfacibili* con $n = 50$, considerando un piccolo intervallo di m/n nell'intorno del punto critico.

presenza dei problemi più difficili, richiedendo in media qualche migliaio di passi anziché i $2^{50} \approx 10^{15}$ richiesti dall'enumerazione delle tabelle di verità; terzo, WALKSAT è molto più veloce di DPLL in tutti i casi.

Naturalmente, tutto ciò si applica solo a problemi generati casualmente: quelli reali non avranno necessariamente la stessa struttura in termini di proporzioni di letterali positivi e negativi, densità delle connessioni tra le clausole etc. Tuttavia, nella pratica, WALKSAT e algoritmi simili sono molto efficaci anche per risolvere problemi reali, spesso tanto quanto i migliori algoritmi specializzati appositamente sviluppati. Risolutori come CHAFF trattano normalmente problemi con migliaia di simboli e milioni di clausole. Queste osservazioni suggeriscono che la combinazione dell'euristica a conflitti minimi e di un comportamento legato alla "camminata casuale" forniscono una capacità *di uso generale* applicabile alla risoluzione della maggior parte delle situazioni che richiedono ragionamento di tipo combinatorio.

7.7 Agenti basati sulla logica proposizionale

In questo paragrafo riprenderemo tutti i concetti che abbiamo visto sin qui e costruiremo un agente in grado di funzionare usando il calcolo proposizionale. Tratteremo due categorie di agenti: quelli che utilizzano gli algoritmi di inferenza e una base di conoscenza, come l'agente generico della Figura 7.1, e quelli che valutano

espressioni logiche direttamente sotto forma di circuiti. Metteremo alla prova entrambe le tipologie di agente nel mondo del wumpus, scoprendo che ognuna di esse mostra gravi limitazioni.

Localizzare pozzi e mostri usando l'inferenza logica

Cominciamo con un agente che ragiona logicamente sulla posizione di pozzi, wumpus e stanze sicure. L'agente parte con una base di conoscenza che definisce la "fisica" del mondo. È noto che la posizione [1,1] non contiene pozzi né wumpus; quindi, $\neg P_{1,1}$ e $\neg W_{1,1}$. Per ogni locazione $[x, y]$ l'agente conosce una formula che specifica quando si può percepire uno spostamento d'aria:

$$B_{x,y} \Leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y}) . \quad (7.1)$$

Per ogni locazione $[x, y]$ l'agente conosce una formula che specifica quando si può percepire una forte puzza:

$$S_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y}) . \quad (7.2)$$

Infine, l'agente sa che in tutto il mondo c'è un solo wumpus. Questa conoscenza viene espressa in due parti; prima di tutto diciamo che ce n'è almeno uno:

$$W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,3} \vee W_{4,4} .$$

Quindi specifichiamo che c'è *al massimo* un wumpus. Un modo di far ciò è specificare che, data una coppia qualsiasi di stanze, una di essere dev'essere per forza priva di mostri. Date n locazioni, risultano $n(n-1)/2$ formule come $\neg W_{1,1} \vee \neg W_{1,2}$. Per un mondo di dimensioni 4×4 , quindi, avremo all'inizio un totale di 155 formule che contengono 64 simboli distinti.

Il programma agente, riportato nella Figura 7.19, comunica (attraverso un'azione TELL) alla sua base di conoscenza ogni nuova percezione riguardante movimenti d'aria e odori molesti (oltre a far ciò deve tener aggiornate delle variabili per memorizzare la stanza corrente e quelle che sono già state visitate: ne parleremo tra poco). Il programma deve quindi scegliere quale stanza visitare tra quelle che fanno parte della frontiera, costituita dall'insieme di stanze adiacenti a quelle già visitate. Una stanza di frontiera $[i, j]$ è *dimostrata sicura* se la base di conoscenza implica la formula $(\neg P_{i,j} \wedge \neg W_{i,j})$. Dopo questa la scelta migliore è una stanza *possibilmente sicura*, caso che si verifica quando l'agente non può dimostrare che la stanza *contiene effettivamente* un pozzo o il wumpus: sarebbe a dire, quando la base di conoscenza *non* implica $(P_{i,j} \vee W_{i,j})$.

Il calcolo dell'implicazione in ASK può essere implementato usando uno qualsiasi dei metodi che abbiamo descritto in questo capitolo. TV-IMPLICA? (v. Figura 7.10) è palesemente inutilizzabile, dato che dovrebbe enumerare 2^{64} righe. DPLL (v. Figura 7.16) esegue le inferenze richieste in pochi millisecondi, grazie principalmente all'euristica di propagazione delle unità. È anche possibile usare WALKSAT, con le solite avvertenze circa la sua incompletezza. Nei mondi del

wumpus, non trovare alcun modello dopo 10.000 commutazioni significa che la formula è certamente insoddisfacibile, per cui non sarà possibile commettere errori dovuti all'incompletezza.

CP-AGENTE-WUMPUS funziona bene in un modo ragionevolmente piccolo, ma la sua base di conoscenza ha una caratteristica che ci lascia profondamente scontenti. *KB* contiene informazioni sulla “fisica” del mondo, nella forma riportata sopra in (7.1) e (7.2), *per ogni singola stanza*. Al crescere delle dimensioni del mondo, la base di conoscenza iniziale dovrà essere sempre più grande. Sarebbe di gran lunga preferibile avere due sole formule in grado di specificare le regole sulla presenza di spostamenti d’aria e di puzza in *tutte* le locazioni. Questo però va oltre le possibilità espressive del calcolo proposizionale: nel prossimo capitolo studieremo un linguaggio logico più potente, con cui sarà facile formulare concetti simili.

Tener traccia della posizione e dell’orientamento

Il programma agente della Figura 7.19 “bara” perché tiene traccia della sua posizione *al di fuori* della base di conoscenza e non usando il ragionamento logico.¹³ Per far ciò nel modo “corretto” dovremmo esprimere la posizione attraverso proposizioni. Un primo approccio potrebbe prevedere l’uso di simboli come $L_{1,1}$ per indicare che l’agente si trova nella posizione [1,1]. La base di conoscenza iniziale potrebbe allora comprendere formule come

$$L_{1,1} \wedge RivoltoADestra \wedge Avanti \Rightarrow L_{2,1}.$$

Possiamo vedere immediatamente che questa soluzione non è accettabile. Infatti se l’agente comincia l’esecuzione in [1,1] rivolto a destra e avanza, la base di conoscenza implicherà sia $L_{1,1}$ (la posizione iniziale) che $L_{2,1}$ (la nuova posizione). Ma queste proposizioni non possono essere vere entrambe! Il problema è che le proposizioni relative alla posizione dovrebbero riferirsi a momenti diversi. Dobbiamo usare $L_{1,1}^1$ per indicare che l’agente si trova in [1,1] all’istante 1, $L_{2,1}^2$ per indicare che si trova in [2,1] all’istante 2 e così via. Anche le proposizioni relative all’orientamento e alle azioni devono dipendere dal tempo. La formula corretta, quindi, è

$$L_{1,1}^1 \wedge RivoltoADestra^1 \wedge Avanti^1 \Rightarrow L_{2,1}^2,$$

$$RivoltoADestra^1 \wedge GiraASinistra^1 \Rightarrow RivoltoInAlto^2,$$

e così via. Come si vede, costruire una base di conoscenza completa e corretta per tener traccia di tutto ciò che succede nel mondo del wumpus non è affatto facile; rimandiamo quindi una discussione esaustiva al Capitolo 10. Il punto cruciale da comprendere è che la base di conoscenza iniziale dovrà contenere formule come le

¹³ Il lettore attento avrà già notato che questo ci ha permesso di gestire con finezza il collegamento tra le percezioni nude, come *Brezza*, e le proposizioni legate alla posizione, come $B_{1,1}$.

```

function CP-AGENTE-WUMPUS(percezione) returns un'azione
  inputs: percezione, una lista, [fetore, brezza, scintillio]
  static: KB, una base di conoscenza che inizialmente
    contiene le "regole fisiche" del mondo del wumpus
    x, y, orientamento, la posizione dell'agente (inizialmente 1,1)
    e il suo orientamento (inizialmente destra)
    visitate, un array che indica quali stanze sono state visitate, inizialmente false
    azione, l'azione più recente effettuata dall'agente, inizialmente null
    piano, una sequenza di azioni, inizialmente vuota

  aggiorna x, y, orientamento, visitate in base all'azione
  if fetore then TELL(KB,  $F_{x,y}$ ) else TELL(KB,  $\neg F_{x,y}$ )
  if brezza then TELL(KB,  $B_{x,y}$ ) else TELL(KB,  $\neg B_{x,y}$ )
  if scintillio then azione  $\leftarrow$  afferra
  else if piano non è vuoto then azione  $\leftarrow$  POP(piano)
  else if per qualche stanza nella frontiera  $[i, j]$ , ASK(KB,  $(\neg P_{i,j} \wedge \neg W_{i,j})$ ) vale true or
    per qualche stanza nella frontiera  $[i, j]$ , ASK(KB,  $(P_{i,j} \vee W_{i,j})$ ) vale false then do
      piano  $\leftarrow$  A*-CERCA-SU-GRAFO(PROBLEMA-DI-ITINERARIO([x, y], orientamento, [i, j],
        visitate))
      azione  $\leftarrow$  POP(piano)
    else azione  $\leftarrow$  una mossa scelta casualmente
  return azione

```

Figura 7.19 Un agente per il mondo del wumpus che usa il calcolo proposizionale per identificare la posizione dei pozzi, del wumpus e delle stanze sicure. La subroutine PROBLEMA-DI-ITINERARIO costruisce un problema di ricerca la cui soluzione è la sequenza di azioni che portano da [*x*, *y*] a [*i*, *j*] passando solo per stanze già visitate.

due qui sopra per ogni istante *t*, oltre che per ogni stanza. Questo significa che, per ogni istante *t* e ogni posizione [*x*, *y*], la base di conoscenza dovrà contenere una formula come

$$L_{x,y}^t \wedge RivoltoADestra^t \wedge Avanti^t \Rightarrow L_{x+1,y}^{t+1}. \quad (7.3)$$

Anche ponendo un limite superiore, come 100, al numero di istanti temporali considerati, la base conterrà comunque decine di migliaia di formule. Lo stesso problema sorgerebbe se aggiungessimo le formule a ogni passo temporale, man mano che se ne presenti la necessità. Questa proliferazione di clausole rende la base di conoscenza illeggibile per un essere umano, ma i risolutori proposizionali possono ancora gestire un mondo del wumpus di dimensione 4×4 con facilità (in effetti, il limite è di circa 100×100 stanze). Tra poco introdurremo la categoria

degli agenti basati su circuito, che offrono una soluzione parziale al problema della proliferazione delle clausole: per avere una soluzione completamente soddisfacente dovremo comunque aspettare fino al Capitolo 8, quando svilupperemo la logica del primo ordine.

Agenti basati su circuiti

Un agente basato su circuiti è un tipo particolare di agente reattivo dotato di stato, secondo la definizione del Capitolo 2. Le percezioni entrano come input in un circuito sequenziale composto da una rete di porte logiche, ognuna delle quali implementa un connettivo logico, e di registri, ognuno dei quali memorizza il valore di verità di una singola proposizione. Gli output del circuito sono costituiti da registri che corrispondono alle azioni: ad esempio, l'output *Afferra* viene impostato a *true* se l'agente vuole afferrare qualcosa. Se l'input *Scintillio* è collegato direttamente all'output *Afferra*, l'agente raccoglierà l'oro che costituisce il suo obiettivo in qualsiasi istante e posizione dovesse avvistarla (v. Figura 7.20).

I circuiti sono valutati in modo analogo a un dataflow: in ogni istante, vengono inseriti gli input e i loro segnali si propagano per tutto il circuito. Una porta logica produce un valore in output non appena ha a disposizione tutti gli input. Questo processo è molto simile alla concatenazione in avanti su un grafo AND-OR come quello della Figura 7.15(b).

Abbiamo detto poco fa che gli agenti basati su circuiti gestiscono il tempo in modo più soddisfacente di quelli basati sull'inferenza proposizionale. Questo è dovuto al fatto che i valori memorizzati nei registri forniscono il valore di verità del corrispondente simbolo proposizionale nell'istante *t corrente*, cosicché non è più necessario averne una copia per ogni diverso istante temporale. Potremmo avere ad esempio un registro *Vivo* che conterrà il valore *true* finché il wumpus è vivo e *false* dopo la sua morte. Questo registro corrisponde al simbolo proposizionale *Vivo^t*, per cui in ogni istante temporale si riferisce a una proposizione differente. Lo stato interno dell'agente, ovvero la sua memoria, viene mantenuto ricollegando l'output di ogni registro al circuito mediante una linea di ritardo (*delay line*), che riporta il suo valore nel passo temporale precedente. La Figura 7.20 mostra un esempio: il valore di *Vivo* è dato dalla congiunzione della negazione di *Ululato* e dal valore nell'istante precedente dello stesso registro *Vivo*. In termini proposizionali, il circuito di *Vivo* implementa il bicondizionale

$$Vivo^t \Leftrightarrow \neg Ululato^t \wedge Vivo^{t-1}. \quad (7.4)$$

Che afferma che il wumpus è vivo nell'istante *t se e solo se* non è stato percepito un ululato nello stesso istante (proveniente da una ferita mortale inflitta in *t - 1*) e se il mostro era vivo in *t - 1*. Possiamo presumere che il circuito sarà inizializzato con il valore di *Vivo* impostato a *true*. Ne consegue che *Vivo* rimarrà vero finché non si verificherà un ululato, istante in cui diventerà falso e lo rimarrà: questo è esattamente il comportamento che desideriamo.

agente basato su circuiti

circuito sequenziale
porte logiche
registri

dataflow

linea di ritardo

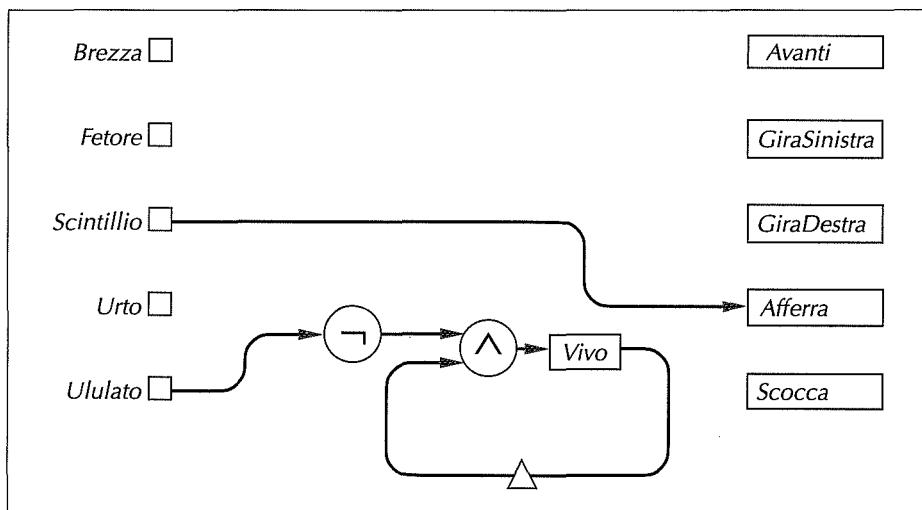


Figura 7.20 Parte di un agente basato su circuiti per il mondo del wumpus. Lo schema mostra gli input, gli output, il circuito per raccogliere l'oro e quello che determina se il wumpus è ancora vivo. I registri sono indicati con dei rettangoli e il ritardo di un passo con un triangolino.

La posizione dell'agente può essere gestita in modo analogo. Abbiamo bisogno di un registro $L_{x,y}$ per ogni x e y ; il suo valore dovrebbe essere *true* se l'agente si trova nella posizione $[x, y]$. Il circuito che calcola il valore di $L_{x,y}$, comunque, è molto più complesso di quello per *Vivo*. Ad esempio, l'agente può trovarsi in $[1,1]$ all'istante t se (a) era già lì al tempo $t-1$ e non si è mosso, o ci ha provato ma ha sbattuto contro un muro; (b) era in $[1,2]$ rivolto in basso e ha fatto un passo avanti; o anche (c) era in $[2,1]$ rivolto a sinistra e ha fatto un passo avanti:

$$\begin{aligned}
 L_{1,1}^t \Leftrightarrow & \quad (L_{1,1}^{t-1} \wedge (\neg Avanti^{t-1} \vee Urto^t)) \\
 & \vee (L_{1,2}^{t-1} \wedge (RivoltInBasso^{t-1} \wedge Avanti^{t-1})) \\
 & \vee (L_{2,1}^{t-1} \wedge (RivoltASinistra^{t-1} \wedge Avanti^{t-1})).
 \end{aligned} \tag{7.5}$$

Il circuito per $L_{1,1}$ è illustrato dalla Figura 7.21. Ogni registro di posizione ha associato un circuito simile. Nell'Esercizio 7.13(b) vi chiederemo di progettare un circuito per gestire le proposizioni relative all'orientamento dell'agente.

I circuiti nelle Figure 7.20 e 7.21 mantengono sempre aggiornato il valore di verità dei registri *Vivo* e $L_{x,y}$. Queste comunque sono proposizioni particolari, perché *il loro corretto valore di verità può sempre essere determinato*. Considerate invece la proposizione $B_{4,4}$, che afferma che nella stanza $[4,4]$ è percepibile una brezza. Benché il valore di questa proposizione sia costante nel tempo, l'agente non lo può

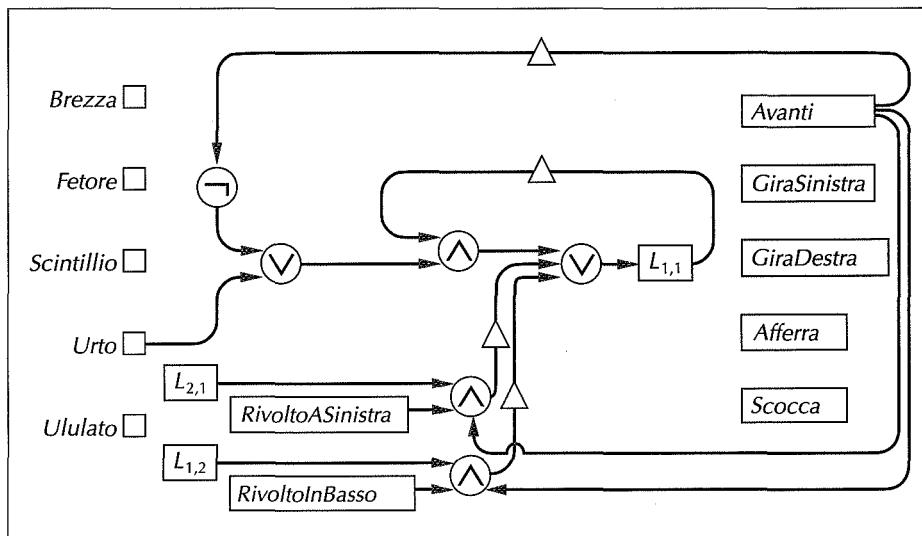


Figura 7.21 Il circuito che determina se l'agente si trova in [1, 1]. Tutti i registri dedicati alla posizione e all'orientamento dell'agente sono collegati a un circuito simile.

determinare finché non ha visitato la stanza [4,4] (o ha dedotto che c'è un pozzo in una posizione adiacente). Il calcolo proposizionale e la logica del primo ordine sono progettate per rappresentare automaticamente i valori vero, falso e indefinito, ma i circuiti no: il registro di $B_{4,4}$ deve contenere *un qualche* valore, *true* o *false*, anche prima della sua effettiva scoperta. Se il valore contenuto nel registro è sbagliato, potrebbe portare l'agente a conclusioni errate. In altre parole, dobbiamo rappresentare tre possibili stati ($B_{4,4}$ può essere noto come vero, noto come falso o indefinito) ma per farlo abbiamo a disposizione un solo bit.

La soluzione del problema è usare due bit invece di uno. $B_{4,4}$ sarà così rappresentato da due registri che chiameremo $K(B_{4,4})$ e $K(\neg B_{4,4})$, dove K sta per “conosciuto” (*known*). Ricordate che questi sono sempre simboli e non espressioni strutturate, anche se il loro aspetto potrebbe farvi pensare altrimenti! Quando sia $K(B_{4,4})$ che $K(\neg B_{4,4})$ sono falsi, significa che il valore di verità di $B_{4,4}$ è ancora sconosciuto (e se sono entrambi veri, significa che c'è qualche errore nella nostra base di conoscenza!). Ora, ogniqualvolta vorremo usare $B_{4,4}$ da qualche parte del circuito, useremo invece $K(B_{4,4})$; parimenti al posto di $\neg B_{4,4}$ useremo $K(\neg B_{4,4})$. In generale rappresenteremo ogni proposizione potenzialmente indeterminata con due proposizioni di conoscenza che stabiliscono se la proposizione sottostante è nota come vera o falsa.

Vedremo tra poco un esempio d'uso delle proposizioni di conoscenza: prima, però, dobbiamo capire come si può determinare il loro valore di verità. Notate che, laddove $B_{4,4}$ ha un valore prefissato, quello di $K(B_{4,4})$ e $K(\neg B_{4,4})$ cambia man mano che l'agente esplora il mondo. Per esempio, $K(B_{4,4})$ inizialmente è falsa, diven-

proposizioni di conoscenza

ta vera non appena lo diventa $B_{4,4}$ (quando l'agente si trova in [4,4] e percepisce uno spostamento d'aria), dopodiché rimane vera per sempre. Abbiamo quindi la formula

$$K(B_{4,4})^t \Leftrightarrow \underbrace{K(B_{4,4})^{t-1}}_{\text{più vero}} \vee (L_{4,4}^t \wedge \text{Brezza}^t). \quad (7.6)$$

Un'equazione simile può essere scritta per $K(\neg B_{4,4})^t$.

Ora che l'agente conosce le stanze ventose, può gestire i pozzi. L'assenza di pozzi in una locazione può essere determinata se e solo se si appura che una qualsiasi delle stanze adiacenti non è ventosa. Per esempio, abbiamo

$$K(\neg P_{4,4})^t \Leftrightarrow K(\neg B_{3,4})^t \vee K(\neg B_{4,3})^t. \quad (7.7)$$

Determinare la *presenza* di un pozzo è più difficile: è necessario rilevare in una stanza adiacente un movimento d'aria che non può essere causato da un altro pozzo:

$$\begin{aligned} K(P_{4,4})^t &\Leftrightarrow (K(B_{3,4})^t \wedge K(\neg P_{2,4})^t \wedge K(\neg P_{3,3})^t) \\ &\quad \vee (K(B_{4,3})^t \wedge K(\neg P_{4,2})^t \wedge K(\neg P_{3,4})^t). \end{aligned} \quad (7.8)$$

Benché i circuiti che determinano la presenza o l'assenza di pozzi siano relativamente complessi, il *numero di porte logiche è costante per ogni locazione*. Questa proprietà è essenziale se vogliamo costruire agenti basati su circuiti in grado di scalare verso l'alto in modo ragionevole. In effetti, si tratta di una proprietà dello stesso mondo del wumpus; si dice che un ambiente ha la caratteristica della località se il valore di verità di ogni proposizione che ci interessa può essere determinato considerando solo un numero finito di altre proposizioni. La località dipende molto strettamente dalle "caratteristiche fisiche" dell'ambiente. Il dominio del campo minato (v. Esercizio 7.11) ad esempio non è locale, perché per determinare la presenza di una mina in una casella potrebbe essere necessario considerare caselle arbitrariamente lontane. Nei domini non locali, gli agenti basati su circuiti non sono sempre applicabili.

C'è un'ultima questione che, fino a questo momento, abbiamo accuratamente evitato: il problema della aciclicità. Un circuito è aciclico se ogni cammino che collega l'output di un registro al suo stesso input comprende al suo interno un elemento di ritardo. Si tratta di un requisito ragionevole perché i circuiti ciclici, intesi come dispositivi fisici, non funzionano affatto! Infatti è sempre possibile che entrino in oscillazioni instabili, fornendo in uscita valori indefiniti. Come esempio di circuito ciclico, considerate la seguente estensione dell'Equazione (7.6):

$$K(B_{4,4})^t \Leftrightarrow K(B_{4,4})^{t-1} \vee (L_{4,4}^t \wedge \text{Brezza}^t) \vee K(P_{3,4})^t \vee K(P_{4,3})^t. \quad (7.9)$$

I disgiunti addizionali $K(P_{3,4})^t$ e $K(P_{4,3})^t$ permettono all'agente di determinare la presenza di spostamenti d'aria in base a quella di pozzi nelle stanze adiacenti, cosa che sembra del tutto ragionevole. Ma sfortunatamente la "ventosità" di una stanza dipende dalla presenza di pozzi e quella dei pozzi, in base a equazioni come la (7.8), dipende dalla presenza di brezze. Ne consegue che il circuito completo terrà dei cicli.

La questione non è che l'Equazione (7.9) estesa sia *sbagliata*. Il problema piuttosto sta nel fatto che le dipendenze incrociate rappresentate da queste equazioni non possono essere risolte semplicemente propagando valori di verità nel circuito logico corrispondente. La versione aciclica che utilizza l'Equazione (7.6), che determina la presenza di spostamenti d'aria con la sola osservazione diretta, è *incompleta* nel senso che l'agente basato su circuiti potrebbe “sapere meno cose” di un agente che applichi l'intera procedura di inferenza. Se per esempio viene percepito uno spostamento d'aria in [1,1], l'agente basato su inferenza potrà concludere che la stessa brezza sarà presente anche in [2,2], cosa impossibile a un agente basato su circuiti che utilizzi l'Equazione (7.6). Costruire un circuito completo è *possibile*: dopotutto, mediante circuiti sequenziali si può emulare qualsiasi computer digitale. La sua realizzazione, comunque, sarebbe significativamente più complessa.

Un confronto

L'agente basato su inferenza e quello basato su circuiti rappresentano gli estremi dichiarativo e procedurale della progettazione, e possono essere confrontati sotto molti punti di vista.

- ♦ *Compattezza*: l'agente basato su circuiti, a differenza di quello che utilizza l'inferenza, non deve possedere una copia separata della sua “conoscenza” per ogni istante temporale. Tutto ciò a cui fa riferimento è l'istante corrente e quello precedente. Entrambi gli agenti hanno bisogno di una copia delle “leggi fisiche” (espresse come formule o circuiti) per ogni locazione, cosa che non li aiuta a scalare bene in ambienti più grandi. Se il mondo contiene molti oggetti che interagiscono in modi complessi, il numero di proposizioni crescerà tanto da rendere impraticabile l'uso di agenti proposizionali: ambienti simili richiedono la potenza espressiva della logica del primo ordine, che presenteremo nel prossimo capitolo. Gli agenti proposizionali di entrambi i tipi sono anche poco indicati per esprimere o risolvere il problema di trovare un cammino che conduce a una stanza sicura (per questo motivo CP-AGENTE-WUMPUS ricorre a un algoritmo di ricerca).
- ♦ *Efficienza computazionale*: nel caso *pessimo*, l'inferenza può richiedere un tempo esponenziale nel numero dei simboli, laddove la valutazione di un circuito prende un tempo che dipende linearmente dalle sue dimensioni (o dalla sua *profondità*, se viene realizzato come dispositivo fisico). Nella *pratica*, comunque, abbiamo visto che DPLL completa le inferenze richieste molto velocemente.¹⁴

¹⁴ In effetti, tutte le inferenze eseguite da un circuito possono essere svolte da DPLL in tempo lineare! Questo perché la valutazione di un circuito, come la concatenazione in avanti, può essere emulata da DPLL usando la regola di propagazione delle unità.

- ◆ *Completezza:* abbiamo già accennato al fatto che l'agente basato su circuiti potrebbe essere incompleto a causa della restrizione sull'aciclicità. Le ragioni di tale incompletezza hanno in realtà radici più profonde. In primo luogo, ricordate che un circuito ha un tempo di esecuzione lineare nelle sue dimensioni: questo significa che, in alcuni ambienti, un circuito completo (cioè uno che calcola il valore di verità di ogni proposizione determinabile) dev'essere esponenzialmente più grande della KB dell'agente basato su inferenza. In caso contrario avremmo trovato il modo di risolvere un problema di implicazione proposizionale in un tempo meno esponenziale, cosa molto improbabile. Una seconda ragione è la natura dello stato interno dell'agente: quello basato su inferenza ricorda ogni percezione e conosce, implicitamente o esplicitamente, ogni formula che deriva dalla percezioni e dalla base di conoscenza iniziale. Ad esempio, dato $B_{1,1}$, l'agente conosce la disgiunzione $P_{1,2} \vee P_{2,1}$, da cui segue $B_{2,2}$. L'agente basato su circuiti, d'altra parte, dimentica tutte le percezioni precedenti ricordando solo le singole proposizioni memorizzate nei registri. Dopo la prima percezione $P_{1,2}$ e $P_{2,1}$ rimarranno così individualmente sconosciute, ragion per cui non verrà tratta alcuna conclusione circa $B_{2,2}$.
- ◆ *Facilità di costruzione:* questo è un aspetto molto importante, difficile da valutare precisamente. Certamente l'autore di questo capitolo ha trovato molto più facile formulare le "leggi fisiche" in modo dichiarativo, laddove la progettazione di circuiti piccoli, aciclici e non troppo incompleti per il rilevamento diretto dei pozzi è sembrata alquanto faticosa.)

In definitiva, sembra necessario accettare dei compromessi tra efficienza computazionale, compattezza, completezza e facilità di costruzione. Un circuito sembra essere la soluzione ottima quando il collegamento tra percezioni e azioni è semplice, come nel caso di *Scintillio* e *Afferra*. Se le relazioni sono più complesse, l'approccio dichiarativo potrebbe essere più adatto. In un dominio come gli scacchi, ad esempio, le regole dichiarative sono concise e facili da formulare (almeno in logica del primo ordine), mentre un circuito in grado di calcolare le mosse direttamente dallo stato della scacchiera sarebbe vasto oltre l'immaginabile.

Il regno animale offre molti esempi di questo tipo di compromessi. Le forme di vita inferiori, che hanno sistemi nervosi molto semplici, sono probabilmente basate su circuiti; quelle più evolute, tra cui gli esseri umani sembrano effettuare inferenze su rappresentazioni esplicite. Questo permette loro di calcolare funzioni agente molto più complesse. Gli umani possiedono anche circuiti per implementare i riflessi, e sono forse capaci di compilare le rappresentazioni dichiarative sotto forma di circuiti quando certe inferenze entrano a far parte dell'abitudine. In questo modo, una progettazione ad *agente ibrido* (v. Capitolo 2) può sfruttare il meglio dei due mondi.

7.8 Riepilogo

Abbiamo presentato gli agenti basati su conoscenza e mostrato come definire una logica utilizzabile per ragionare sul mondo. I punti principali sono i seguenti.

- ◆ Gli agenti intelligenti devono possedere conoscenza del mondo per formulare buone decisioni.
- ◆ All'interno degli agenti la conoscenza è rappresentata mediante **formule** espresse in un **linguaggio di rappresentazione della conoscenza** e memorizzate in una base di conoscenza.
- ◆ Un agente è composto da una base di conoscenza e un meccanismo inferenziale. Le formule sono memorizzate nella base di conoscenza e il meccanismo di inferenza serve a dedurre nuove formule, in base alle quali l'agente può decidere quali azioni intraprendere.
- ◆ Un linguaggio di rappresentazione è definito dalla sua sintassi, che specifica la struttura delle formule, e la sua semantica, che definisce il valore di verità di ogni formula in ogni mondo possibile o modello.
- ◆ La relazione di implicazione tra formule è cruciale per la nostra comprensione del ragionamento. Una formula α implica un'altra formula β se β è vera in tutti i mondi in cui α lo è. Definizioni equivalenti includono la validità della formula $\alpha \Rightarrow \beta$ e la insoddisficiabilità della formula $\alpha \wedge \neg\beta$.
- ◆ L'inferenza è il processo mediante il quale, partendo da un insieme di formule, se ne derivano nuove. Algoritmi di inferenza corretti derivano solo le formule implicate; quelli completi derivano tutte le formule implicate.
- ◆ Il calcolo proposizionale è un linguaggio molto semplice che consiste di simboli proposizionali e connettivi logici. Può gestire proposizioni che sono note come vere, note come false o indefinite.
- ◆ Dato un vocabolario finito di proposizioni, l'insieme dei possibili modelli è finito, per cui l'implicazione più essere verificata enumerando i modelli stessi. Algoritmi di model-checking efficienti per il calcolo proposizionale includono metodi basati su backtracking e ricerca locale e spesso possono risolvere velocemente problemi di grandi dimensioni.
- ◆ Le regole di inferenza sono schemi di ragionamento corretto usati per trovare dimostrazioni. La regola di risoluzione è alla base di un algoritmo di inferenza completo, applicabile a basi di conoscenza in forma normale congiuntiva. La concatenazione in avanti e la concatenazione all'indietro sono algoritmi di ragionamento molto intuitivi, applicabili a basi di conoscenza espresse nella forma di Horn.

- ◆ Si possono costruire due tipi di agenti basati sul calcolo proposizionale: gli **agenti basati su inferenza** sfruttano gli algoritmi di inferenza per tener traccia del mondo e dedurre le sue caratteristiche nascoste, mentre negli **agenti basati su circuiti** le proposizioni sono rappresentate come bit memorizzati in registri e aggiornate secondo le regole della propagazione dei segnali tipica dei circuiti logici.
- ◆ Il calcolo proposizionale è ragionevolmente efficace per alcune attività, ma non scala bene in ambienti di grandi dimensioni perché non ha la potenza espressiva necessaria per gestire in modo compatto il tempo, lo spazio e gli schemi universali che governano le relazioni tra oggetti.

Note storiche e bibliografiche

L'articolo di John McCarthy "Programs with Common Sense" (McCarthy, 1958, 1968) propose la nozione di agenti che usano il ragionamento logico per collegare percezioni e azioni. L'articolo era decisamente a favore della soluzione dichiarativa, sostenendo che dire a un agente quello che deve fare è un modo molto elegante di costruire software. Allen Newell, nell'articolo "The Knowledge Level" (1982), asserisce che gli agenti razionali possono essere descritti e analizzati a un livello astratto definito dalla conoscenza che possiedono piuttosto che dai programmi eseguiti. Gli approcci dichiarativo e procedurale all'IA sono confrontati in Boden (1977). Il dibattito è stato rinfocolato, tra gli altri, da Brooks (1991) e Nilsson (1991).

La logica stessa ha avuto origine nella filosofia e matematica dell'antica Grecia. Negli scritti di Platone si trovano molti principî logici che collegano la struttura sintattica delle frasi con la loro verità o falsità, il loro significato, o la validità delle argomentazioni in cui compaiono. Il primo studio sistematico della logica è quello di Aristotele, il cui lavoro fu raccolto dai discepoli dopo la morte, avvenuta nel 322 a. C., sotto forma di un trattato chiamato *Organon*. I **sillogismi** di Aristotele sono quelli che oggi chiameremo regole di inferenza. Benché i sillogismi includessero elementi sia della logica proposizionale che di quella del primo ordine, secondo gli standard moderni il sistema nella sua interezza era alquanto debole, non permettendo (a differenza della moderna logica proposizionale) l'applicazione di schemi di inferenza a formule di complessità arbitraria.

La scuola di Megara e quella stoica, strettamente imparentate, hanno avuto origine nel V secolo a. C. e sono rimaste attive per molti secoli. Le due scuole hanno introdotto lo studio sistematico dell'implicazione e di altri costrutti base, usati tuttora nel calcolo proposizionale moderno. Le tabelle di verità per la definizione dei connettivi logici sono dovute a Filone di Megara. Gli stoici hanno indicato cinque regole di inferenza base "valide senza dimostrazione", e una di essere era quello che oggi chiamiamo Modus Ponens. A partire da quelle cinque hanno derivato un buon numero di altre regole, applicando tra l'altro il teorema di deduzione (pag. 271), e definendo il concetto di dimostrazione in modo molto più chiaro di Aristotele. Gli stoici sostenevano che la loro logica era completa nel senso che era in grado di catturare tutte le inferenze valide, ma le fonti che sono sopravvissute

sono troppo frammentarie per determinare la verità di quest'affermazione. Un buon resoconto della storia della logica delle scuole di Megara e stoica è fornito da Benson Mates (1953).

L'idea di ridurre l'inferenza logica a un processo puramente meccanico applicato a un linguaggio formale risale a Wilhelm Leibniz (1646–1716). La sua logica matematica comunque era severamente carente, e oggi viene ricordato più per aver introdotto quelle idee come obiettivi che per il suo contributo alla loro realizzazione.

George Boole (1847) introdusse il primo sistema completo e funzionale di logica formale nel suo libro *L'analisi matematica della logica*. La logica di Boole era modellata sulla classica algebra dei numeri reali e usava come metodo principale di inferenza la sostituzione di espressioni logicamente equivalenti. Benché il sistema di Boole non riuscisse ancora a esprimere tutta la potenza del calcolo proposizionale, gli era abbastanza vicina perché altri matematici potessero riempire rapidamente le lacune restanti. La forma normale congiuntiva fu descritta da Schröder (1877), mentre quella di Horn fu introdotta molto più tardi da Alfred Horn (1951). La prima esposizione completa della logica proposizionale moderna (e di quella del primo ordine) si trova nel *Begriffschrift* (“Notazione concettuale”) di Gottlob Frege (1879).

Il primo dispositivo meccanico capace di svolgere inferenze logiche fu costruito dal terzo Earl di Stanhope (1753–1816). Il Dimostratore Stanhope poteva gestire i sillogismi e alcune inferenze della teoria della probabilità. William Stanley Jevons, uno dei ricercatori che hanno migliorato ed esteso il lavoro di Boole, costruì il suo “piano logico” nel 1869 per eseguire inferenze in logica booleana. Un racconto divertente e istruttivo di questi e altri antichi congegni meccanici per il ragionamento è fornito da Martin Gardner (1968). Il primo programma per computer pubblicato dedicato all'inferenza logica è stato Logic Theorist di Newell, Shaw e Simon (1957). LT era progettato per modellare i processi di pensiero degli esseri umani. Martin Davis (1957) in effetti aveva progettato già nel 1954 un programma capace di arrivare a una dimostrazione, ma i risultati di Logic Theorist furono pubblicati poco prima. Sia il programma di Davis del 1954 che Logic Theorist erano basati in parte su metodi “ad hoc” che non influenzarono significativamente la ricerca successiva sulla deduzione automatica.

Le tabelle di verità come metodo per verificare la validità e l'insoddisfabilità di formule nel linguaggio del calcolo proposizionale furono introdotte indipendentemente da Ludwig Wittgenstein (1922) ed Emil Post (1921). Negli anni '30 del 1900, si ebbero grandi progressi sui metodi di inferenza per la logica del primo ordine. In particolare Gödel (1930) dimostrò, usando il teorema di Herbrand (Herbrand, 1930), che si poteva ottenere una procedura completa per l'inferenza nella logica del primo ordine mediante una riduzione al calcolo proposizionale. Riprenderemo quest'argomento nel Capitolo 9; il punto importante è ricordare che lo sviluppo di algoritmi proposizionali efficienti negli anni '60 fu motivato in gran parte dall'interesse, da parte dei matematici, per lo sviluppo di un dimostratore di teoremi efficiente per la logica del primo ordine. L'algoritmo Davis-Putnam (Davis e Putnam, 1960) è stato il primo algoritmo efficiente per la risoluzione pro-

posizionale, ma nella gran parte dei casi si rivelò molto meno efficiente del DPLL con backtracking presentato due anni dopo (1962). La regola di risoluzione completa e una dimostrazione della sua completezza fecero la loro comparsa in un fondamentale articolo di J. A. Robinson (1965), che mostrò anche come svolgere ragionamento del primo ordine senza ricorrere a tecniche proposizionali.

Stephen Cook (1971) dimostrò che determinare la soddisfacibilità di una formula nel calcolo proposizionale è un problema NP-completo. Dal momento che determinare l'implicazione è equivalente a determinare l'insoddisfacibilità, il problema è co-NP-completo. Sono noti molti sottoinsiemi della logica proposizionale per cui il problema della soddisfacibilità è risolvibile in tempo polinomiale; le clausole di Horn sono uno di tali sottoinsiemi. L'algoritmo di concatenazione in avanti di complessità lineare per le clausole di Horn è dovuto a Dowling e Gallier (1984), che lo descrivono come un processo di dataflow simile alla propagazione dei segnali in un circuito. La soddisfacibilità è diventata uno degli esempi canonici di riduzione NP; ad esempio Kaye (2000) ha mostrato che il gioco Campo Minato (v. Esercizio 7.11) è NP-completo.

Per tutti gli anni '80 vari autori provarono ad applicare la ricerca locale al problema della soddisfacibilità; gli algoritmi erano tutti basati sull'idea di minimizzare il numero di clausole insoddisfatte (Hansen e Jaumard, 1990). Un algoritmo particolarmente efficace fu sviluppato da Gu (1989) e indipendentemente da Selman et al. (1992), che lo chiamò GSAT e dimostrò che era in grado di risolvere una vasta gamma di problemi molto difficili molto velocemente. L'algoritmo WALKSAT descritto in questo capitolo è stato presentato in Selman et al. (1996).

La "transizione di fase" nella soddisfacibilità dei problemi k -SAT casuali fu osservata per la prima volta da Simon e Dubois (1989). Risultati empirici dovuti a Crawford e Auton (1993) suggeriscono che il punto critico corrisponde a un rapporto clausole/variabili di circa 4,24 per problemi casuali 3-SAT di grandi dimensioni; lo stesso articolo presenta anche un'implementazione molto efficiente di DPLL. Bayardo e Schrag (1997) descrivono un'altra implementazione efficiente di DPLL, che utilizza tecniche prese dal campo del soddisfacimento di vincoli, mentre (Moskewicz et al., 2001) descrivono CHAFF, che può risolvere problemi di verifica hardware con un milione di variabili ed è stato il vincitore della SAT 2002 Competition. Li e Anbulagan (1997) discutono euristiche basate sulla propagazione delle unità che permettono di realizzare risolutori veloci. Cheeseman et al. (1991) presentano dati su una varietà di problemi correlati e ipotizzano che tutti i problemi NP-difficili abbiano una transizione di fase. Kirkpatrick e Selman (1994) discutono vari modi in cui tecniche di fisica statistica potrebbero spiegare la "forma" precisa della transizione di fase. L'analisi teorica della sua *posizione* è tuttora alquanto carente: tutto ciò che si può provare è che giace nell'intervallo [3,003, 4,598] per il 3-SAT casuale. Cook e Mitchell (1997) offrono un eccellente panoramica dei risultati su questo e molti altri argomenti legati alla soddisfacibilità.

Le prime investigazioni teoriche hanno mostrato che DPLL ha una complessità del caso medio polinomiale per certe distribuzioni comuni di problemi. Questa scoperta potenzialmente eccitante divenne molto meno eccitante quando Franco e

Paull (1983) mostraron che gli stessi problemi potevano essere risolti in un tempo costante semplicemente provando assegnamenti casuali. Il metodo di generazione randomizzata che abbiamo descritto in questo capitolo produce problemi molto più difficili. Spinti dal successo empirico della ricerca locale, Koutsoupias e Papadimitriou (1992) hanno mostrato che un semplice algoritmo hill-climbing può risolvere *quasi tutte* le istanze di problemi di soddisfacibilità molto velocemente, suggerendo che i problemi difficili siano rari. Inoltre, Schöning (1999) ha presentato una variante randomizzata di GSAT il cui tempo di esecuzione su problemi 3-SAT è 1,333ⁿ nel caso pessimo: ancora esponenziale, ma molto più veloce degli algoritmi precedenti. I problemi di soddisfacibilità sono ancora un'area di ricerca molto attiva; la collezione di articoli in Du et al. (1999) è un buon punto di partenza.

Gli agenti basati su circuiti di possono far risalire al fondamentale articolo di McCulloch e Pitts (1943) che ha dato origine al campo delle reti neurali. Contrariamente a quanto si crede, l'articolo si occupava dell'implementazione nel cervello di un agente booleano basato su circuiti. Gli agenti basati su circuito, comunque, hanno ricevuto poca attenzione: l'eccezione più notevole è il lavoro di Stan Rosenschein (Rosenschein, 1985; Kaelbling e Rosenschein, 1990) che ha sviluppato tecniche per compilare agenti basati sui circuiti partendo da descrizioni dichiarative dell'ambiente. I circuiti che aggiornano proposizioni memorizzate in registri sono strettamente imparentati all'**assioma di stato successore** sviluppato da Reiter (1991) per la logica del primo ordine. Il lavoro di Rod Brooks (1986, 1989) dimostra l'efficacia della progettazione basata su circuiti per il controllo dei robot: un argomento che riprenderemo nel Capitolo 25 (2° volume). Brooks (1991) sostiene che i progetti basati sui circuiti rappresentano *tutto* quello che serve per l'IA, e che la rappresentazione e il ragionamento sono ingombranti, costosi e superflui. Secondo noi nessuno dei due approcci, preso singolarmente, è sufficiente.

Il mondo del wumpus è stato inventato da Gregory Yob (1975). Ironicamente, Yob lo sviluppò perché si era stufato dei giochi che avevano luogo in una griglia bidimensionale di posizioni: la topologia del mondo originale infatti era un dodecaedro. Noi l'abbiamo riportato nella vecchia, noiosa griglia. Michael Genesereth ha suggerito per primo di usare il mondo del wumpus come ambiente di test per lo sviluppo di agenti.

Esercizi

- 7.1 Descrivete il mondo del wumpus facendo riferimento alle caratteristiche degli ambienti elencate nel Capitolo 2.
- 7.2 Supponiamo che un agente sia giunto fino al punto illustrato nella Figura 7.4(a), non avendo percepito nulla in [1,1], uno spostamento d'aria in [2,1] e la puzza del wumpus in [1,2], e che ora si preoccupi dei contenuti delle stanze [1,3], [2,2] e [3,1]. Ognuna di esse può contenere un pozzo, mentre

una (al massimo) può contenere il wumpus. Seguendo l'esempio della Figura 7.5, costruite l'insieme dei mondi possibili (dovreste trovarne 32). Indicate in quali mondi la KB è vera e in quali sono vere le seguenti formule:

α_2 = "Non c'è alcun pozzo in [2,2]".

α_3 = "C'è un wumpus in [1,3]".

Dimostrate quindi che $KB \models \alpha_2$ e $KB \models \alpha_3$.

- 7.3 Considerate il problema di determinare se una formula del calcolo proposizionale è vera in un dato modello.

a. Scrivete un algoritmo ricorsivo $CP\text{-VERO?}(s, m)$ che restituisce *true* se e solo se la formula s è vera nel modello m (m assegna un valore di verità a ogni simbolo di s). L'algoritmo dovrebbe avere una complessità lineare nelle dimensioni della formula (una versione di questa funzione è già disponibile nel nostro deposito online di codice: la potete usare per svolgere i punti successivi).

b. Fornite tre esempi di formule di cui si può determinare il valore di verità in un modello *parziale* che non specifica il valore di tutti i simboli.

c. Dimostrate che, in generale, il valore di verità (se esiste) di una formula in un modello parziale non può essere determinato in modo efficiente.

d. Modificate il vostro algoritmo $CP\text{-VERO?}$ In modo che possa talvolta determinare il valore di verità partendo da modelli parziali, mantenendo la sua struttura ricorsiva e il tempo di esecuzione lineare. Fornite tre esempi di formule il cui valore *non* viene determinato dal vostro algoritmo nel caso di modelli parziali.

e. Investigate se l'algoritmo modificato rende $TV\text{-IMPLICA?}$ Più efficiente.

- 7.4 Dimostrate le seguenti asserzioni:

a. α è valida se e solo se $True \models \alpha$.

b. Per ogni α , $False \models \alpha$.

c. $\alpha \models \beta$ se e solo se la formula $(\alpha \Rightarrow \beta)$ è valida.

d. $\alpha \equiv \beta$ se e solo se la formula $(\alpha \Leftrightarrow \beta)$ è valida.

e. $\alpha \models \beta$ se e solo se la formula $(\alpha \wedge \neg \beta)$ è insoddisfacibile.

- 7.5 Considerate un vocabolario che contiene solo quattro proposizioni, A , B , C e D . Quanti modelli esistono per le seguenti formule?

a. $(A \wedge B) \vee (B \wedge C)$

b. $A \vee B$

c. $A \Leftrightarrow B \Leftrightarrow C$

- 7.6 Abbiamo definito quattro diversi connettivi logici binari.

a. Ne esistono altri che potrebbero essere utili?

b. Quanti possibili connettivi binari distinti possono esistere?

c. Perché qualcuno di essi non è molto utile?

- 7.7 Usando un metodo a vostra scelta, verificate tutte le equivalenze della Figura 7.11.
- 7.8 Decidete se le ognuna delle seguenti formule è valida, insoddisfacibile o nessuna delle due. Verificate le vostre decisioni usando le tabelle di verità o le regole di equivalenza della Figura 7.11.
- $Fumo \Rightarrow Fumo$
 - $Fumo \Rightarrow Fuoco$
 - $(Fumo \Rightarrow Fuoco) \Rightarrow (\neg Fumo \Rightarrow \neg Fuoco)$
 - $Fumo \vee Fuoco \vee \neg Fuoco$
 - $((Fumo \wedge Calore) \Rightarrow Fuoco) \Leftrightarrow ((Fumo \Rightarrow Fuoco) \vee (Calore \Rightarrow Fuoco))$
 - $(Fumo \Rightarrow Fuoco) \Rightarrow ((Fumo \wedge Calore) \Rightarrow Fuoco)$
 - $Grosso \vee Stupido \vee (Grosso \Rightarrow Stupido)$
 - $(Grosso \wedge Stupido) \vee \neg Stupido$
- 7.9 (Adattato da Barwise e Etchemendy (1993)). Data il seguente testo, potete dimostrare che l'unicorno è un animale mitico? È magico? E potete dire se ha un corno?
- Se l'unicorno è un animale mitico, è immortale, ma se non è mitico, allora è un mammifero mortale. Se è immortale o un mammifero, allora ha un corno.
L'unicorno è magico se ha un corno.
- 7.10 Ogni formula del calcolo proposizionale è logicamente equivalente all'asserzione che ogni mondo possibile in cui la formula sarebbe falsa non è verificato. Partendo da quest'osservazione, dimostrate che ogni formula può essere scritta in CNF.
- 7.11 Campo Minato, il ben noto gioco per computer, è strettamente imparentato con il mondo del wumpus. Un mondo “minato” è costituito da una griglia rettangolare di N caselle che contengono M mine invisibili distribuite casualmente. L'agente può verificare la presenza di mine in qualsiasi casella; se però quest'ultima contiene una mina il risultato sarà la morte istantanea. L'ambiente indica la presenza di mine rivelando, per ogni casella esaminata, il numero di mine ortogonalmente o diagonalmente adiacenti. Lo scopo del gioco è marcare tutte le caselle che non contengono mine.
- Sia $X_{i,j}$ vero se e solo se la casella $[i, j]$ contiene una mina. Scrivete la formula che asserisce che esistono esattamente due mine adiacenti a $[1,1]$ utilizzando una combinazione logica di proposizioni $X_{i,j}$.
 - Generalizzate la formula del punto (a) spiegando come costruire una formula CNF che asserisce che k delle n caselle vicine a una casella data contengono mine.
 - Spiegate precisamente in che modo un agente potrebbe usare DPLL per dimostrare che una data casella contiene (o non contiene) una mina, ignorando il vincolo globale che il numero totale delle mine sia M .

- d. Supponiamo di aver costruito il vincolo globale con il metodo indicato al punto (b). In che modo il numero delle clausole dipende da M e N ? Suggerite un modo di modificare DPLL in modo che il vincolo globale non debba essere rappresentato esplicitamente.
- e. Ci sono delle conclusioni derivate dal metodo del punto (c) che risultano invalidate quando si prende in considerazione il vincolo globale?
- f. Fornite esempi di configurazioni di valori che producono *dipendenze a lungo raggio* tali per cui il contenuto di una casella non ancora esaminata fornirebbe informazioni sulla presenza di una mina in una casella molto distante [suggerimento: considerate un griglia di caselle $N \times 1$].

7.12 Questo esercizio esamina le relazioni tra le clausole e le formule di implicazione.

- a. Dimostrate che la clausola $(\neg P_1 \vee \dots \vee \neg P_m \vee Q)$ è logicamente equivalente alla formula di implicazione $(P_1 \wedge \dots \wedge P_m) \Rightarrow Q$.
- b. Dimostrate che ogni clausola (indipendentemente dal numero di letterali positivi) può essere scritta nella forma $(P_1 \wedge \dots \wedge P_m) \Rightarrow (Q_1 \vee \dots \vee Q_n)$, dove le P e le Q sono simboli proposizionali. Una base di conoscenza composta esclusivamente da formule come questa si dice **in forma normale implicativa** o **in forma di Kowalski**.
- c. Scrivete la regola di risoluzione completa per le formule in forma normale implicativa.

7.13 In questo esercizio dovrete sviluppare ulteriormente l'agente per il mondo del wumpus basato su circuiti.

- a. Scrivete un'equazione simile alla (7.4) per la proposizione *Freccia*, che deve risultare vera se l'agente possiede ancora una freccia. Disegnate il circuito corrispondente.
- b. Fate lo stesso per *RivoltatoADestra*, usando come modello l'Equazione (7.5).
- c. Create versioni delle Equazioni 7.7 e 7.8 per localizzare il wumpus, e disegnatene il circuito.

7.14 Discutete che cosa si intende per comportamento *ottimo* nel mondo del wumpus. Mostrate che la nostra definizione di CP-AGENTE-WUMPUS non è ottima, e suggerite in che modo si potrebbe migliorarla.

7.15 Estendete CP-AGENTE-WUMPUS in modo che tenga traccia di tutti i fatti rilevanti *all'interno* della base di conoscenza.

7.16 Quanto tempo ci vuole per dimostrare che $KB \models \alpha$ con DPLL quando α è un letterale già contenuto in KB ? Spiegate.

7.17 Seguite passo passo il comportamento di DPLL sulla base di conoscenza della Figura 7.15 mentre cerca di dimostrare Q , e confrontatelo con l'algoritmo di concatenazione in avanti.



Capitolo 8

Logica del primo ordine

In cui notiamo che al mondo esiste una grande varietà di oggetti, alcuni dei quali hanno relazioni con altri oggetti, e ci sforziamo di ragionare su di essi.

Nel Capitolo 7 abbiamo visto come un agente basato sulla conoscenza potrebbe rappresentare il mondo in cui opera e dedurre le azioni che deve intraprendere. Come linguaggio di rappresentazione abbiamo usato il calcolo proposizionale, sufficiente per presentare i concetti base della logica e degli agenti basati sulla conoscenza. Sfortunatamente, il calcolo proposizionale è un linguaggio troppo poco potente per rappresentare la conoscenza di ambienti complessi in modo compatto. In questo capitolo prenderemo in esame la logica del primo ordine,¹ che è sufficientemente espressiva per rappresentare buona parte della nostra conoscenza comune. Inoltre questa logica è equivalente o forma la base di molti altri linguaggi di rappresentazione, ed è stata studiata intensivamente per molti decenni. Cominceremo nel Paragrafo 8.1 con una discussione dei linguaggi di rappresentazione in generale; il Paragrafo 8.2 presenta la sintassi e la semantica della logica del primo ordine; i Paragrafi 8.3 e 8.4 ne illustrano l'uso applicato a semplici rappresentazioni.

logica del primo ordine

¹ Chiamata anche calcolo dei predicati di primo ordine, e talvolta indicata con l'acronimo inglese FOL o FOPC.

8.1 Ancora sulla rappresentazione

In questo paragrafo tratteremo la natura dei linguaggi di rappresentazione. La discussione ci spingerà a sviluppare la logica del primo ordine, molto più espressiva del calcolo proposizionale introdotto nel capitolo precedente. Esamineremo lo stesso calcolo proposizionale e altri linguaggi per capire i loro punti forti e le manchevolezze: il discorso sarà necessariamente abbozzato, dato che dovremo concentrare in pochi paragrafi interi secoli di pensiero, tentativi ed errori.

I linguaggi di programmazione (come C++ o Java o Lisp) sono la categoria di linguaggi formali di uso più comune. I programmi stessi, in senso proprio, rappresentano solo processi computazionali. Le strutture dati al loro interno possono rappresentare fatti; ad esempio, un programma potrebbe usare una matrice 4×4 per memorizzare il contenuto di un mondo del wumpus. Di conseguenza, un'istruzione nel linguaggio di programmazione come *Mondo[2,2] ← Pozzo* è un modo abbastanza naturale di asserire che c'è un pozzo nella posizione [2,2]. Rappresentazioni come questa sono, come si dice, *ad hoc*; i database sono stati sviluppati precisamente per fornire un modo più generale e indipendente dal dominio di memorizzare e gestire i fatti. Ciò che normalmente manca totalmente ai linguaggi di programmazione è un meccanismo generale che consenta di derivare fatti da altri fatti; ogni aggiornamento a una struttura dati viene effettuato da una procedura specifica del dominio i cui dettagli sono derivati, da parte del programmatore, proprio dalla conoscenza del dominio stesso. Questo approccio **procedurale** si pone in contrasto con la natura **dichiarativa** del calcolo proposizionale, che separa nettamente la conoscenza dall'inferenza, e in cui quest'ultima rimane totalmente indipendente dal dominio.

Una seconda lacuna delle strutture dati definite dai programmi (e dai database, peraltro) è l'impossibilità di esprimere facilmente concetti come, ad esempio, "c'è un pozzo in una delle posizioni [2,2] o [3,1]" oppure "se il wumpus è in [1,1] allora non è in [2,2]". I programmi possono memorizzare un solo valore per ogni variabile, e alcuni sistemi permettono anche l'uso del valore "indefinito", ma non possiedono affatto la potenza espressiva necessaria a gestire informazione parziale.

Il calcolo proposizionale è un linguaggio dichiarativo perché la sua semantica è basata su una relazione di verità che collega le formule e i mondi possibili. Inoltre l'uso della disgiunzione e della negazione lo rendono sufficientemente espressivo da gestire informazione parziale. Il calcolo proposizionale ha poi una terza caratteristica desiderabile nei linguaggi di rappresentazione, la **composizionalità**. In un linguaggio composito, il significato di una formula è una funzione del significato delle sue parti. Ad esempio, " $S_{1,4} \wedge S_{1,2}$ " è correlato al significato di " $S_{1,4}$ " e di " $S_{1,2}$ ". Sarebbe ben strano se " $S_{1,4}$ " significasse che si sente puzzava nella stanza [1,4] e " $S_{1,2}$ " fosse vero se si sente puzzava nella stanza [1,2], ma contemporaneamente " $S_{1,4} \wedge S_{1,2}$ " indicasse che la Francia e la Polonia hanno pareggiato 1-1 nel-

la partita di qualificazione di hockey su ghiaccio della settimana scorsa. È chiaro che la mancanza di composizionalità rende il lavoro di un sistema basato su ragionamento molto più difficile.

Come abbiamo visto nel Capitolo 7, la logica proposizionale non ha la potenza espressiva per descrivere un ambiente con molti oggetti *in modo conciso*: per esempio, per gestire la presenza di spostamenti d'aria e pozzi siamo stati costretti a scrivere per ogni stanza una regola separata come

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

D'altra parte, in linguaggio naturale sembra abbastanza facile dire, una volta per tutte, “nelle stanze adiacenti a quelle che contengono un pozzo si percepisce uno spostamento d'aria”. La sintassi e la semantica dei linguaggi usati dagli esseri umani, in qualche modo, rendono possibile descrivere l'ambiente in modo conciso.

Se ci fermiamo a pensare un attimo, dobbiamo dire che i linguaggi naturali come l'italiano o l'inglese sono effettivamente molto espressivi. Siamo riusciti a scrivere quasi tutto questo libro in un linguaggio naturale, ricorrendo solo occasionalmente ad altri linguaggi come la logica, la matematica o il linguaggio grafico dei diagrammi. Una lunga tradizione di studi nei campi della linguistica e della filosofia considera quello naturale essenzialmente un linguaggio di rappresentazione della conoscenza di tipo dichiarativo, la cui semantica si è cercato a lungo di definire formalmente. Un simile programma di ricerca, qualora dovesse avere successo, sarebbe di grande utilità all'intelligenza artificiale perché permetterebbe di usare un linguaggio naturale (o un suo derivato) all'interno dei sistemi di rappresentazione e ragionamento.

Nella visione moderna il linguaggio naturale ha uno scopo leggermente diverso, servendo più come mezzo di comunicazione che di pura rappresentazione. Quando una persona punta il dito e dice, “Guarda!”, l'ascoltatore viene a sapere che, poniamo, Superman si staglia nel cielo contro il profilo dei grattacieli. Eppure non diremmo che questo fatto è intrinsecamente codificato nella frase “Guarda!”. Il significato della frase dipende dal suo contenuto ma anche dal contesto in cui è stata pronunciata: chiaramente nessuno penserebbe di inserire una frase come “Guarda!” in una base di conoscenza e di essere in grado di recuperarne il significato senza memorizzare anche una rappresentazione del contesto; considerazione che solleva la questione di come sia possibile rappresentare il contesto stesso. Inoltre, i linguaggi naturali non sono compostionali: il significato di una frase come “a quel punto ella lo vide” può dipendere da un contesto costruito da molte frasi precedenti e successive. Infine, i linguaggi naturali hanno anche il difetto di contenere ambiguità, che causa molte difficoltà al ragionamento. Come dice Pinker (1995): “Quando le persone pensano alla parola *spring*, certamente non devono chiedersi se stanno pensando a una stagione (*spring* come “primavera”) o a un oggetto che fa *boing* (*spring* come “molla”); e se a una parola possono corrispondere due pensieri, i pensieri non possono essere parole”.

oggetti
relazioni
funzioni

proprietà

Il nostro approccio sarà quello di partire da un aspetto fondamentale della logica proposizionale – una semantica dichiarativa e compositiva, indipendente dal contesto e non ambigua – e su tali fondamenta costruire una logica più espressiva, prendendo in prestito metodi di rappresentazione dal linguaggio naturale ed evitando le sue limitazioni. Quando prendiamo in esame la sintassi di un linguaggio naturale, gli elementi che si distinguono più facilmente sono i sostantivi e i sintagmi nominali che si riferiscono a **oggetti** (stanze, pozzi, wumpus) e i verbi e i sintagmi verbali che si riferiscono a **relazioni** tra gli oggetti (è ventosa, è adiacente a, scocca). Alcune relazioni, dette **funzioni**, hanno un solo “valore” per ogni “input”. È facile elencare alcuni esempi di oggetti, relazioni e funzioni.

- ◆ Oggetti: persone, cavalli, numeri, teorie, Ronald McDonald, colori, partite a baseball, guerre, secoli . . .
- ◆ Relazioni: possono essere unarie, nel qual caso prendono il nome di **proprietà** (come quella di essere rossi, tondi, falsi, primi, alti). Le più generali sono *n*-arie e comprendono relazioni come fratello di, più grande di, all'interno di, parte di, avvenuto dopo, di colore, possiede, sta in mezzo a . . .
- ◆ Funzioni: padre di, miglior amico, terzo inning di, uno più di, all'inizio di . . .

In effetti, quasi tutte le asserzioni possono essere ricondotte agli oggetti, le proprietà e le relazioni che le compongono.

- ◆ “Uno più due fa tre”.
- ◆ Oggetti: uno, due, tre, uno più due; Relazione: fa; Funzione: più (“uno più due” è il nome dell'oggetto che si ottiene applicando la funzione “più” agli oggetti “uno” e “due”. Tre è un altro nome dello stesso oggetto).
- ◆ “Le stanze adiacenti a quella che contiene un wumpus sono puzzolenti”.
- ◆ Oggetti: wumpus, stanze; Proprietà: puzzolente; Relazione: adiacenti.
- ◆ “Il malvagio Re Giovanni governò l'Inghilterra nel 1200”.
Oggetti: Giovanni, Inghilterra, 1200; Relazione: governò; Proprietà: malvagio, re.

Il linguaggio della **logica del primo ordine**, di cui definiremo sintassi e semantica nel prossimo paragrafo, è costruito intorno agli oggetti e alle relazioni. La sua importanza nei campi della matematica, della filosofia e dell'intelligenza artificiale deriva proprio dal fatto che tali discipline (e, in effetti, gran parte dell'esistenza umana di tutti i giorni) possono essere proficuamente pensate come attività che hanno a che fare con oggetti e con le relazioni che li collegano. La logica del primo ordine può anche esprimere fatti che riguardano alcuni o tutti gli oggetti dell'universo. Questo rende possibile esprimere leggi generali o regole, come “le stanze adiacenti a un wumpus puzzano”.

La differenza fondamentale tra la logica proposizionale e quella del primo ordine sta nell'**impegno ontologico** assunto da ognuno dei due linguaggi, ovvero le sue ipotesi circa la natura della realtà.

La logica proposizionale, ad esempio, dà per scontato che i fatti nel mondo sono veri oppure no. Ogni fatto può essere in uno di due possibili stati: *true* o *false*.² La logica del primo ordine accoglie ulteriori ipotesi, e in particolare che il mondo consista di oggetti legati da relazioni che possono o meno essere verificate. Le logiche specializzate si impegnano ontologicamente ancora di più: la **logica temporale**, ad esempio, parte dal presupposto che i fatti siano verificati in un determinato *momento* e che i tempi (che possono essere istanti o intervalli) siano tutti ordinati. In questo modo le logiche specializzate possono conferire a certe categorie di oggetti (e agli assiomi che li riguardano) uno stato “privilegiato”, invece di definirli semplicemente nella base di conoscenza. La **logica di ordine superiore** considera oggetti le stesse relazioni e funzioni a cui la logica di primo ordine fa riferimento. In questo modo è possibile formulare asserzioni che riguardano *tutte* le relazioni, definendo ad esempio che cosa si intende per transitività di una relazione. A differenza della maggior parte delle logiche specializzate, quella di ordine superiore è più espressiva della logica di primo ordine in senso stretto, il che significa che alcune sue formule non possono essere espresse con un numero finito di formule di logica del primo ordine.

Una logica può anche essere caratterizzata dai suoi **impegni epistemologici**, ovvero dai diversi stati di conoscenza che permette nei confronti di ciascun fatto. Sia nella logica proposizionale che in quella del primo ordine, una formula rappresenta un fatto che l’agente può ritener vero, oppure falso; inoltre può anche non avere un’opinione. In queste logiche, quindi, a ogni formula può essere associato uno di tre possibili stati di conoscenza. I sistemi che usano la **teoria della probabilità** d’altra parte permettono qualsiasi *grado di credenza*, da 0 (sicura falsità) a 1 (totale certezza).³ Un agente del mondo del wumpus probabilistico, ad esempio, potrebbe credere che il mostro si trovi nella posizione [1,3] con probabilità 0,75. Gli impegni ontologici ed epistemologici di cinque logiche diverse sono riassunti nella Figura 8.1.

Nel prossimo paragrafo ci tufferemo nei dettagli della logica del primo ordine. Come a uno studente di fisica è richiesta una certa familiarità con la matematica, così chi studia l’IA deve sviluppare una certa abilità nel lavorare con la notazione logica. In ogni caso, è importante *non* farsi prendere troppo la mano dai

impegno ontologico

logica temporale

logica di ordine superiore

impegni epistemologici

² Al contrario, i fatti nella logica sfumata o fuzzy hanno un grado di verità che va da 0 a 1. La frase “Vienna è una grande città”, ad esempio, nel nostro mondo potrebbe essere vera solo per un grado 0,6.

³ È importante non confondere il grado di credenza della teoria della probabilità con quello di verità della logica fuzzy. In effetti, alcuni sistemi fuzzy permettono di esprimere incertezza (grado di credenza) riguardo ai gradi di verità.

IL LINGUAGGIO DEL PENSIERO

I filosofi e gli psicologi hanno riflettuto a lungo su come gli esseri umani e gli altri animali rappresentino la conoscenza. È chiaro che l'evoluzione del linguaggio naturale ha giocato un ruolo importante nello sviluppo di questa capacità negli umani. D'altra parte, molti studi di psicologia sembrano dimostrare che le persone non usano direttamente il linguaggio nelle loro rappresentazioni interne. Ad esempio, con quale delle seguenti due frasi si apre il Paragrafo 8.1?

"In questo paragrafo tratteremo la natura dei linguaggi di rappresentazione..."

"Questo paragrafo presenta i linguaggi di rappresentazione della conoscenza..."

Wanner (1974) ha scoperto che, in un test come questo, le persone forniscono la risposta giusta circa il 50% delle volte, con una frequenza cioè simile a quella di chi tiri completamente a indovinare. Tuttavia, le stesse persone ricordano il contenuto di ciò che hanno letto con un'accuratezza superiore al 90%. Questo suggerisce che gli esseri umani processano internamente le parole trasformandole in una forma di rappresentazione non verbale, che è ciò che chiamiamo **memoria**. L'esatto meccanismo con cui il linguaggio permette e controlla la rappresentazione delle idee negli esseri umani rimane una questione affascinante. La famosa **ipotesi di Sapir-Whorf** sostiene che il linguaggio che parliamo influenza profondamente il modo in cui pensiamo e prendiamo decisioni, in particolar modo determinando la struttura delle categorie che usiamo per dividere il mondo in diversi tipi di oggetti. Whorf (1956) asserì che gli eschimesi hanno diverse parole per riferirsi alla neve, e per questo ne hanno un'esperienza diversa rispetto a chi parla un altro linguaggio. Alcuni linguisti hanno contestato la base sperimentale di quest'asserzione: Pullum (1991) ha sostenuto che l'Inuit, lo Yupik e altri linguaggi imparentati con essi hanno più o meno lo stesso numero di parole dell'inglese per riferirsi a concetti legati alla neve. Altri ricercatori hanno supportato l'ipotesi (Fortescue, 1984). Sembra comunque vero, al di là di ogni discussione, il fatto che le popolazioni che hanno molta familiarità con qualche aspetto del mondo sviluppino vocabolari molto più dettagliati: gli entomologi, ad esempio, suddividono quelli che la maggior parte di noi chiamerebbero genericamente *scarabei* in centinaia di migliaia di specie, e hanno familiarità personale con molte di esse (il biologo J. B. S. Haldane una volta si lamentò della "smodata passione per i Coleotteri" da parte del Creatore). Inoltre, gli sciatori esperti sono soliti riferirsi alla neve con una varietà di termini disparati come polvere, zuppa, purè, gromma, grano, cemento, crosta, zucchero, asfalto, flanella, bambagia, pastone e così via; un livello di distinzione del tutto ignoto alle persone normali. Quello che non è chiaro è la direzione della causalità: gli sciatori si accorgono delle distinzioni solo imparando le parole, oppure le sperimentano personalmente e associano le loro sensazioni alle etichette usate dalla comunità? Questa distinzione è particolarmente importante nello studio dell'apprendimento dei bambini. Ancora oggi non abbiamo compreso in quale misura siano intrecciati l'apprendimento del linguaggio e quello del pensiero. Per esempio, la conoscenza del nome usato per riferirsi a un concetto come *scapolo* rende più facile formulare e ragionare su concetti più complessi che includono quel nome, come *scapolo appetibile*?

dettagli *specifici* della notazione logica: dopotutto, ne esistono dozzine. Le cose più importanti sono il modo in cui il linguaggio facilita la definizione di rappresentazioni concise e come la sua semantica rende possibile l'applicazione di procedure di ragionamento corrette.

linguaggio	impegno ontologico (ciò che esiste nel mondo)	impegno epistemologico (le credenze di un agente circa un fatto)
logica proposizionale	fatti	vero/falso/sconosciuto
logica del primo ordine	fatti, oggetti, relazioni	vero/falso/sconosciuto
logica temporale	fatti, oggetti, relazioni, tempi	vero/falso/sconosciuto
teoria della probabilità	fatti	grado di credenza $\in [0, 1]$
logica fuzzy	fatti con gradi di verità $\in [0, 1]$	valore interno conosciuto

Figura 8.1 Cinque linguaggi formali con i rispettivi impegni ontologici ed epistemologici.

8.2 Sintassi e semantica della logica del primo ordine

Cominceremo specificando in modo più preciso il modo in cui i mondi possibili della logica del primo ordine riflettono l'impegno ontologico nei confronti di oggetti e relazioni. Introdurremo quindi i vari elementi del linguaggio, spiegando man mano la loro semantica.

Modelli per la logica del primo ordine

Ricorderete dal Capitolo 7 che i modelli di un linguaggio logico sono le strutture formali che costituiscono i mondi possibili che possono essere presi in considerazione. Nel calcolo proposizionale, i modelli sono semplici insiemi di valori di verità assegnati ai simboli. I modelli della logica del primo ordine sono più interessanti: per prima cosa, contengono oggetti! Il dominio di un modello è l'insieme degli oggetti che contiene, chiamati appunto elementi del dominio. La Figura 8.2 mostra un modello con cinque oggetti: Riccardo Cuor di Leone, Re d'Inghilterra dal 1189 al 1199; suo fratello minore, il malvagio Re Giovanni, che regnò dal 1199 al 1215; le gambe sinistre di Riccardo e Giovanni; e una corona.

Gli oggetti del modello possono essere messi in relazione in molti modi. Nella figura, Riccardo e Giovanni sono fratelli. Formalmente, una relazione non è altro che un insieme di tuple di oggetti collegati (una tupla è una collezione ordinata di oggetti, e si scrive racchiusa tra parentesi angolari). Così, la relazione di "fratellanza" nel modello è costituita dall'insieme

$$\{ \langle \text{Riccardo Cuor di Leone}, \text{Re Giovanni} \rangle, \\ \langle \text{Re Giovanni}, \text{Riccardo Cuor di Leone} \rangle \}. \quad (8.1)$$

dominio
elementi del dominio

tuple

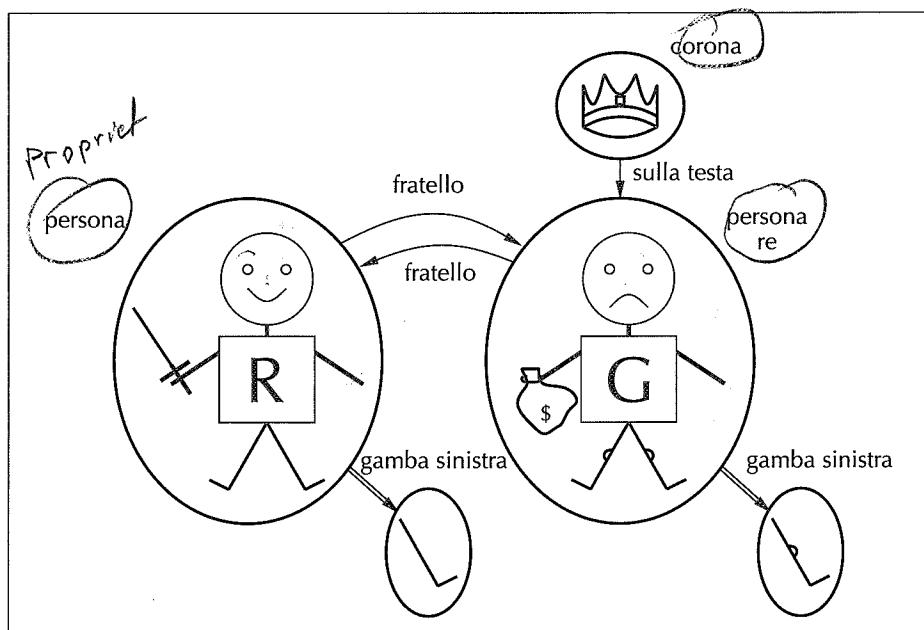


Figura 8.2 Un modello che contiene cinque oggetti, due relazioni binarie, tre relazioni unarie (indicate mediante etichette sugli oggetti) e una funzione unaria, la gamba sinistra.

Notate che qui sopra abbiamo indicato i nomi per esteso ma, se preferite, potete sostituire mentalmente le immagini della figura al loro posto. La corona è sulla testa di Giovanni, per cui la relazione “sulla testa” contiene solo una tupla, \langle la corona, Re Giovanni \rangle . La relazione “fratello” e quella “sulla testa” sono binarie; questo significa che collegano coppie di oggetti. Il modello contiene anche relazioni unarie, o proprietà: “persona” è vera sia per Riccardo che per Giovanni; la proprietà “re” vale solo per Giovanni (presumibilmente perché, a questo punto, Riccardo è già morto); infine, la proprietà “corona” è vera solo per la corona.

Alcuni tipi di relazioni si possono considerare meglio come funzioni, in quanto ogni dato oggetto può essere collegato attraverso di esse a esattamente un altro oggetto. Per esempio, ogni persona ha una gamba sinistra, ragion per cui il modello ha una funzione unaria “gamba sinistra” che include le seguenti corrispondenze:

$$\begin{aligned} \langle \text{Riccardo Cuor di Leone} \rangle &\rightarrow \text{la gamba sinistra di Riccardo} \\ \langle \text{Re Giovanni} \rangle &\rightarrow \text{la gamba sinistra di Giovanni}. \end{aligned} \quad (8.2)$$

Se vogliamo essere formali, dobbiamo dire che i modelli nella logica del primo ordine richiedono funzioni totali, ovvero che ci dev’essere un valore per ogni tupla di input. Questo significa che anche la corona deve avere una gamba sinistra, è anche tutte le gambe devono a loro volta avere una gamba sinistra. Per risolvere que-

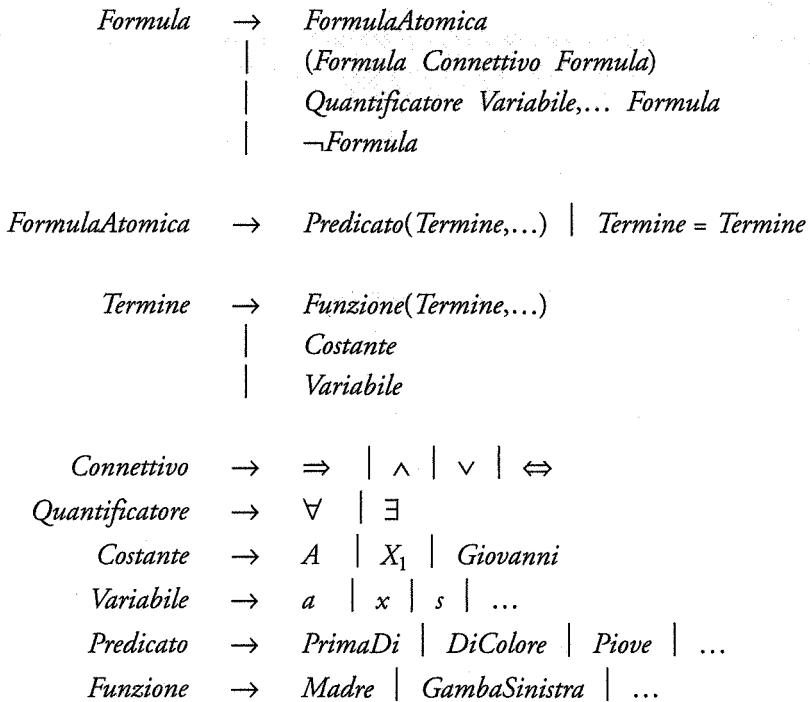


Figura 8.3 La sintassi della logica del primo ordine con uguaglianza, espressa nella forma di Backus–Naur (v. pag. 631 se non siete familiari con questa notazione). La sintassi è precisa per quanto riguarda le parentesi; i commenti di pag. 264 riguardanti le parentesi e la precedenza degli operatori si applicano allo stesso modo alla logica del primo ordine.

sto problema si può usare un artificio che prevede l'aggiunta di un oggetto “invisibile” che fa le veci di gamba sinistra di tutti gli oggetti che non hanno effettivamente una gamba sinistra, incluso l'oggetto stesso. Fortunatamente, finché nessuno esprime formule che riguardano le gambe sinistre degli oggetti che non hanno gambe, questi particolari tecnici sono superflui.

Simboli e interpretazioni

Passiamo ora a considerare la sintassi del linguaggio. Il lettore impaziente può leggere una descrizione completa dalla grammatica formale della logica del primo ordine, nella Figura 8.3.

Gli elementi sintattici base sono i simboli utilizzati per indicare gli oggetti, le relazioni e le funzioni. Di conseguenza ce ne sono tre tipi: simboli di costante, che rappresentano oggetti; simboli di predicato, che rappresentano relazioni; simboli di funzione, che rappresentano appunto funzioni. Adotteremo la convenzione che

simboli di costante
simboli di predicato
simboli di funzione

arità

interpretazione

interpretazione intesa

i simboli cominciano sempre con lettere maiuscole: useremo quindi simboli di costante come *Riccardo* e *Giovanni*; simboli di predicato come *Fratello*, *SullaTesta*, *Persona*, *Re* e *Corona*; e il simbolo di funzione *GambaSinistra*. Come nel caso dei simboli proposizionali, la scelta dei nomi è totalmente libera. Ogni simbolo di predicato e di funzione ha una specifica arità che indica il numero di parametri.

Per determinare la verità delle formule, la semantica deve metterle in relazione con i modelli. Per far questo abbiamo bisogno di un'interpretazione che specifichi esattamente a quali oggetti, relazioni e funzioni fanno riferimento i simboli di costante, predicato e funzione. Un'intepretazione possibile per il nostro modello – che chiameremo interpretazione intesa – è la seguente:

- ◆ *Riccardo* si riferisce a Riccardo Cuor di Leone e *Giovanni* al malvagio Re Giovanni;
- ◆ *Fratello* indica la relazione di fratellanza, ovvero l'insieme di tuple di oggetti fornite dall'Equazione (8.1); *SullaTesta* la relazione di "stare sulla testa" che si verifica tra la corona e Re Giovanni; *Persona*, *Re* e *Corona* fanno riferimento agli insiemi di oggetti che sono effettivamente persone, re e corone;
- ◆ *GambaSinistra* indica la funzione "gamba sinistra di", cioè la corrispondenza specificata dall'Equazione (8.2).

È possibile formulare molte altre interpretazioni per collegare questi simboli allo stesso modello. Ad esempio, potremmo chiamare *Riccardo* la corona e *Giovanni* la gamba sinistra di Re Giovanni. Dato che nel modello ci sono cinque oggetti, solo per i simboli di costante *Riccardo* e *Giovanni* le interpretazioni possibili sono 25. Notate che non è obbligatorio che tutti gli oggetti abbiano un nome; la nostra interpretazione intesa ad esempio non specifica quelli delle gambe e della corona. È anche possibile che un oggetto abbia più nomi; in una certa interpretazione sia *Riccardo* che *Giovanni* potrebbero riferirsi alla corona. Se vi sembra che questo possa portare a confusione, ricordate che anche nella logica proposizionale è perfettamente possibile che ci siano modelli in cui i simboli *Nuvoloso* e *Soleggiato* sono entrambi veri; è compito della base di conoscenza escludere i modelli inconsistenti.

Il valore di verità di ogni formula è determinato da un modello e un'interpretazione dei simboli che la compongono. Ne consegue che implicazione, validità etc. sono definiti in termini di tutti i possibili modelli e tutte le possibili interpretazioni. È importante notare che il numero di elementi del dominio in ogni modello potrebbe essere illimitato, come ad esempio nel caso degli interi o dei numeri reali. Anche il numero dei modelli possibili risulta così illimitato, così come quello delle interpretazioni. Verificare l'implicazione attraverso l'enumerazione di tutti i modelli, cosa possibile in logica proposizionale, in quella del primo ordine è fuori discussione. Anche se il numero di oggetti è molto piccolo, infatti, quello delle combinazioni cresce molto rapidamente: solo con i simboli del nostro esempio e un dominio di cinque oggetti ci sono circa 10^{25} combinazioni (v. Esercizio 8.5).

Termini

Un termine è un'espressione logica che si riferisce a un oggetto. I simboli di costante sono quindi termini, ma non è sempre il caso di assegnare un simbolo distinto a ogni oggetto: per esempio, nel linguaggio di tutti i giorni useremo l'espressione "la gamba sinistra di Re Giovanni" e non le daremo certo un nome specifico. Questo è proprio lo scopo dei simboli di funzione: invece di usare un simbolo di costante, potremo scrivere *GambaSinistra(Giovanni)*. Nel caso generale, un termine complesso è formato da un simbolo di funzione seguito da una lista tra parentesi dei suoi argomenti, costituiti a loro volta da termini. È importante tener sempre presente che un termine complesso è solo un modo complicato di dare il nome a un oggetto: non è affatto una "chiamata di subroutine" che "restituisce un valore". Non c'è una procedura *GambaSinistra* che prende come input una persona e restituisce una gamba. Possiamo ragionare sulle gambe sinistre (ad esempio affermando la regola generale che tutte le persone ne hanno una, e quindi deducendo che anche Giovanni deve averla) senza mai neppure fornire una definizione di *GambaSinistra*. Questa è una cosa decisamente impossibile da fare con le procedure dei linguaggi di programmazione.⁴

termine

La semantica formale dei termini si spiega senza difficoltà. Consideriamo un termine $f(t_1, \dots, t_n)$. Il simbolo di funzione f si riferisce a una qualche funzione del modello (che chiameremo F); i termini usati come argomento fanno riferimento a oggetti del dominio (che indicheremo con d_1, \dots, d_n); nella sua interezza il termine indica quindi l'oggetto che corrisponde al valore della funzione F applicata a d_1, \dots, d_n . Per esempio, supponiamo che il simbolo di funzione *GambaSinistra* si riferisca alla funzione mostrata nell'Equazione (8.2) e che *Giovanni* sia Re Giovanni: allora *GambaSinistra(Giovanni)* sarà la gamba sinistra di Re Giovanni. In questo modo, l'interpretazione fissa l'oggetto corrispondente a ogni termine.

Formule atomiche

Ora che possiamo riferirci agli oggetti con i termini e alle relazioni con i simboli di predicato, possiamo mettere insieme le due cose per dare origine a formule atomiche capaci di asserire fatti. Una formula atomica è composta da un simbolo di predicato seguito da una lista di termini tra parentesi:

Fratello(Riccardo, Giovanni).

⁴ Le espressioni λ forniscono una notazione utile per costruire nuovi simboli di funzione "al momento". La funzione che restituisce il quadrato del suo parametro, ad esempio, può essere scritta come $(\lambda x \ x \times x)$ e può essere applicata agli argomenti proprio come qualsiasi altro simbolo di funzione. Un'espressione λ può anche essere definita e utilizzata come simbolo di predicato (v. Capitolo 22, nel 2° vol.). L'operatore *Lambda* del Lisp funziona esattamente in questo modo. Notate che l'uso di λ in questo modo non aumenta il potere espressivo della logica del primo ordine, dal momento che ogni formula che include un'espressione λ può essere sostituita da una formula equivalente che non ne fa uso.

La formula afferma, in base all'interpretazione intesa che abbiamo fornito, che Riccardo Cuor di leone è un fratello di Re Giovanni.⁵ Le formule atomiche possono avere termini complessi come argomenti. Così,

Sposato(Padre(Riccardo), Madre(Giovanni))

afferma (ancora una volta, secondo un'interpretazione adeguata) che il padre di Riccardo Cuor di Leone è sposato con la madre di Re Giovanni.

Una formula atomica è vera in un dato modello, sotto una determinata interpretazione, se la relazione a cui fa riferimento il simbolo di predicato è verificata tra gli oggetti a cui fanno riferimento gli argomenti.

Formule complesse

Proprio come nel calcolo proposizionale, possiamo usare i connettivi logici per costruire formule complesse. La semantica delle formule costruite con i connettivi logici è identica al caso proposizionale. Ecco quattro formule che, secondo la nostra interpretazione intesa, sono vere nel modello della Figura 8.2:

Fratello(GambaSinistra(Riccardo), Giovanni)

Fratello(Riccardo, Giovanni) \wedge Fratello(Giovanni, Riccardo)

Re(Riccardo) \vee Re(Giovanni)

\neg Re(Riccardo) \Rightarrow Re(Giovanni).

Quantificatori

Avendo a disposizione una logica basata sugli oggetti, è naturale che si desideri esprimere caratteristiche di intere collezioni di oggetti senza doverli enumerare uno per uno. I quantificatori ci permettono di farlo. La logica del primo ordine comprende due quantificatori standard, quello universale e quello esistenziale.

Quantificazione universale (\forall)

Come ricorderete, nel Capitolo 7 abbiamo avuto molte difficoltà nell'esprimere leggi generali per mezzo del calcolo proposizionale. Regole come "le stanze adiacenti a quella che contiene il wumpus sono puzzolenti" e "tutti i re sono persone" sono, al contrario, molto comuni nella logica del primo ordine. Ci occuperemo della prima nel Paragrafo 8.3; la seconda si scrive

$$\forall x \text{ Re}(x) \Rightarrow \text{Persona}(x).$$

Tutti i re sono persone.

$$\forall x \text{ re } \text{Re}(x) \Rightarrow \text{Person}(x)$$

⁵ Seguiremo la convenzione di ordinamento dei parametri secondo cui $P(x, y)$ si legge "x è un P di y".

\forall si legge "per ogni..." ; per cui l'intera formula afferma che "per ogni x , se x è un re, allora x è una persona". Il simbolo x prende il nome di variabile. Per convenzione, le variabili si scrivono in minuscole. Una variabile da sola è anche un termine, e per questo motivo può anche essere usata come argomento di una funzione, come in GambaSinistra(x). Un termine che non comprende variabili si chiama termine ground.

variabile

Intuitivamente la formula $\forall x P$, dove P è una qualsiasi espressione logica, afferma che P vale per ogni oggetto x . Più precisamente, $\forall x P$ vale true in un dato modello secondo una data interpretazione se P è vera in tutte le possibili interpretazioni estese costruite a partire da quell'interpretazione specificando l'elemento del dominio a cui x fa riferimento.

termine ground

Tutto ciò sembra molto complicato, ma in effetti è solo un modo formale di enunciare il significato intuitivo della quantificazione universale. Considerate il modello della Figura 8.2 e la sua interpretazione intesa. Possiamo estendere tale interpretazione in cinque modi:

interpretazioni estese

 \downarrow

$x \ni$ den
Interpretation
come tutte gli
elementi del
dominio o.

$x \rightarrow$ Riccardo Cuor di Leone

$x \rightarrow$ Re Giovanni

$x \rightarrow$ la gamba sinistra di Riccardo

$x \rightarrow$ la gamba sinistra di Giovanni

$x \rightarrow$ la corona.

La formula universalmente quantificata $\forall x Re(x) \Rightarrow Persona(x)$ è vera, nell'interpretazione originale, se la formula $Re(x) \Rightarrow Persona(x)$ è vera in ognuna delle cinque interpretazioni estese. In altre parole, la formula universalmente quantificata è equivalente all'asserzione delle cinque formule seguenti.

Riccardo Cuor di Leone è un re \Rightarrow Riccardo Cuor di Leone è una persona.

Re Giovanni è un re \Rightarrow Re Giovanni è una persona.

La gamba sinistra di Riccardo è un re \Rightarrow la gamba sinistra di Riccardo è una persona.

La gamba sinistra di Giovanni è un re \Rightarrow la gamba sinistra di Giovanni è una persona.

La corona è un re \Rightarrow la corona è una persona.

Esaminiamo attentamente questo insieme di asserzioni. Dato che, nel nostro modello, l'unico re è Giovanni, la seconda formula afferma che egli è una persona, come ci potremmo aspettare. Ma che dire delle altre quattro formule, che sembrano ragionare di gambe e corone? Anch'esse fanno parte del significato di "tutti i re sono persone"? In effetti, anche le altre quattro asserzioni sono vere nel modello, ma non affermano nulla sulla natura umana di gambe e corone, e neppure dello stesso Riccardo. La ragione sta nel fatto che nessuno di questi oggetti è un re. Guardan-

do la tabella di verità di \Rightarrow (Figura 7.8), possiamo vedere che un'implicazione è sempre vera quando le sue premesse sono false, in modo del tutto indipendente dalla verità della conclusione. Così, scrivendo la formula universalmente quantificata, che equivale a un'intera lista di formule singole, asseriamo la verità della conclusione solo per quegli oggetti per cui è vera la premessa, e non diciamo nulla di tutti quelli per cui la premessa è falsa. La tabella di verità di \Rightarrow risulta quindi ideale per scrivere regole generali mediante l'uso di quantificatore universali.

Un errore comune, commesso frequentemente anche dai lettori diligenti che hanno letto più volte questo paragrafo, è usare la congiunzione al posto dell'implicazione. La formula

$$\forall x \text{Re}(x) \wedge \text{Persona}(x)$$

sarebbe equivalente ad asserire

Riccardo Cuor di Leone è un re \wedge Riccardo Cuor di Leone è una persona;

Re Giovanni è un re \wedge Re Giovanni è una persona;

la gamba sinistra di Riccardo è un re \wedge la gamba sinistra di Riccardo è una persona;

e così via. Naturalmente, non è questo ciò che vogliamo affermare.

Quantificazione esistenziale (\exists)

La quantificazione universale permette di scrivere formule che riguardano tutti gli oggetti. In modo analogo, è possibile formulare asserzioni circa alcuni oggetti dell'universo senza citarli per nome, usando un quantificatore esistenziale. Per dire, ad esempio, che Re Giovanni ha una corona sulla testa, scriviamo

$$\exists x \text{Corona}(x) \wedge \text{SullaTesta}(x, \text{Giovanni}) .$$

$\exists x$ si legge "esiste un x tale che..." o anche "per qualche x ...".

Intuitivamente, la formula $\exists x P$ afferma che P è vera per almeno un oggetto x . Più precisamente, $\exists x P$ è vera in un dato modello secondo una data interpretazione se P vale true in almeno una interpretazione estesa che assegna x a un elemento del dominio. Nel nostro esempio, questo significa che deve valere almeno una delle seguenti formule.

Riccardo Cuor di Leone è una corona \wedge Riccardo Cuor di Leone è sulla testa di Giovanni.

Re Giovanni è una corona \wedge Re Giovanni è sulla testa di Giovanni.

La gamba sinistra di Riccardo è una corona \wedge la gamba sinistra di Riccardo è sulla testa di Giovanni.

La gamba sinistra di Giovanni è una corona \wedge la gamba sinistra di Giovanni è sulla testa di Giovanni.

La corona è una corona \wedge la corona è sulla testa di Giovanni.

La quinta asserzione è vera nel modello, per cui anche la formula quantificata esistenzialmente lo è. Notate che, in base alla nostra definizione, la formula sarebbe vera anche in un modello in cui Giovanni portasse due corone. Questo è del tutto compatibile con l'affermazione originale "Re Giovanni ha una corona sulla testa".⁶

Proprio come \Rightarrow sembra essere il connettivo più naturale da usare insieme a \forall , \wedge è quello che meglio si presta a essere utilizzato con \exists . Usare Δ come connettivo principale insieme a \forall portava, nell'esempio che abbiamo mostrato nel paragrafo precedente, a un'affermazione troppo forte; usare \Rightarrow con \exists conduce solitamente a formule troppo deboli. Considerate la seguente formula:

$$\exists x \text{ Corona}(x) \Rightarrow \text{SullaTesta}(x, \text{Giovanni}).$$

A prima vista, questa sembra essere un'espressione abbastanza ragionevole. Applicando la semantica, vediamo che la formula afferma che almeno una delle seguenti asserzioni è vera:

Riccardo Cuor di Leone è una corona \Rightarrow Riccardo Cuor di Leone è sulla testa di Giovanni;

Re Giovanni è una corona \Rightarrow Re Giovanni è sulla testa di Giovanni;

la gamba sinistra di Riccardo è una corona \Rightarrow la gamba sinistra di Riccardo è sulla testa di Giovanni;

e così via. Ora, un'implicazione è vera se sono vere sia la sua premessa che la sua conseguenza, oppure se la sua premessa è falsa. Così, se Riccardo Cuor di Leone non è una corona, allora la prima asserzione è vera e il quantificatore esistenziale è soddisfatto. In pratica, un'implicazione esistenzialmente quantificata è vera in ogni modello che contiene almeno un oggetto per cui la sua premessa è falsa; ne consegue che formule di questo tipo sono davvero molto deboli.

Quantificatori nidificati

Spesso vorremo esprimere formule più complesse usando più quantificatori. Il caso più semplice si ha quando i quantificatori sono dello stesso tipo: per esempio, "i fratelli sono consanguinei" può essere scritto come

$$\forall x \forall y \text{ Fratello}(x, y) \Rightarrow \text{Consanguineo}(x, y).$$

Quantificatori consecutivi dello stesso tipo possono essere scritti anche usando un solo quantificatore con più variabili. Per dire, ad esempio, che la consanguineità è una relazione simmetrica, possiamo scrivere

$$\forall x, y \text{ Consanguineo}(x, y) \Leftrightarrow \text{Consanguineo}(y, x).$$

⁶ Esiste una variante del quantificatore esistenziale, solitamente scritta $\exists!$ o $\exists!$, che significa "esiste esattamente un". Lo stesso significato può essere espresso usando l'uguaglianza, come vedremo nel Paragrafo 8.2.

In altri casi potrà capitare di mescolare quantificatori diversi. Dire che “ognuno ama qualcuno” significa che per ogni persona c’è una persona da essa amata:

$$\forall x \exists y Ama(x, y).$$

D’altra parte, per affermare che “c’è qualcuno che è amato da tutti” scriviamo

$$\exists y \forall x Ama(x, y).$$

L’ordine con cui sono scritti i quantificatori, quindi, è molto importante. Per rendere la formula più chiara possiamo inserire parentesi: $\forall x (\exists y Ama(x, y))$ afferma che *ognuno* ha una particolare proprietà, e in particolare il fatto che ama qualcuno. D’altra parte, $\exists x (\forall y Ama(x, y))$ dice che *qualcuno* nel mondo ha una particolare caratteristica, e cioè che *tutti lo amano*.

Qualche confusione può sorgere quando due quantificatori sono applicati allo stesso nome di variabile: considerate la formula

$$\forall x [Corona(x) \vee (\exists x Fratello(Riccardo, x))].$$

Qui la x in $(Fratello(Riccardo, x))$ è quantificata esistenzialmente. La regola è che la variabile “appartiene” al quantificatore più interno che la menziona; da quel punto in poi non sarà più soggetta a nessun’altra quantificazione.⁷ Un altro modo di spiegare cosa succede è dire che $\exists x Fratello(Riccardo, x)$ è una formula che riguarda Riccardo (e dice che ha un fratello), e non riguarda x ; di conseguenza scrivere $\forall x$ al suo esterno non ha alcun effetto. In effetti, in modo del tutto equivalente avremmo potuto scrivere $\exists z Fratello(Riccardo, z)$. Poiché tutto questo rappresenta una fonte di confusione, d’ora in poi useremo sempre variabili diverse.

Connessioni tra \forall e \exists

I due quantificatori in effetti sono strettamente correlati attraverso la negazione. Dire che tutti odiano i broccoli è come dire che non esiste nessuno a cui piacciono, e viceversa:

$$\forall x \neg Gradisce(x, Broccoli) \quad \text{è equivalente a} \quad \neg \exists x Gradisce(x, Broccoli).$$

Parimenti, dire che “a tutti piace il gelato” significa che non c’è proprio nessuno a cui non piaccia:

$$\forall x Gradisce(x, Gelato) \quad \text{è equivalente a} \quad \neg \exists x \neg Gradisce(x, Gelato).$$

⁷ La potenziale interferenza tra quantificatori applicati allo stesso nome di variabile è proprio la ragione per cui per definire la semantica delle formule quantificate si adotta il meccanismo, alquanto barocco, delle interpretazioni estese. L’approccio, intuitivamente più semplice, di sostituire semplicemente un oggetto per ogni occorrenza di x fallirebbe in questo esempio, perché la x in $Fratello(Riccardo, x)$ sarebbe “catturata” dalla sostituzione. Le interpretazioni estese gestiscono correttamente anche questo caso, perché l’assegnamento a x del quantificatore più interno ha la prevalenza su quello più esterno.

Dal momento che \forall è di fatto una congiunzione sull'universo degli oggetti, mentre \exists è una disgiunzione, non dovrebbe sorprendere che obbediscano alle leggi di De Morgan. Ecco le regole di De Morgan per le formule quantificate e non:

$$\begin{array}{ll} \forall x \neg P \equiv \neg \exists x P & \neg P \wedge \neg Q \equiv \neg(P \vee Q) \\ \neg \forall x P \equiv \exists x \neg P & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\ \forall x P \equiv \neg \exists x \neg P & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\ \exists x P \equiv \neg \forall x \neg P & P \vee Q \equiv \neg(\neg P \wedge \neg Q). \end{array}$$

In definitiva non è strettamente necessario disporre sia di \forall che di \exists , proprio come non è indispensabile avere sia \wedge che \vee . In ogni caso la leggibilità è più importante della parsimonia, ragion per cui continueremo a usare entrambi i quantificatori.

Uguaglianza

La logica del primo ordine comprende un altro modo di costruire formule atomiche, senza usare predicati e termini. Possiamo utilizzare il simbolo di uguaglianza per affermare che due termini fanno riferimento allo stesso oggetto. Ad esempio,

$$\text{Padre}(Giovanni) = \text{Enrico}$$

asserisce che l'oggetto a cui si riferisce $\text{Padre}(Giovanni)$ e quello a cui si riferisce Enrico sono effettivamente lo stesso. Dato che ogni interpretazione fissa il riferimento di ogni termine, determinare la verità di una formula di uguaglianza consiste semplicemente nel verificare se i due termini si riferiscono allo stesso oggetto.

Il simbolo di uguaglianza si può usare per esprimere fatti riguardanti una data funzione, come fa l'esempio appena proposto nei confronti del simbolo Padre ; lo si può usare anche con la negazione per specificare che due termini non sono lo stesso oggetto. Per dire che Riccardo ha almeno due fratelli, possiamo scrivere

$$\exists x, y \text{ Fratello}(x, \text{Riccardo}) \wedge \text{Fratello}(y, \text{Riccardo}) \wedge \neg(x = y).$$

La formula

$$\exists x, y \text{ Fratello}(x, \text{Riccardo}) \wedge \text{Fratello}(y, \text{Riccardo})$$

non ha lo stesso significato. In particolare, è vera nel modello della Figura 8.2, in cui Riccardo ha un fratello solo, considerate infatti l'interpretazione estesa in cui sia x che y sono assegnate a Re Giovanni. L'aggiunta di $\neg(x = y)$ permette di eliminare modelli simili. Come abbreviazione di $\neg(x = y)$ talvolta si usa l'abbreviazione $x \neq y$.

simbolo di uguaglianza

domini

asserzioni

query

obiettivi

query

sostituzione
lista di legami

8.3 Usare la logica del primo ordine

Ora che abbiamo definito un linguaggio logico espressivo, è giunto il momento di imparare a usarlo: il modo migliore per farlo è attraverso gli esempi. Abbiamo già visto alcune semplici formule che illustrano i vari aspetti della sintassi della logica; in questo paragrafo forniremo rappresentazioni più sistematiche di alcuni semplici domini. Nella rappresentazione della conoscenza, un dominio è semplicemente una parte del mondo riguardo a cui vogliamo esprimere appunto della conoscenza.

Cominceremo con una breve descrizione dell'interfaccia TELL/ASK per le basi di conoscenza del primo ordine. Esamineremo poi i domini delle relazioni familiari, dei numeri, degli insiemi, delle liste, e infine il mondo del wumpus. Il paragrafo successivo presenta un esempio più articolato (i circuiti elettronici) mentre nel Capitolo 10 ci occuperemo dell'intero universo.

Asserzioni e query nella logica del primo ordine

Proprio come nella logica proposizionale, per aggiungere formule alla base di conoscenza si usa TELL. Formule siffatte sono chiamate asserzioni. Ad esempio, possiamo asserire che Giovanni è un re e che tutti i re sono persone:

TELL(KB, Re(Giovanni)) → asserzione . form,
TELL(KB, $\forall x \text{ Re}(x) \Rightarrow \text{Persona}(x)$).

Per porre domande alla base di conoscenza si usa ASK: ad esempio,

ASK(KB, Re(Giovanni))

restituisce true. Le domande poste con ASK prendono il nome di query o anche obiettivi (ma non si deve fare confusione con gli stati desiderati da un agente). In generale, ogni query che è implicata logicamente dalla base di conoscenza dovrebbe avere risposta affermativa. Ad esempio, date le due asserzioni qui sopra, la query

ASK(KB, Persona(Giovanni))

Dovrebbe anch'essa restituire true. Possiamo anche costruire query quantificate, come

ASK(KB, $\exists x \text{ Persona}(x)$).

La risposta a questa query potrebbe essere true, ma in tal caso non sarebbe utile né divertente: in effetti, sarebbe come rispondere "sì" alla domanda "scusi, sa che ore sono?". Una query con variabili esistenziali pone la domanda "esiste una x tale che...", e palesemente la risposta migliore è restituire tale x . Per fornire una risposta di questo genere, la forma standard è la sostituzione / lista di legami, costituita da un insieme di coppie variabile/termine. In questo caso, date solo le due asserzioni sopra, la risposta sarebbe $\{x/Giovanni\}$. Nel caso ci siano più risposte, è possibile restituire una lista di sostituzioni.

Il dominio della parentela

Il primo esempio che consideriamo è il dominio delle relazioni di parentela, che include fatti come "Elisabetta è la madre di Carlo" e "Carlo è il padre di William" e regole come "la nonna di un individuo è la madre di uno dei suoi genitori".

Gli oggetti del nostro dominio sono tutte persone. Avremo due predicati unari, *Maschio* e *Femmina*. Le relazioni di parentela tra genitori, fratelli, coniugi e così via saranno rappresentate da predicati binari: *Genitore*, *Consanguineo*, *Fratello*, *Sorella*, *Figlio*, *Figlia*, *Progenie*, *Coniuge*, *Moglie*, *Marito*, *Nonno*, *Nipote*, *Cugino*, *Zia* e *Zio*. Per *Madre* e *Padre* useremo delle funzioni, dato che ogni persona ha esattamente un genitore maschio e una femmina (almeno secondo l'idea originale della natura).

Possiamo esaminare a turno ogni funzione e predicato e scrivere quello che sappiamo in relazione agli altri simboli. Per esempio, la madre di una persona è il suo genitore di sesso femminile:

$$\forall m, c \quad \text{Madre}(c) = m \Leftrightarrow \text{Femmina}(m) \wedge \text{Genitore}(m, c).$$

Il marito è il coniuge maschio di una persona:

$$\forall w, h \quad \text{Marito}(h, w) \Leftrightarrow \text{Maschio}(h) \wedge \text{Coniuge}(h, w).$$

Maschio e femmina sono categorie disgiunte:

$$\forall x \quad \text{Maschio}(x) \Leftrightarrow \neg \text{Femmina}(x).$$



Genitore e progenie sono relazioni inverse:

$$\forall p, c \quad \text{Genitore}(p, c) \Leftrightarrow \text{Progenie}(c, p).$$

Un nonno (o nonna) è un genitore di un genitore:

$$\forall g, c \quad \text{Nonno}(g, c) \Leftrightarrow \exists p \quad \text{Genitore}(g, p) \wedge \text{Genitore}(p, c).$$

Un consanguineo (con questo termine traduciamo l'inglese *sibling*) è un altro figlio dei genitori di un individuo:

$$\forall x, y \quad \text{Consanguineo}(x, y) \Leftrightarrow x \neq y \wedge \exists p \quad \text{Genitore}(p, x) \wedge \text{Genitore}(p, y).$$

Potremmo andare avanti così per parecchie pagine, e l'Esercizio 8.11 vi chiederà proprio questo.

Ognuna di queste formule può essere considerata un assioma nel dominio della parentela. Normalmente gli assiomi sono associati ai domini puramente matematici (e in effetti ne vedremo tra poco alcuni dedicati ai numeri), ma sono necessari in tutti i domini, rappresentando l'informazione base da cui si possono derivare conclusioni utili. I nostri assiomi di parentela sono anche definizioni: hanno tutti la forma $\forall x, y P(x, y) \Leftrightarrow \dots$. Gli assiomi definiscono la funzione *Madre* e i predicati *Marito*, *Maschio*, *Genitore*, *Nonno* e *Consanguineo* in termini di altri predicati. La "base" delle definizioni è costituita da un insieme di predicati (*Progenie*,

assioma

definizioni

Coniuge e *Femmina*) che servono da riferimento per definire gli altri. Questo è un modo molto naturale di costruire la rappresentazione di un dominio, ed è analogo al modo con cui si costruiscono i pacchetti software definendo nuove procedure sulla base di funzioni primitive di libreria. Notate che l'insieme dei predicati primitivi non dev'essere necessariamente unico: avremo potuto altrettanto bene partire dall'insieme *Genitore*, *Coniuge* e *Maschio*. In alcuni domini, come vedremo, non esiste neppure un insieme base chiaramente identificabile.

Non tutte le formule logiche riguardanti un dominio sono assiomi. Alcune sono teoremi, ovvero formule deducibili dagli assiomi stessi. Considerate ad esempio l'asserzione che la relazione di consanguineità è simmetrica:

$$\forall x, y \quad \text{Consanguineo}(x, y) \Leftrightarrow \text{Consanguineo}(y, x).$$

Questo è un assioma o un teorema? In effetti, è un teorema che segue logicamente dall'assioma che definisce la consanguineità. Se chiedessimo alla base di conoscenza il valore di verità di questa formula, il risultato dovrebbe essere *true*.

Da un punto di vista puramente logico, una base di conoscenza necessita solo degli assiomi e non dei teoremi, dato che questi ultimi non estendono l'insieme di conclusioni che possono essere derivate. In pratica, tuttavia, i teoremi sono essenziali per ridurre il costo computazionale della derivazione di nuove formule. Senza teoremi un sistema dovrebbe ripartire ogni volta dai principî base, come un matematico che dovesse derivare nuovamente le regole dell'analisi per ogni nuovo problema.

Non tutti gli assiomi sono definizioni; alcuni si limitano a fornire informazioni generali aggiuntive riguardanti certi predicati. In effetti, alcuni predicati non hanno neppure una definizione completa perché non abbiamo abbastanza informazioni per caratterizzarli pienamente. Per esempio, non c'è un modo semplice di completare la formula:

$$\forall x \text{ Persona}(x) \Leftrightarrow \dots$$

Fortunatamente, la logica del primo ordine ci permette di usare il predicato *Persona* senza definirlo completamente. Invece di far ciò possiamo scrivere specifiche parziali delle caratteristiche possedute da ogni persona e di quelle che fanno sì che un individuo sia una persona:

$$\begin{aligned} \forall x \text{ Persona}(x) &\Rightarrow \dots \\ \forall x \dots &\Rightarrow \text{Persona}(x). \end{aligned}$$

Gli assiomi possono anche essere “semplici fatti” come *Maschio(Jim)* e *Coniuge(Jim, Laura)*. Tali fatti costituiscono la descrizione di specifiche istanze di problemi, permettendo di rispondere a domande altrettanto specifiche. Le risposte saranno quindi teoremi implicati dagli assiomi. Spesso potrà capitare di attendere risposte che invece la base di conoscenza non è in grado di dare: ad esempio, da *Maschio(George)* e *Coniuge(George, Laura)* ci si potrebbe aspettare di inferire *Femmina(Laura)*; tuttavia per far questo gli assiomi che abbiamo fornito non sono sufficienti. Questo significa che ne manca uno: l'Esercizio 8.8 vi chiederà di formularlo.

Numeri, insiemi e liste

I numeri sono forse l'esempio più lampante di come sia possibile costruire una grande teoria partendo da un numero microscopico di assiomi. Qui descriveremo la teoria dei numeri naturali, ovvero degli interi non negativi. Utilizzeremo un predicato $NumNat$ che sarà vero per i numeri naturali. Ci servirà poi un solo simbolo di costante 0 e un solo simbolo di funzione, S (successore). Gli assiomi di Peano definiscono i numeri naturali e l'addizione.⁸ I numeri naturali sono definiti ricorsivamente:

numeri naturali

assiomi di Peano

$NumNat(0)$ → costante.

$\forall n \ NumNat(n) \Rightarrow NumNat(S(n))$. + m se m è Natura → è Natura successore.

Questo significa che 0 è un numero naturale e che, per ogni oggetto n, se n è un numero naturale allora lo è anche il suo successore $S(n)$. Ne consegue che i numeri naturali sono 0, $S(0), S(S(0))$ e così via. Dobbiamo scrivere degli assiomi per restringere la funzione successore:

$\forall n \ 0 \neq S(n)$.

$\forall m, n \ m \neq n \Rightarrow S(m) \neq S(n)$.

Ora possiamo definire l'addizione in termini di funzione successore:

$\forall m \ NumNat(m) \Rightarrow + (0, m) = m$

$\forall m, n \ NumNat(m) \wedge NumNat(n) \Rightarrow + (S(m), n) = S(+ (m, n))$.

Il primo di questi assiomi dice che aggiungere 0 a qualsiasi numero naturale m restituisce m stesso. Notate l'uso del simbolo di funzione binaria “+” nel termine $+ (m, 0)$; normalmente lo avremmo scritto $m + 0$, usando la notazione infissa. Quella usata qui, invece, prende il nome di notazione prefissa. Per rendere più facile la lettura delle formule che riguardano i numeri permetteremo l'uso della notazione infissa: inoltre possiamo scrivere $S(n)$ come $n + 1$, e così il secondo assioma diventa

notazione infissa

notazione prefissa

$\forall m, n \ NumNat(m) \wedge NumNat(n) \Rightarrow (m + 1) + n = (m + n) + 1$.

Quest'assioma riduce l'addizione all'applicazione ripetuta della funzione successore.

L'uso della notazione infissa è un esempio di zucchero sintattico, un'estensione o abbreviazione della sintassi standard che non modifica la semantica. Ogni formula che utilizza lo zucchero può essere “de-zuccherata” per produrre la formula equivalente nella sintassi ordinaria della logica del primo ordine.

zucchero sintattico

⁸ Gli assiomi di Peano includono anche il principio di induzione, che è una formula di logica del secondo ordine e non del primo. Spiegheremo l'importanza di questa distinzione nel Capitolo 9.

A questo punto è molto semplice definire la moltiplicazione come ciclo di addizioni ripetute, l'elevamento a potenza come ciclo di moltiplicazioni, la divisione intera con il resto, i numeri primi e così via. Così, l'intera teoria dei numeri (inclusa la crittografia) può essere costruita partendo da una costante, una funzione, un predicato e quattro assiomi.

Anche il dominio degli insiemi è fondamentale per la matematica, come del resto per il ragionamento (tanto che è possibile costruire la teoria dei numeri partendo da quella degli insiemi). Vogliamo essere in grado di rappresentare singoli insiemi, tra cui l'insieme vuoto: ci serve un modo per costruire insiemi aggiungendo singoli elementi o prendendo l'unione o l'intersezione di due insiemi. Vorremo sapere se un elemento fa parte o no di un insieme e avere il modo di distinguere gli insiemi da oggetti che non lo sono.

Come zucchero sintattico useremo il normale vocabolario della teoria degli insiemi. L'insieme vuoto è una costante, indicata con $\{\}$. C'è un solo predicato unario $Set(s)$ che vale true per gli insiemi. I prediciati binari sono $x \in s$ (x è un elemento dell'insieme s) e $s_1 \subseteq s_2$ (s_1 è un sottoinsieme, non necessariamente proprio, di s_2). Le funzioni binarie sono $s_1 \cap s_2$ (intersezione), $s_1 \cup s_2$ (unione) e $\{x|s\}$ (l'insieme risultante dall'aggiunta dell'elemento x a s). Un possibile insieme di assiomi è il seguente.

1. Gli unici insiemi sono quello vuoto e quelli costruiti aggiungendo qualcosa a un insieme:

$$\forall s \ Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \ Set(s_2) \wedge s = \{x|s_2\}) .$$

2. L'insieme vuoto non ha alcun elemento; in altre parole non c'è modo di scomporlo nella coppia formata da un insieme più piccolo e un elemento singolo:

$$\neg \exists x, s \ \{x|s\} = \{ \} .$$

3. Aggiungere a un insieme un elemento che vi è già presente non ha alcun effetto:

$$\forall x, s \ x \in s \Leftrightarrow s = \{x|s\} .$$

4. I soli membri di un insieme sono gli elementi che vi sono stati aggiunti. Per esprimere questo ricorsivamente, diciamo che x è membro di s se e solo se s è uguale a un insieme s_2 a cui è stato aggiunto un elemento y , laddove y è lo stesso di x o x è un membro di s_2 :

$$\forall x, s \ x \in s \Leftrightarrow [\exists y, s_2 (s = \{y|s_2\} \wedge (x = y \vee x \in s_2))] .$$

5. Un insieme è un sottoinsieme di un altro insieme se e solo se tutti i suoi membri sono anche membri del secondo insieme:

$$\forall s_1, s_2 \ s_1 \subseteq s_2 \Leftrightarrow (\forall x \ x \in s_1 \Rightarrow x \in s_2) .$$

6. Due insiemi sono uguali se e solo se ognuno è un sottoinsieme dell'altro:

$$\forall s_1, s_2 \ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1) .$$

7. Un oggetto appartiene all'intersezione di due insiemi se e solo se è membro di entrambi:

$$\forall x, s_1, s_2 \ x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2) .$$

8. Un oggetto appartiene all'unione di due insiemi se e solo se è membro dell'uno o dell'altro:

$$\forall x, s_1, s_2 \ x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2) .$$

Le liste sono simili agli insiemi, con la differenza che sono ordinate e che lo stesso elemento può apparire più volte. Per esse possiamo usare il vocabolario del Lisp: *Nil* è la lista costante priva di elementi; *Cons*, *Append*, *First* e *Rest* sono funzioni; *Find* è il predicato che verifica se un elemento è presente nella lista. *List?* è un predicato che vale solo per le liste. Come nel caso degli insiemi, nelle formule logiche che coinvolgono le liste è frequente l'uso di zucchero sintattico. Così, la lista vuota si indica con []; il termine *Cons(x, y)*, dove *y* è una lista non vuota, si scrive *[x|y]*. Il termine *Cons(x, Nil)* (ovvero la lista che contiene il solo elemento *x*) si scrive *[x]*. Una lista di più elementi, come *[A, B, C]*, corrisponde al termine nidificato *Cons(A, Cons(B, Cons(C, Nil)))*. Nell'Esercizio 8.14 vi sarà richiesto di scrivere gli assiomi per le liste.

liste

Il mondo del wumpus

Nel Capitolo 7 abbiamo già scritto alcuni assiomi in logica proposizionale per il mondo del wumpus. Gli assiomi in logica del primo ordine che forniremo ora sono molto più concisi e catturano in modo molto naturale esattamente ciò che vogliamo esprimere.

Ricorderete che l'agente nel mondo del wumpus riceve un vettore di percezioni composto da cinque elementi. La formula del primo ordine memorizzata nella base di conoscenza deve includere sia la percezione che l'istante nel quale è stata ricevuta; in caso contrario, l'agente farebbe una gran confusione tra le percezioni di tempi diversi. Una tipica formula riguardante le percezioni sarà quindi

$$\text{Percezione}([\text{Fetore}, \text{Brezza}, \text{Scintillio}, \text{None}, \text{None}], 5) \leftarrow \text{tempo} \rightarrow \text{ora} .$$

Qui *Percezione* è un predicato binario mentre *Fetore* etc. sono costanti raccolte in una lista. Le azioni possibili possono essere rappresentate mediante termini logici:

$$\text{Gira}(\text{Destra}), \text{Gira}(\text{Sinistra}), \text{Avanti}, \text{Scocca}, \text{Afferra}, \text{Lascia}, \text{Sali}.$$

Per determinare l'azione migliore, il programma agente costruirà una query come

$$\exists a \ BestAction(a, 5) .$$

ASK dovrebbe risolvere questa query restituendo una sostituzione come {a/Afferra}. Il programma agente potrà quindi restituire Afferra come azione da effettuare, ma prima dovrà comunicare alla sua base di conoscenza (mediante TELL) che sta per intraprendere un'azione Afferra.

I dati relativi alle percezioni implicano, da soli, alcuni fatti riguardanti lo stato corrente. Ad esempio:

$$\begin{aligned}\forall t, s, g, m, c \ Percezione([s, Brezza, g, m, c], t) &\Rightarrow Brezza(t), \\ \forall t, s, b, m, c \ Percezione([s, b, Scintillio, m, c], t) &\Rightarrow Scintillio(t),\end{aligned}$$

e così via. Queste regole costituiscono una forma molto semplice di processo di ragionamento che prende appunto il nome di percezione, e che studieremo approfonditamente nel Capitolo 24, nel 2° volume. Notate la quantificazione sul tempo t: nel calcolo proposizionale, avremmo avuto bisogno di una copia di ogni formula per ogni passo temporale.

Semplificati comportamenti "di riflesso" possono essere implementati da formule di implicazione quantificata. Per esempio, possiamo scrivere

$$\forall t \ Scintillio(t) \Rightarrow BestAction(Afferra, t).$$

Date la percezione e le regole viste sopra, questo fornirebbe la conclusione desiderata BestAction(Afferra, 5): la cosa giusta da fare è raccogliere l'oro. Notate la corrispondenza diretta tra questa regola e il collegamento percezione-azione nell'agente basato su circuiti della Figura 7.20; si può dire che il circuito stesso realizzi una quantificazione implicita sul tempo.

Fin qui le formule che abbiamo visto riguardanti il tempo sono state tutte sincroniche, mettendo in relazione proprietà di uno stato del mondo con altre proprietà dello stesso stato. Formule che permettono di esprimere ragionamento "attraverso il tempo" sono chiamate diacroniche: l'agente, ad esempio, deve sapere come mettere insieme l'informazione riguardante la posizione precedente con l'azione appena intrapresa per determinare la sua posizione corrente. Rimanderemo la discussione delle formule diacroniche al Capitolo 10; per adesso daremo per scontato di essere in grado di svolgere le inferenze necessarie al calcolo della posizione e degli altri prediciati dipendenti dal tempo.

Abbiamo rappresentato le percezioni e le azioni; ora è giunto il momento di rappresentare l'ambiente stesso. Cominceremo con gli oggetti: i candidati più ovvi sono le stanze, i pozzi e il wumpus. Potremmo dare un nome a ogni locazione – Stanza_{1,2} e così via – ma poi saremmo obbligati a esprimere il fatto che Stanza_{1,2} e Stanza_{1,3} sono adiacenti come informazione aggiuntiva, e per di più saremmo costretti a farlo per ogni coppia di stanze. In questo caso è meglio usare un termine complesso in cui la riga e la colonna della posizione compaiono come indici interni: ad esempio possiamo usare una lista come [1, 2]. A questo punto possiamo definire le stanze adiacenti nel modo seguente:

$$\begin{aligned}\forall x, y, a, b \ Adiacente([x, y], [a, b]) &\Leftrightarrow \\ [a, b] \in \{[x+1, y], [x-1, y], [x, y+1], [x, y-1]\} &.\end{aligned}$$

sincroniche

diacroniche

*Stanze
Colonne*

Potremmo anche dare un nome diverso a ogni pozzo, ma anche questa scelta sarebbe inappropriata, benché la ragione sia diversa: infatti non c'è alcun motivo di distinguere un pozzo dall'altro.⁹ È molto più semplice usare un predicato unario *Pozzo* che sarà vero per le stanze che contengono pozzi. Infine, dato che esiste esattamente un wumpus al posto di un predicato unario si può usare altrettanto bene una costante *Wumpus*. Il wumpus occupa esattamente una stanza, per cui sarà una buona idea usare una funzione come *Tana(Wumpus)* per denominarla. Questo ci permette di evitare completamente il ponderoso insieme di formule richieste dal calcolo proposizionale per dire che c'è una e una sola stanza contenente un wumpus (e pensate a quanto peggiorerebbe il caso proposizionale se i wumpus fossero due).

La posizione dell'agente cambia nel tempo, per cui scriviamo *Pos(Agente, s, t)* per indicare che l'agente si trova nella stanza *s* nell'istante *t*. Data la sua posizione corrente, l'agente può dedurre le proprietà della stanza dalle sue percezioni in quel momento. Per esempio, se l'agente percepisce una brezza, allora quella stanza è ventosa:

$$\forall s, t \ Pos(\text{Agente}, s, t) \wedge \text{Brezza}(t) \Rightarrow \text{Ventosa}(s) .$$

È utile sapere che una stanza è ventosa, dato che sappiamo che i pozzi non possono muoversi: notate infatti che *Ventosa* non prevede un argomento che specifichi il tempo.

Avendo scoperto quali locazioni sono ventose o puzzolenti e, cosa molto importante, quali non lo sono, l'agente può dedurre la posizione dei pozzi e dello stesso wumpus. Due tipi di regole sincroniche ci permettono di effettuare deduzioni simili.

♦ Regole diagnostiche

regole diagnostiche

Le regole diagnostiche, partendo da fatti osservati, ci conducono alle cause nascoste. Per trovare i pozzi, due intuibili regole ci dicono che se una stanza è ventosa qualcun'altra adiacente deve contenere un pozzo,

$$\forall s \ Ventosa(s) \Rightarrow \exists r \ Adiacente(r, s) \wedge \text{Pozzo}(r) ,$$

⁹ In modo analogo la maggior parte delle persone non distingue tra di loro gli uccelli che, quando si fa inverno, migrano verso le regioni più calde. Un ornitologo che volesse studiarne gli schemi di migrazione, la percentuale di sopravvivenza e così via sarà effettivamente costretto a dare un nome a ogni uccello per mezzo di un anello intorno alla zampa, a causa della necessità di tener traccia di ogni individuo.

e che se una stanza non è ventosa, nessuna di quelle adiacenti conterrà un pozzo:¹⁰

$$\forall s \neg \text{Ventosa}(s) \Rightarrow \neg \exists r \text{ Adiacente}(r, s) \wedge \text{Pozzo}(r).$$

Combinando queste due, otteniamo la formula bicondizionale

$$\boxed{\forall s \text{ Ventosa}(s) \Leftrightarrow \exists r \text{ Adiacente}(r, s) \wedge \text{Pozzo}(r)}. \quad (8.3)$$

Regole causali

Le regole causali riflettono la direzione del meccanismo di causa-effetto nel mondo: qualche sua proprietà nascosta è all'origine della generazione di specifiche percezioni. Ad esempio, un pozzo fa sì che tutte le stanze adiacenti siano ventose,

$$\forall r \text{ Pozzo}(r) \Rightarrow (\forall s \text{ Adiacente}(r, s) \Rightarrow \text{Ventosa}(s))$$

e se tutte le stanze adiacenti a una stanza data sono prive di pozzi, tale stanza non sarà ventosa:

$$\forall s [\forall r \text{ Adiacente}(r, s) \Rightarrow \neg \text{Pozzo}(r)] \Rightarrow \neg \text{Ventosa}(s).$$

Con un po' di pazienza, si può dimostrare che queste due formule, prese insieme, sono logicamente equivalenti al bicondizionale dell'Equazione (8.3). Il bicondizionale stesso può essere considerato una regola causale, perché specifica in che modo il valore di verità di Ventosa è generato partendo dallo stato del mondo.

ragionamento basati
su modello

Quelli che sfruttano le regole causali sono chiamati sistemi di ragionamento basati su modello, perché le regole formano un modello del funzionamento dell'ambiente. La distinzione tra ragionamento basato su modello e ragionamento diagnostico è importante in molti campi dell'IA. La diagnosi medica in particolare è stata un'area di ricerca molto attiva, in cui gli approcci basati sull'associazione diretta tra sintomi e malattie (su regole diagnostiche, appunto) sono stati gradualmente rimpiazzati da approcci che utilizzano un modello esplicito del processo patologico in atto e delle modalità con cui esso si manifesta attraverso i sintomi. Quest'argomento verrà ripreso nel Capitolo 13, nel 2° volume.

Qualsiasi sia il tipo di rappresentazione usato dall'agente, se gli assiomi descrivono correttamente e completamente il funzionamento del mondo e come vengono prodotte le percezioni, ogni procedura di inferenza logica completa potrà inferire la più ricca descrizione possibile del mondo, date le percezioni disponibili. In questo modo il

¹⁰ C'è una tendenza naturale, nelle persone, a dimenticare di scrivere informazioni in forma negativa come questa. In una conversazione questa tendenza è più che normale: in effetti sarebbe molto strano dire "su questo tavolo ci sono due tazze, e non ce ne sono tre o più", sebbene dal punto di vista strettamente formale la prima frase sia vera anche quando le tazze sono tre. Ritorneremo su quest'argomento nel Capitolo 10.

progettista dell'agente si può concentrare sull'espressione corretta della conoscenza, senza preoccuparsi troppo dei processi di deduzione. Inoltre, come abbiamo visto, la logica del primo ordine può rappresentare il mondo del wumpus in modo altrettanto conciso della descrizione in linguaggio naturale che abbiamo fornito nel Capitolo 7.

8.4 Ingegneria della conoscenza nella logica del primo ordine

Il paragrafo precedente ha illustrato l'uso della logica del primo ordine per rappresentare la conoscenza in tre semplici domini. In questo paragrafo descriveremo il processo generale di costruzione di una base di conoscenza; tale processo prende il nome di ingegneria della conoscenza (*knowledge engineering*). Un ingegnere della conoscenza investiga un particolare dominio, impara quali concetti sono importanti e scrive una rappresentazione formale degli oggetti al suo interno e delle relazioni tra essi. Presenteremo il processo di ingegneria della conoscenza nel dominio dei circuiti elettronici, che dovrebbe esservi già abbastanza familiare, in modo da rendere possibile concentrarsi al massimo sui problemi di rappresentazione. L'approccio adottato è quello più adatto allo sviluppo di basi di conoscenza *di uso specifico*, il cui dominio è precisamente limitato e di cui si conosce già la gamma di possibili interrogazioni. Le basi di conoscenza *di uso generale*, che possono essere interrogate riguardo all'intero scibile umano, saranno discusse nel Capitolo 10.

ingegneria della
conoscenza

Il processo di ingeengneria della conoscenza

Tra i diversi progetti di ingegneria della conoscenza ci sono grandi differenze di contenuto, dimensione e difficoltà, ma tutti includono i seguenti passi.

1. Identificare il compito della base di conoscenza. L'ingegnere della conoscenza deve delineare la gamma di domande a cui la KB dovrà rispondere e le categorie di fatti disponibili per ogni specifica istanza di problema. Per esempio, la base di conoscenza del wumpus dovrà essere in grado di scegliere le azioni da intraprendere o dovrà semplicemente rispondere a domande riguardanti il contenuto dell'ambiente? I fatti percepiti dai sensori includeranno la posizione corrente? Il compito della KB determinerà quale conoscenza dev'essere rappresentata per collegare istanze del problema alle specifiche risposte. Questo passo è analogo al processo di descrizione PEAS nella progettazione degli agenti, visto nel Capitolo 2.
2. Raccogliere la conoscenza rilevante. L'ingegnere della conoscenza potrebbe già essere un esperto del dominio, o potrebbe lavorare insieme ad altri esperti per estrarre quello che sanno: quest'ultimo processo prende il nome di

acquisizione della conoscenza

ontologia

acquisizione della conoscenza. In questa fase non viene utilizzata una rappresentazione formale: lo scopo è comprendere l'entità della base di conoscenza, determinata dal suo compito, e capire come funziona effettivamente il dominio. Nel mondo del wumpus, definito da un insieme di regole artificiali, è facile identificare la conoscenza rilevante (ma notate che la definizione di stanze adiacenti non era stata fornita esplicitamente sotto forma di regola). Nei domini reali, il problema della rilevanza può risultare abbastanza difficile; un sistema dedicato alla simulazione di progetti VLSI, ad esempio, potrebbe prendere in considerazione o no la presenza di capacità parassite o dell'effetto pellicolare.

3. *Definire un vocabolario di predicati, funzioni e costanti.* A questo punto occorre tradurre i concetti importanti del dominio in nomi di simboli logici. Nel far questo sorgono molti problemi di *stile*: come lo stile di programmazione, anche quello di ingegneria della conoscenza può avere un impatto significativo sul successo di un progetto. Ad esempio, i pozzi devono essere rappresentati da oggetti o da predicati unari applicati alle stanze? L'orientamento fisico dell'agente sarà una funzione o un predicato? La posizione del wumpus dipenderà dal tempo? Una volta fatte queste scelte, il risultato sarà un vocabolario che prende il nome di ontologia del dominio. Con la parola ontologia si intende una particolare teoria riguardante la natura dell'esistenza. Un'ontologia definisce le categorie di oggetti esistenti, ma non le loro specifiche caratteristiche e le relazioni tra esse.
4. *Codificare la conoscenza generale riguardante il dominio.* L'ingegnere della conoscenza scrive gli assiomi per tutti gli elementi del vocabolario. Questo definisce precisamente (nei limiti del possibile) il significato di ogni termine, permettendo all'esperto del dominio di verificare il contenuto. Spesso questo passo rivela malintesi o lacune nel vocabolario, che dovranno essere colmate tornando al passo precedente e ripetendo il processo iterativamente.
5. *Codificare una descrizione della specifica istanza del problema.* Se l'ontologia è ben definita, questo passo dovrebbe risultare semplice: si tratta di scrivere semplici formule atomiche che riguardano istanze di concetti che fanno già parte dell'ontologia. Per un agente logico le istanze dei problemi sono fornite dai sensori, e la base di conoscenza iniziale è riempita di formule aggiuntive allo stesso modo con cui i programmi tradizionali ricevono dati di input.
6. *Interrogare la procedura di inferenza e ottenere da essa risposte.* A questo punto si possono raccogliere i frutti del proprio lavoro: la procedura di inferenza, partendo dagli assiomi e dai fatti specifici del problema, sarà in grado di derivare i fatti che ci interessa conoscere.
7. *Fare il debugging della base di conoscenza.* Purtroppo le risposte saranno raramente corrette al primo tentativo. Per essere più precisi le risposte saranno quelle giuste per la base di conoscenza così com'è scritta, presumendo che la pro-

cedura di inferenza sia corretta, ma potranno essere diverse da quelle che ci saremmo aspettati. Se per esempio manca un assioma alcune interrogazioni non potranno avere risposta: ne potrebbe scaturire un processo di debugging al quanto faticoso. Assiomi mancanti, o troppo deboli, possono essere identificati facilmente cercando interruzioni inattese nella catena di ragionamento. Per esempio, se la base di conoscenza include uno degli assiomi diagnostici per i pozzi,

$$\forall s \text{ Ventosa}(s) \Rightarrow \exists r \text{ Adiacente}(r, s) \wedge \text{Pozzo}(r),$$

ma non l'altro, l'agente non sarà mai capace di dimostrare l'*assenza* dei pozzi. Gli assiomi scorretti possono essere identificati perché rappresentano affermazioni false riguardo il mondo. Per esempio, la formula

$$\forall x \text{ NumeroDiGambe}(x, 4) \Rightarrow \text{Mammifero}(x)$$

è falso per i rettili, gli anfibi e, cosa ancora più importante, i tavoli. *La falsità di questa formula può essere determinata indipendentemente dal resto della base di conoscenza.* Al contrario, un tipico errore in un programma potrebbe avere quest'aspetto:

```
offset = posizione + 1.
```

È impossibile determinare se quest'istruzione è corretta senza esaminare il resto del programma per vedere se, ad esempio, offset è usato per riferirsi alla posizione corrente o a quella immediatamente successiva, o se il valore di posizione è stato modificato da un'altra istruzione dimodoché anche offset dovrebbe essere aggiornato.

Per comprendere appieno questo processo in sette passi, lo applicheremo ora a un esempio più esteso: il dominio dei circuiti elettronici.

Il dominio dei circuiti elettronici

Seguendo la metodologia a sette passi del knowledge engineering, svilupperemo un'ontologia e una base di conoscenza che ci permetteranno di ragionare sui circuiti elettrici del tipo mostrato nella Figura 8.4.

Identificare il compito

È possibile identificare molti compiti nel dominio dei circuiti digitali. Al livello di astrazione più alto, si può analizzarne la funzionalità: ad esempio, il circuito nella Figura 8.4 esegue effettivamente un'addizione corretta? Se tutti gli input sono "alti", qual è l'output della porta A₂? È anche interessante porre domande sulla struttura del circuito stesso. Ad esempio, quali sono le porte collegate al primo morsetto di input? Il circuito contiene anelli di retroazione? In questo paragrafo, il compito del-



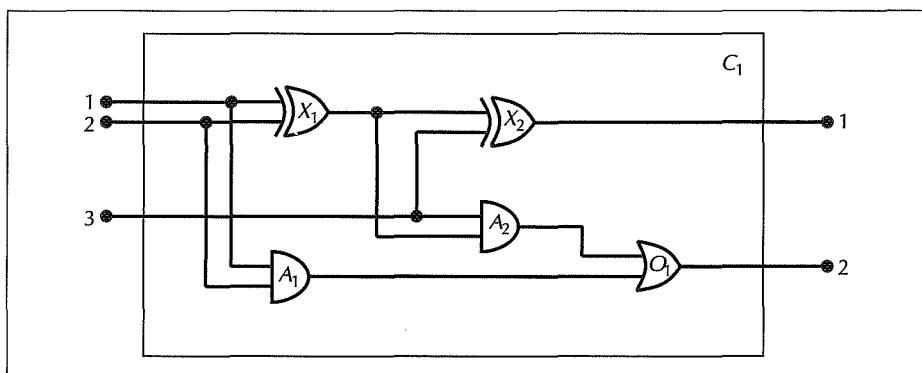


Figura 8.4 Un circuito digitale C_1 che dovrebbe essere un sommatore a un bit con riporto. I primi due input rappresentano gli addendi, il terzo è il riporto in ingresso. Il primo output è la somma, il secondo il riporto passato al sommatore successivo. Il circuito contiene due porte XOR, due porte AND e una porta OR.

la base di conoscenza sarà rispondere a domande come queste. Naturalmente ci sono livelli di analisi più dettagliati, che prendono in considerazione i ritardi temporali, l'area occupata dal circuito, il consumo di potenza, il costo di produzione e così via. Ognuno di questi livelli richiederebbe conoscenza aggiuntiva.

Raccogliere la conoscenza rilevante

Cosa sappiamo dei circuiti digitali? Per i nostri scopi, è sufficiente sapere che sono fatti di piste e di porte logiche. I segnali si propagano lungo le piste fino agli input delle porte, che producono di conseguenza un segnale di output che viaggia su un'altra pista. Per determinare il valore dei segnali, dobbiamo conoscere il funzionamento delle porte logiche. Ne esistono quattro tipi diversi: le porte AND, OR e XOR hanno due morsetti di input, quelle NOT soltanto uno. Tutte le porte logiche hanno un solo morsetto di output. I circuiti, come le porte, hanno morsetti di input e di output.

Per ragionare sulla funzionalità di un circuito e sulla sua struttura non è necessario parlare esplicitamente delle piste, dei loro cammini e delle giunzioni che li uniscono. Basta conoscere i collegamenti tra i morsetti; è sufficiente dire che l'output di una porta è collegato all'input di un'altra senza menzionare il cavo che li mette fisicamente in contatto. Molti altri aspetti del dominio sono irrilevanti ai fini della nostra analisi, come la dimensione, la forma, il colore o il costo dei vari componenti.

Se il nostro scopo andasse oltre la semplice verifica del funzionamento al livello delle porte, la nostra ontologia sarebbe diversa. Ad esempio, se fossimo interessati al debugging di circuiti difettosi, sarebbe stata una buona idea includere nell'ontologia le piste di rame, dato che una pista difettosa può corrompere il se-

gnale trasportato. Per risolvere gli errori di sincronizzazione avremmo dovuto prendere in considerazione i ritardi temporali introdotti dai transitori elettrici nelle porte. Se fossimo interessati alla costruzione di un prodotto in grado di essere venduto facilmente, sarebbe stato importante il costo del circuito e la sua velocità rispetto ai concorrenti sul mercato.

Definire un vocabolario

Ora sappiamo che vogliamo parlare di circuiti, morsetti, segnali e porte. Il passo successivo è scegliere le funzioni, i predicati e le costanti per rappresentare tali entità. Cominceremo dalle singole porte e ci muoveremo verso l'alto.

Prima di tutto, dobbiamo poter distinguere una porta dalle altre. Per far questo possiamo assegnare un nome a ognuna di esse, per mezzo di una costante: X_1 , X_2 e così via. Benché ogni posta sia collegata al circuito in un modo particolare, il suo *comportamento*, cioè il modo in cui trasforma i segnali di input in un segnale di output, dipende unicamente dal suo *tipo*. Per indicare il tipo di una porta possiamo usare una funzione.¹¹ Possiamo scrivere, ad esempio, $\text{Tipo}(X_1) = \text{XOR}$. Questo introduce la costante *XOR* per un tipo particolare di porta; le altre costanti saranno *OR*, *AND* e *NOT*. La funzione *Tipo* non è l'unico modo di esprimere la distinzione ontologica: avremmo potuto usare un predicato binario, $\text{Tipo}(X_1, \text{XOR})$, o diversi predicati di tipo singoli, come $\text{XOR}(X_1)$. Tutte queste soluzioni avrebbero funzionato bene ma scegliendo la funzione, data la sua peculiare semantica, non abbiamo più bisogno di aggiungere un assioma che specifichi che una porta può avere un solo tipo.

Passiamo a considerare i morsetti. Una porta o un circuito possono avere uno o più input e uno o più output. Potremmo semplicemente assegnare a ognuno di essi un nome con una costante, proprio come abbiamo fatto con le porte. Così, la porta X_1 potrebbe avere dei morsetti chiamati $X_1\text{In}_1$, $X_1\text{In}_2$ e $X_1\text{Out}_1$. In ogni caso, si dovrebbe cercare di resistere alla tendenza di generare lunghi nomi composti. Assegnare a qualcosa il nome $X_1\text{In}_1$ non la rende il primo input di X_1 ; sarà ancora necessario specificare questo fatto con un'esplicita asserzione. Probabilmente è meglio utilizzare una funzione, proprio come ne abbiamo usata una per denominare *GambaSinistra(Giovanni)* la gamba sinistra di Re Giovanni. Diciamo quindi che $\text{In}(1, X_1)$ denota il primo morsetto di input per la porta logica X_1 . Per i morsetti di output utilizzeremo un'analogia funzione *Out*.

¹¹ Notate che abbiamo usato nomi che cominciavano con la lettera appropriata (A_1 , X_1 e così via) unicamente per facilitare la lettura degli esempi. La base di conoscenza deve comunque contenere informazioni circa il tipo di ogni porta.

I collegamenti tra le porte logiche possono essere rappresentati con il predicato *Collegati*, che prende come argomenti due morsetti, come in $\text{Collegati}(\text{Out}(1, X_1), \text{In}(1, X_2))$.

Infine, dobbiamo sapere se un determinato segnale è acceso o spento (alto o basso, vero o falso e così via). Una possibilità è usare un predicato unario, *On*, che varrà true quando il segnale a un morsetto è acceso. Questo, tuttavia, renderebbe alquanto difficile esprimere interrogazioni come “quali sono i possibili valori dei segnali ai morsetti di output del circuito C1?”. Di conseguenza introdurremo come oggetti i due “valori del segnale” 1 e 0, nonché una funzione *Segnale* che prende come argomento un morsetto e denota il valore del segnale in quel punto.

Codificare la conoscenza generale riguardante il dominio

Dover specificare poche regole generali è segno di una buona ontologia. Quando poi ogni regola può essere espressa in modo chiaro e conciso significa che è buono anche il nostro vocabolario. In questo esempio occorrono solo sette regole semplici per descrivere tutto quello che dobbiamo sapere sui circuiti.

1. Se due morsetti sono collegati devono avere lo stesso segnale:

$$\forall t_1, t_2 \quad \text{Collegati}(t_1, t_2) \Rightarrow \text{Segnale}(t_1) = \text{Segnale}(t_2)$$

2. A ogni morsetto il segnale vale 1 o 0, ma non entrambi:

$$\begin{aligned} \forall t \quad & \text{Segnale}(t) = 1 \vee \text{Segnale}(t) = 0 \\ & 1 \neq 0 \end{aligned}$$

3. Essere collegati è un predicato commutativo:

$$\forall t_1, t_2 \quad \text{Collegati}(t_1, t_2) \Leftrightarrow \text{Collegati}(t_2, t_1)$$

4. L'output di una porta OR vale 1 se e solo se uno dei suoi input è 1:

$$\begin{aligned} \forall g \quad & \text{Tipo}(g) = \text{OR} \Rightarrow \\ & \text{Segnale}(\text{Out}(1, g)) = 1 \Leftrightarrow \exists n \quad \text{Segnale}(\text{In}(n, g)) = 1 \end{aligned}$$

5. L'output di una porta AND vale 0 se e solo se uno dei suoi input è 0:

$$\begin{aligned} \forall g \quad & \text{Tipo}(g) = \text{AND} \Rightarrow \\ & \text{Segnale}(\text{Out}(1, g)) = 0 \Leftrightarrow \exists n \quad \text{Segnale}(\text{In}(n, g)) = 0 \end{aligned}$$

6. L'output di una porta XOR vale 1 se e solo se i suoi input sono diversi:

$$\begin{aligned} \forall g \quad & \text{Tipo}(g) = \text{XOR} \Rightarrow \\ & \text{Segnale}(\text{Out}(1, g)) = 1 \Leftrightarrow \text{Segnale}(\text{In}(1, g)) \neq \text{Segnale}(\text{In}(2, g)) \end{aligned}$$

7. L'output di una porta NOT è diverso dal suo input:

$$\forall g \quad (\text{Tipo}(g) = \text{NOT}) \Rightarrow \text{Segnale}(\text{Out}(1, g)) \neq \text{Segnale}(\text{In}(1, g))$$

Codificare un'istanza specifica del problema

La seguente descrizione codifica il circuito mostrato nella Figura 8.4 come circuito C_1 . Per prima cosa indichiamo il tipo delle porte:

$$\begin{array}{ll} \text{Tipo}(X_1) = \text{XOR} & \text{Tipo}(X_2) = \text{XOR} \\ \text{Tipo}(A_1) = \text{AND} & \text{Tipo}(A_2) = \text{AND} \\ \text{Tipo}(O_1) = \text{OR} & \end{array}$$

Quindi specifichiamo i collegamenti tra le porte:

$$\begin{array}{ll} \text{Collegati}(\text{Out}(1, X_1), \text{In}(1, X_2)) & \text{Collegati}(\text{In}(1, C_1), \text{In}(1, X_1)) \\ \text{Collegati}(\text{Out}(1, X_1), \text{In}(2, A_2)) & \text{Collegati}(\text{In}(1, C_1), \text{In}(1, A_1)) \\ \text{Collegati}(\text{Out}(1, A_2), \text{In}(1, O_1)) & \text{Collegati}(\text{In}(2, C_1), \text{In}(2, X_1)) \\ \text{Collegati}(\text{Out}(1, A_1), \text{In}(2, O_1)) & \text{Collegati}(\text{In}(2, C_1), \text{In}(2, A_1)) \\ \text{Collegati}(\text{Out}(1, X_2), \text{Out}(1, C_1)) & \text{Collegati}(\text{In}(3, C_1), \text{In}(2, X_2)) \\ \text{Collegati}(\text{Out}(1, O_1), \text{Out}(2, C_1)) & \text{Collegati}(\text{In}(3, C_1), \text{In}(1, A_2)) . \end{array}$$

Interrogare la procedura di inferenza

Quali combinazioni di input faranno sì che il primo output di C_1 (il bit di somma) valga 0 e il secondo output di C_1 (il bit di riporto) valga 1?

$$\exists i_1, i_2, i_3 \quad \text{Segnale}(\text{In}(1, C_1)) = i_1 \wedge \text{Segnale}(\text{In}(2, C_1)) = i_2 \wedge \text{Segnale}(\text{In}(3, C_1)) = i_3 \\ \wedge \text{Segnale}(\text{Out}(1, C_1)) = 0 \wedge \text{Segnale}(\text{Out}(2, C_1)) = 1 .$$

Le risposte saranno costituite da sostituzioni delle variabili i_1 , i_2 e i_3 tali che la formula risultante sia implicata dalla base di conoscenza. Ce ne sono tre:

$$\{i_1/1, i_2/1, i_3/0\} \quad \{i_1/1, i_2/0, i_3/1\} \quad \{i_1/0, i_2/1, i_3/1\} .$$

Quali sono i possibili insiemi di valori di tutti i morsetti del circuito sommatore?

$$\exists i_1, i_2, i_3, o_1, o_2 \quad \text{Segnale}(\text{In}(1, C_1)) = i_1 \wedge \text{Segnale}(\text{In}(2, C_1)) = i_2 \\ \wedge \text{Segnale}(\text{In}(3, C_1)) = i_3 \wedge \text{Segnale}(\text{Out}(1, C_1)) = o_1 \wedge \text{Segnale}(\text{Out}(2, C_1)) = o_2 .$$

Quest'ultima interrogazione restituirà una tabella input-output completa per il dispositivo, che può essere usata per verificare che sommi effettivamente in modo corretto i suoi input. Questo è un semplice esempio di verifica dei circuiti. Possiamo usare la definizione del circuito per costruire sistemi digitali più complessi, per i quali sarà possibile effettuare lo stesso tipo di verifica (v. Esercizio 8.17). Questo tipo di sviluppo strutturato, che si appoggia su una base di conoscenza, è applicabile a una varietà di domini in cui è possibile esprimere i concetti più complessi partendo da quelli più semplici.

verifica dei circuiti

Fare il debugging della base di conoscenza

Possiamo perturbare la base di conoscenza in vari modi per vedere quale tipo di errori emerge nel suo comportamento. Per esempio, supponiamo di omettere l'asserzione che $1 \neq 0$.¹² Improvvisamente il sistema non potrà calcolare alcun output del circuito, eccetto che per i casi di input 000 e 110. Per localizzare il problema con precisione possiamo verificare gli output di ogni porta, chiedendo per esempio

$$\exists i_1, i_2 \text{ o } \text{Segnale}(\text{In}(1, C_1)) = i_1 \wedge \text{Segnale}(\text{In}(2, C_1)) = i_2 \wedge \text{Segnale}(\text{Out}(1, X_1))$$

il che rivelerà che l'output di X_1 non è noto nel caso degli input 10 e 01. A questo punto basterà considerare gli assiomi delle porte XOR, così come si applicano a X_1 :

$$\text{Segnale}(\text{Out}(1, X_1)) = 1 \Leftrightarrow \text{Segnale}(\text{In}(1, X_1)) \neq \text{Segnale}(\text{In}(2, X_1)).$$

Se gli input valgono, poniamo, 1 e 0, questo si riduce a

$$\text{Segnale}(\text{Out}(1, X_1)) = 1 \Leftrightarrow 1 \neq 0.$$

Ora il problema emerge in tutta evidenza: il sistema non è in grado di inferire che $\text{Segnale}(\text{Out}(1, X_1)) = 1$, così occorre dirgli che $1 \neq 0$.

8.5 Riepilogo

In questo capitolo abbiamo introdotto la logica del primo ordine, un linguaggio di rappresentazione molto più potente del calcolo proposizionale. I punti più importanti sono i seguenti.

- ◆ I linguaggi di rappresentazione della conoscenza dovrebbero essere dichiarativi, compositionali, espressivi, indipendenti dal contesto e non ambigui.
- ◆ Le logiche differiscono nei loro impegni ontologici e impegni epistemologici.
- ◆ Mentre il calcolo proposizionale si impegna solo sull'esistenza dei fatti, l'impegno della logica del primo ordine coinvolge l'esistenza di oggetti e relazioni e guadagna quindi potere espressivo.
- ◆ Un **mondo possibile** o **modello** per la logica del primo ordine è definito da un insieme di oggetti, dalle relazioni tra di essi e dalle funzioni che possono esservi applicate.

¹² Questo tipo di omissione è abbastanza frequente, perché gli esseri umani tipicamente danno per scontato che nomi diversi si riferiscano a oggetti differenti. Lo stesso faranno anche i sistemi di programmazione logica, che descriveremo nel prossimo capitolo.

- ◆ I simboli di costante danno un nome agli oggetti, i simboli di predicato alle relazioni, i simboli di funzione alle funzioni. Un'interpretazione specifica una corrispondenza tra i simboli e il modello. I termini complessi applicano simboli di funzione ad altri termini per dare un nome a un oggetto. Data un'interpretazione e un modello, si può determinare il valore di verità di una formula.
- ◆ Una formula atomica consiste in un predicato applicato a uno o più termini; la formula è vera quando la relazione indicata dal predicato è verificata tra gli oggetti a cui fanno riferimento i termini. Le formule complesse utilizzano i connettivi logici proprio come accade nel calcolo proposizionale, mentre le formule quantificate permettono di esprimere regole generali.
- ◆ Sviluppare una base di conoscenza in logica del primo ordine richiede un attento processo di analisi del dominio, scelta di un vocabolario e codifica degli assiomi necessari a supportare le inferenze desiderate.

Note storiche e bibliografiche

Sebbene già la logica di Aristotele si occupasse di generalizzazioni sugli oggetti, la nascita della logica del primo ordine si può far risalire all'introduzione dei quantificatori da parte di Gottlob Frege (1879) nel suo *Begriffsschrift* ("Scrittura dei concetti" o "Notazione concettuale"). La capacità di Frege di nidificare i quantificatori rappresentò un grande passo avanti, ma la sua notazione (di cui appare un esempio sul frontespizio di questo libro) era decisamente scomoda. La notazione usata oggi è dovuta principalmente a Giuseppe Peano (1889), ma la sua semantica è virtualmente identica a quella originale di Frege. In modo alquanto sorprendente, gli assiomi di Peano sono dovuti in gran parte a Grassmann (1861) e Dedekind (1888).

Una delle maggiori barriere allo sviluppo della logica del primo ordine è stata l'eccessiva attenzione dei ricercatori verso i predicati unari, con la conseguente esclusione di quelli ad aritma superiore. Questa vera e propria fissazione sui predicati unari ha rappresentato una costante nei sistemi logici da Aristotele fino a Boole incluso. Il primo a trattare in modo sistematico le relazioni è stato Augustus De Morgan (1864), che ha fornito il seguente esempio per mostrare il tipo di inferenze che non potevano essere gestite dalla logica di Aristotele: "tutti i cavalli sono animali; quindi la testa di un cavallo è la testa di un animale". Questa inferenza non è alla portata della logica aristotelica, perché ogni regola valida in grado di supportarla deve prima analizzare la formula con il predicato binario " x è la testa di y ". La logica delle relazioni è stata studiata approfonditamente da Charles Sanders Peirce (1870), che arrivò a sviluppare la logica del primo ordine indipendentemente da Frege, sebbene un po' più tardi (Peirce, 1883).

Leopold Löwenheim (1915) fornì un trattamento sistematico della teoria dei modelli per la logica del primo ordine: nel suo lavoro veniva trattato anche il simbolo di uguaglianza come parte integrante della logica. I risultati di Löwenheim furono estesi da Thoralf Skolem (1920). Alfred Tarski (1935, 1956), utilizzando la teoria degli insiemi, fornì una definizione esplicita di verità e di soddisfacimento in base alla teoria dei modelli.

McCarthy (1958) è stato il più importante responsabile dell'adozione della logica del primo ordine come strumento per la costruzione di sistemi di intelligenza artificiale. In seguito, l'applicazione della logica all'AI progredì significativamente con lo sviluppo da parte di Robinson (1965) della risoluzione, una procedura inferenziale completa che descriveremo nel Capitolo 9. L'approccio logicista attecchì a Stanford: Cordell Green (1969a, 1969b) sviluppò un sistema di ragionamento del primo ordine, QA3, la qual cosa portò ai primi tentativi di costruzione di un robot logico presso lo SRI (Fikes e Nilsson, 1971). La logica del primo ordine fu applicata da Zohar Manna e Richard Waldinger (1971) al ragionamento sui programmi e più tardi, grazie a Michael Genesereth (1984), sui circuiti. In Europa, la programmazione logica (una forma ristretta di ragionamento del primo ordine) conobbe uno sviluppo applicato all'analisi del linguaggio (Colmerauer et al., 1973) e ai sistemi dichiarativi in generale (Kowalski, 1974). La logica computazionale si guadagnò uno spazio importante anche a Edinburgo attraverso il progetto LCF (Logic for Computable Functions) (Gordon et al., 1979). Tutti questi sviluppi saranno esaminati ulteriormente nei Capitoli 9 e 10.

Ci sono diversi buoni testi introduttivi dedicati alla logica del primo ordine: uno dei più leggibili è di Quine (1982). Enderton (1972) offre una prospettiva orientata in senso matematico, mentre un trattamento molto formale della logica del primo ordine e di molti altri argomenti avanzati è fornito da Bell e Machover (1977). Manna e Waldinger (1985) hanno scritto un'introduzione di facile lettura alla logica dal punto di vista informatico. Gallier (1986) fornisce un'esposizione matematica estremamente rigorosa della logica del primo ordine e una grande mole di materiale riguardante il suo uso nel ragionamento automatico. *Logical Foundations of Artificial Intelligence* (Genesereth e Nilsson, 1987) rappresenta sia una solida introduzione alla logica che un primo trattamento sistematico degli agenti logici con percezioni e azioni.

Esercizi

- 8.1 Un base di conoscenza logica rappresenta il mondo per mezzo di un insieme di formule privo di una struttura esplicita. Una rappresentazione analogica, invece, ha una struttura fisica che corrisponde direttamente a quella dell'oggetto rappresentato. Per esempio, la mappa stradale di una re-

gione può essere considerata una rappresentazione analogica della sua configurazione fisica, dato che la sua struttura bidimensionale corrisponde alla superficie piana del terreno.

- a. Fornite cinque esempi di *simboli* nel linguaggio delle mappe.
 - b. Una formula *esplicita* è scritta fisicamente dal creatore della rappresentazione, mentre una *implicita* scaturisce da altre formule esplicite grazie alle caratteristiche della rappresentazione analogica. Fornite tre esempi di formule *implicite* e tre *esplicite* nel linguaggio delle mappe.
 - c. Fornite tre esempi di fatti riguardanti la struttura fisica della vostra regione che non possono essere rappresentati nel linguaggio delle mappe.
 - d. Fornite due esempi di fatti che sono molto più facili da esprimere con il linguaggio delle mappe che con la logica del primo ordine.
 - e. Fornite altri due esempi di rappresentazioni analogiche utili. Quali sono i vantaggi e gli svantaggi di ognuno di questi linguaggi?
- 8.2 Considerate una base di conoscenza che contenga solo due formule: $P(a)$ e $P(b)$. Questa KB implica che $\forall x P(x)$? Giustificate la vostra risposta facendo riferimento ai modelli.
- 8.3 La formula $\exists x, y \ x = y$ è valida? Spiegate.
- 8.4 Scrivete una formula logica tale che ogni modello in cui tale formula è vera contenga esattamente un oggetto.
- 8.5 Considerate un vocabolario che contiene c simboli di costante, p_k simboli di predicato di ogni arità k e f_k simboli di funzione di ogni arità k , con $1 \leq k \leq A$. Le dimensioni del dominio siano fissate a D . Per ogni possibile combinazione interpretazione-modello, ogni simbolo di predicato o di funzione corrisponde a una relazione o funzione della stessa arità. Potete presumere che le funzioni del modello permettano che alcune tuple di input non abbiano alcun valore (ovvero che il valore di quella funzione corrisponda all'oggetto invisibile). Scrivete una formula per calcolare le possibili combinazioni interpretazione-modello per un dominio con D elementi, senza preoccuparvi di eliminare le combinazioni ridondanti.
- 8.6 Rappresentate le seguenti formule in logica del primo ordine, usando un vocabolario corretto (che dovete definire).
- a. Alcuni studenti hanno scelto di studiare francese nella primavera del 2001.
 - b. Tutti gli studenti che studiano francese passano l'esame.
 - c. Solo uno studente ha scelto di studiare greco nella primavera del 2001.
 - d. Il voto migliore all'esame di greco è sempre superiore a quello migliore in francese.
 - e. Ogni persona che acquista una polizza assicurativa è previdente.

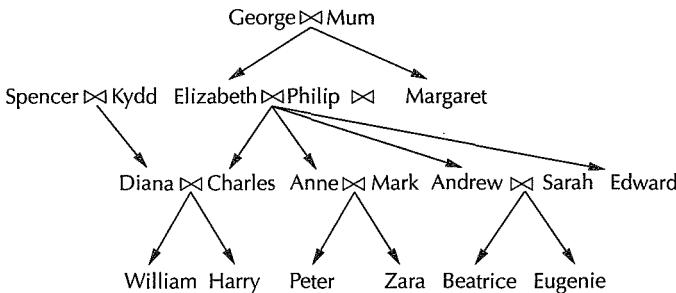
$\forall p$

$Person(p) \Rightarrow Personality(p)$

$$\forall \exists \forall \exists \forall \exists \text{ Person}(x) = \exists y \text{ August(y, x)}$$

- f. Nessuna persona acquista una polizza costosa.
- g. Esiste un agente che vende polizze solo a persone che non sono assicurate.
- h. Esiste un barbiere che rade tutte le persone della città che non si radono da sole. $\exists b \forall x \text{ Barber}(b, x)$
- i. Una persona nata nel Regno Unito, i cui genitori sono entrambi cittadini del Regno Unito o residenti nel Regno Unito, è cittadina del Regno Unito per nascita.
- j. Una persona nata al di fuori del Regno Unito, ma che ha almeno un genitore cittadino del Regno Unito, è cittadina del regno Unito per discendenza.
- k. I politici possono ingannare qualche persona tutto il tempo, o tutte le persone per qualche tempo, ma non possono ingannare tutte le persone tutto il tempo.
- 8.7 Rappresentate la formula "tutti i tedeschi parlano lo stesso linguaggio" per mezzo del calcolo dei predicati. Usate $Parla(x, l)$ per indicare che la persona x parla il linguaggio l .
- 8.8 Quale assioma è necessario per inferire il fatto $Femmina(Laura)$ dati i fatti $Maschio(Jim)$ e $Coniuge(Jim, Laura)$?
- 8.9 Scrivete un insieme generale di fatti e di assiomi per rappresentare l'affermazione "Wellington seppe della morte di Napoleone" e per rispondere correttamente alla domanda "Napoleone seppe della morte di Wellington?".
- 8.10 Riscrivete i fatti riguardanti il mondo del wumpus del Paragrafo 7.5 in logica del primo ordine anziché proposizionale. Quanto è più compatta questa nuova versione?
- 8.11 Scrivete gli assiomi che descrivono i predicati *Nipote*, *Bisnonno*, *Fratello*, *Sorella*, *Figlia*, *Figlio*, *Zia*, *Zio*, *Cognato*, *Cognata* e *PrimoCugino*. Cercate la definizione precisa di "cugino di grado m , rimosso n volte" e scrivetene la definizione in logica del primo ordine.
Quindi scrivete i fatti mostrati nell'albero genealogico della Figura 8.5. Scegliete un sistema di ragionamento logico adatto e ditegli (attraverso azioni TELL) tutte le formule che avete scritto, e infine chiedetegli (ASK) di chi è nonna Elizabeth, chi sono i cognati di Diana e i bisnonni di Zara.
- 8.12 Scrivete una formula che asserisce che $+$ è una funzione commutativa. La vostra formula è implicata dagli assiomi di Peano? Se è così, spiegate il perché; in caso contrario fornite un modello in cui gli assiomi sono veri e la vostra formula falsa.
- 8.13 Spieghate cosa c'è di sbagliato nella seguente proposta di definizione del predicato di appartenenza a un insieme \in :

$$\begin{aligned} \forall x, s \quad x \in \{x|s\} \\ \forall x, s \quad x \in s \Rightarrow \forall y \quad x \in \{y|s\}. \end{aligned}$$

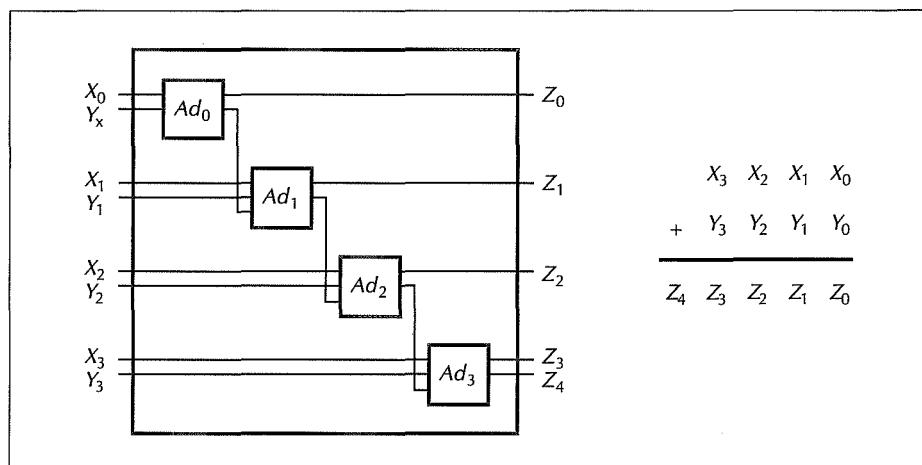
**Figura 8.5**

Un tipico albero genealogico.
Il simbolo "▷" collega le coppie di coniugi e le frecce puntano ai figli.

- 8.14 Usando come esempio quelli relativi agli insiemi, scrivete degli assiomi per il dominio delle liste, includendo tutte le costanti, le funzioni e i predicati menzionati nel capitolo.
- 8.15 Spiegate cosa c'è di sbagliato nella seguente proposta di definizione di stanze adiacenti nel mondo del wumpus:
- $$\forall x, y \text{ Adiacente}([x, y], [x + 1, y]) \wedge \text{Adiacente}([x, y], [x, y + 1]).$$
- 8.16 Scrivete gli assiomi necessari per ragionare sulla posizione del wumpus, usando un simbolo di costante *Wumpus* e un binario predicato *In(Wumpus, Posizione)*. Ricordate che esiste un solo wumpus.
- 8.17 Estendete il vocabolario del Paragrafo 8.4 in modo da definire l'addizione per numeri binari a n bit. Fatto questo codificate la descrizione del circuito sommatore a quattro bit della Figura 8.6 e ponete le domande necessarie per verificare che sia effettivamente corretta.
- 8.18 La rappresentazione dei circuiti fornita in questo capitolo è più dettagliata del necessario, se tutto quello che ci interessa è la loro funzionalità. Una formulazione più semplice descrive ogni porta logica o circuito a m input e n output mediante un predicato con $m + n$ argomenti, tale che il predicato sia vero solo quando input e output sono consistenti. Ad esempio, le porte NOT sono descritte dal predicato binario $NOT(i, o)$, per cui sono fatti noti $NOT(0, 1)$ e $NOT(1, 0)$. Una composizione di porte si definisce mediante congiunzioni di predicati in cui variabili condivise indicano connessioni dirette. Ad esempio, un circuito NAND può essere composto da AND e NOT:
- $$\forall i_1, i_2, o_a, o \text{ NAND}(i_1, i_2, o) \Leftrightarrow AND(i_1, i_2, o_a) \wedge NOT(o_a, o).$$



Figura 8.6
Un sommatore a quattro bit.



Usando questa rappresentazione, definite il sommatore a un bit della Figura 8.4 e quello a quattro bit della Figura 8.6, e spiegate quali interrogazioni usereste per verificare i progetti. Quali categorie di query supportate dalla rappresentazione che abbiamo visto nel Paragrafo 8.4 *non* sono supportate da questa?

- 8.19 Procuratevi un modulo di richiesta di passaporto per il vostro paese, identificate le regole che determinano l'elgibilità per la richiesta e traducetele in logica del primo ordine, seguendo i passi descritti nel Paragrafo 8.4.

Capitolo 9

L'inferenza nella logica del primo ordine

Nel quale definiamo procedure efficienti per rispondere a domande poste con il linguaggio della logica del primo ordine.

Nel Capitolo 7 abbiamo definito il concetto di inferenza e abbiamo mostrato come sia possibile effettuare inferenze corrette e complete nel calcolo proposizionale. In questo capitolo estenderemo tali risultati per ottenere algoritmi che possono dare una risposta (se questa esiste) a qualsiasi domanda espressa in logica del primo ordine. Si tratta di un risultato significativo, perché con il linguaggio della logica del primo ordine e un po' di lavoro si può esprimere praticamente qualsiasi cosa.

Il Paragrafo 9.1 introduce le regole di inferenza per i quantificatori e mostra come si può ridurre l'inferenza del primo ordine a quella proposizionale, sebbene questo comporti grandi costi. Il Paragrafo 9.2 presenta l'idea di unificazione, mostrando come possa essere usata per costruire regole di inferenza che operano direttamente sulle formule del primo ordine. Discuteremo poi tre grandi famiglie di algoritmi di inferenza del primo ordine: la concatenazione in avanti e le sue applicazioni ai databases deduttivi e ai sistemi di produzioni è trattata nel Paragrafo 9.3; la concatenazione all'indietro e i sistemi di programmazione logica sono sviluppati nel Paragrafo 9.4; infine, i sistemi di dimostrazione di teoremi basati sulla risoluzione sono descritti nel Paragrafo 9.5. In generale si cerca sempre di utilizzare il metodo più efficiente tra quelli in grado di gestire i fatti e gli assiomi espressi. Il ragionamento applicato alle formule del primo ordine più generali, che utilizza la risoluzione, è solitamente meno efficiente di quello che opera su clausole definite mediante la concatenazione in avanti o all'indietro.

9.1 Inferenza proposizionale e inferenza del primo ordine

*Modus Ponens
risoluzione*

Questo paragrafo e il successivo presentano le idee alla base dei moderni sistemi di inferenza logica. Cominceremo con alcune semplici regole che si possono applicare alle formule quantificate per ottenere formule prive di quantificatori. Questo porta naturalmente a supporre che l'inferenza del primo ordine possa essere effettuata convertendo l'intera base di conoscenza in logica proposizionale per poi usare l'inferenza proposizionale, che già conosciamo. Nel paragrafo seguente esamineremo una "scorciatoia" che ci porterà a formulare metodi di inferenza in grado di manipolare direttamente le formule del primo ordine.

Regole di inferenza per i quantificatori

Cominciamo con i quantificatori universali. Supponiamo che la nostra base di conoscenza contenga il classico detto popolare che tutti i felavidi sono malvagi:

$$\forall x \quad Re(x) \wedge Avido(x) \Rightarrow Malvagio(x).$$

Sembra plausibile inferire tutte le seguenti formule:

$$Re(Giovanni) \wedge Avido(Giovanni) \Rightarrow Malvagio(Giovanni).$$

$$Re(Riccardo) \wedge Avido(Riccardo) \Rightarrow Malvagio(Riccardo).$$

$$Re(Padre(Giovanni)) \wedge Avido(Padre(Giovanni)) \Rightarrow Malvagio(Padre(Giovanni)).$$

⋮

Istanziazione universale

la regola di istanziazione universale (in breve UI, dall'acronimo inglese) afferma che possiamo inferire tutte le formule ottenute sostituendo un termine ground (cioè privo di variabili) alla variabile.¹ Per scrivere formalmente la regola di inferenza useremo la nozione di sostituzione introdotta nel Paragrafo 8.3. Indichiamo con SUBST(θ , α) il risultato dell'applicazione della sostituzione θ alla formula α . La regola allora si scrive

$$\frac{\forall v \quad \alpha}{\text{SUBST}(\{v/g\}, \alpha)} \quad \text{Istanziamento}$$

per ogni variabile v e termine ground g . Le tre formule qui sopra, ad esempio, sono ottenute con le sostituzioni $\{x/Giovanni\}$, $\{x/Riccardo\}$ e $\{x/Padre(Giovanni)\}$.

¹ Non confondete queste sostituzioni con le interpretazioni estese usate per definire la semantica dei quantificatori. La sostituzione rimpiazza una variabile con un termine (sintattico) per produrre una nuova formula, laddove un'interpretazione mette in corrispondenza variabili e oggetti del dominio.

La regola di istanziazione esistenziale è leggermente più complicata. Per ogni formula α , variabile v e simbolo di costante k che non appare da nessun'altra parte nella base di conoscenza,

$$\frac{\exists v \ \alpha}{\text{SUBST } (\{v/k\}, \alpha)}$$

Per esempio, dalla formula

$$\exists x \text{ Corona}(x) \wedge \text{SullaTesta}(x, \text{Giovanni})$$

possiamo inferire la formula

$$\text{Corona}(C_1) \wedge \text{SullaTesta}(C_1, \text{Giovanni})$$

A patto che C_1 non appaia in alcun altro punto della base di conoscenza. Sostanzialmente la formula esistenziale afferma che esiste un qualche oggetto che soddisfa una condizione e il processo di istanziazione si limita a dargli un nome; naturalmente quel nome non deve già appartenere a un altro oggetto. La matematica ci offre un altro esempio: supponiamo di scoprire un numero un po' più grande di 2,71828 che soddisfa l'equazione $d(x^y)/dy = x^y$ per x . Possiamo dare a questo numero un nome come e , ma sarebbe un errore assegnargli quello di un oggetto preesistente, come π . In logica, il nuovo nome viene chiamato costante di Skolem. L'istanziazione esistenziale è un caso speciale di un processo più generale chiamato skolemizzazione, che tratteremo nel Paragrafo 9.5.

Oltre a essere un po' più complicata dell'istanziazione universale, quella esistenziale ha un ruolo leggermente diverso nell'inferenza. Laddove la UI può essere applicata più volte per produrre conseguenze diverse, l'istanziazione esistenziale può essere applicata solo una volta, dopodiché la formula quantificata esistenzialmente può essere scartata. Ad esempio, una volta che abbiamo aggiunto la formula $\text{Uccide}(\text{Assassino}, \text{Vittima})$ non abbiamo più bisogno della formula $\exists x \text{ Uccide}(x, \text{Vittima})$. Formalmente la nuova base di conoscenza non è logicamente equivalente alla precedente, ma si può dimostrare che è inferenzialmente equivalente nel senso che è soddisfacibile esattamente tutte le volte in cui lo è la base di conoscenza originale.

Riduzione all'inferenza proposizionale

Avendo a disposizione le regole che permettono di inferire formule non quantificate da quelle quantificate, diventa possibile ridurre l'inferenza del primo ordine a quella proposizionale. In questo paragrafo delineeremo i concetti principali; i dettagli saranno forniti nel Paragrafo 9.5.

Prima di tutto occorre dire che, proprio come una formula quantificata esistenzialmente può essere rimpiazzata da una sua istanza, così una formula quantificata universalmente può essere sostituita dall'insieme di tutte le possibili istanze. Supponete ad esempio che la base di conoscenza contenga solo le formule:

istanziazione esistenziale

$\exists v \ \alpha$

$$\begin{aligned}
 & \forall x Re(x) \wedge Avido(x) \Rightarrow Malvagio(x) \\
 & Re(Giovanni) \\
 & Avido(Giovanni) \\
 & Fratello(Riccardo, Giovanni) .
 \end{aligned} \tag{9.1}$$

Applicando la UI alla prima formula con tutte le possibili sostituzioni di termini ground del vocabolario, in questo caso $\{x/Giovanni\}$ e $\{x/Riccardo\}$, otteniamo

$$\begin{aligned}
 & Re(Giovanni) \wedge Avido(Giovanni) \Rightarrow Malvagio(Giovanni) , \\
 & Re(Riccardo) \wedge Avido(Riccardo) \Rightarrow Malvagio(Riccardo) ,
 \end{aligned}$$

e possiamo così scartare la formula universalmente quantificata. Ora basta considerare le formule atomiche ground come $Re(Giovanni)$ o $Avido(Giovanni)$ alla stregua di simboli proposizionali per vedere che la stessa base di conoscenza è essenzialmente proposizionale. Di conseguenza possiamo applicare uno qualsiasi degli algoritmi completi proposti nel Capitolo 7 per ottenere conclusioni come $Malvagio(Giovanni)$.

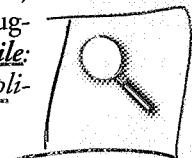
proposizionalizzazione

Questa tecnica di **proposizionalizzazione** può essere applicata in modo assolutamente generale, come dimostriamo nel Paragrafo 9.5; questo significa che ogni base di conoscenza del primo ordine e ogni interrogazione possono essere proposizionalizzati in modo tale da preservare le implicazioni. Ne consegue che possiamo dire di possedere una procedura decisionale completa per risolvere l'implicazione... o forse no. C'è un problema: quando la base di conoscenza include un simbolo di funzione, l'insieme di possibili sostituzioni di termini ground diventa infinito! Per esempio, se la base di conoscenza contiene il simbolo Padre, si possono costruire infiniti termini nidificati come $Padre(Padre(Padre(Giovanni)))$. I nostri algoritmi proposizionali avranno una certa difficoltà a gestire un insieme di formule infinitamente grande.

Fortunatamente, il famoso teorema di Jacques Herbrand (1930) ci assicura che se la formula è implicata dalla base di conoscenza originale, quella del primo ordine, allora ci sarà una dimostrazione che richiede solo un sottoinsieme finito della base proposizionalizzata. Dato che in qualsiasi sottoinsieme finito la profondità di annidamento dei termini deve essere limitata, possiamo generare prima tutte le istanziazioni con simboli di costante (Riccardo e Giovanni), poi tutti i termini di profondità 1 (Padre(Riccardo) e Padre(Giovanni)), quindi tutti i termini di profondità 2, e così via finché non saremo in grado di costruire la dimostrazione proposizionale della formula implicata.

Il metodo che abbiamo delineato per effettuare inferenze in logica del primo ordine attraverso la proposizionalizzazione è completo: ogni formula implicata può essere dimostrata. Questo è un risultato notevole, dato che lo spazio di tutti i possibili modelli è infinito. D'altra parte, finché la dimostrazione è terminata non sappiamo se la formula è effettivamente implicata! Che cosa succede quando non lo è? Possiamo determinarlo? Nella logica del primo ordine, questo risulta impossibile; il nostro procedimento continuerà a generare termini sempre più profondi,

ma non sapremo mai se ci troviamo incastrati in un ciclo senza speranza o se la dimostrazione sta per terminare. Questo problema ricorda molto da vicino la terminazione delle macchine di Turing: lo stesso Turing (1936) e Alonzo Church (1936) hanno dimostrato, in modi alquanto differenti, che non c'è alcun modo di sfuggirvi. Il problema dell'implicazione, per la logica del primo ordine, è semidecidibile: questo significa che gli algoritmi possono determinare se una qualsiasi formula è implacata, ma nessun algoritmo potrà mai dimostrare che una formula non lo è.



9.2 Unificazione e lifting

Nel paragrafo precedente abbiamo descritto l'inferenza nella logica del primo ordine com'era compresa fino all'inizio degli anni '60. Il lettore attento avrà notato che la proposizionalizzazione è piuttosto inefficiente. In effetti, data l'interrogazione *Malvagio(x)* e la base di conoscenza dell'Equazione (9.1), sembra abbastanza inutile generare formule come *Re(Riccardo) \wedge Avido(Riccardo) \Rightarrow Malvagio(Riccardo)*. A un essere umano, del resto, l'inferenza *Malvagio(Giovanni)* dalle formule

$$\begin{aligned} \forall x \text{Re}(x) \wedge \text{Avido}(x) &\Rightarrow \text{Malvagio}(x) \\ \text{Re}(\text{Giovanni}) \\ \text{Avido}(\text{Giovanni}) \end{aligned}$$

sembra assolutamente ovvia. Ora vedremo come tale inferenza può essere resa ovvia anche per un computer.

Una regola di inferenza del primo ordine

Per inferire che Giovanni è malvagio si procede così: si trova un qualche x tale che ~~è re e è avido~~, dopodiché si inferisce che questo x è malvagio. Più generalmente, se esiste una sostituzione θ che rende la premessa dell'implicazione identica a una formula già presente nella base di conoscenza, allora dopo aver applicato θ possiamo asserirne anche la conclusione. In questo caso, la sostituzione $\{x/\text{Giovanni}\}$ raggiunge lo scopo.

Possiamo far sì che il passo di inferenza svolga ulteriore lavoro: supponiamo che invece di limitarci a dire che *Avido(Giovanni)*, sappiamo che *tutti* sono avidi:

$$\forall y \text{Avido}(y). \tag{9.2}$$

In questo caso vorremmo continuare a essere in grado di concludere che *Malvagio(Giovanni)*, perché sappiamo che Giovanni è un re (fatto noto) e che è avido (perché lo sono tutti). Per far questo dobbiamo trovare una sostituzione sia per le variabili nell'implicazione che per quelle nelle altre formule coinvolte. In questo caso, applicare la sostituzione $\{x/\text{Giovanni}, y/\text{Giovanni}\}$ alle premesse dell'implicazione *Re(x) \wedge Avido(x)* nonché alle formule della base di conoscenza *Re(Giovanni)* e *Avido(y)* le renderà identiche. Di conseguenza potremo inferire la conclusione dell'implicazione.

Modus Ponens
generalizzato

Questo processo di inferenza può essere espresso da una singola regola chiamata **Modus Ponens generalizzato**: per le formule atomiche p_i, p'_i , e q , ove ci sia una sostituzione θ tale che $\text{SUBST}(\theta, p'_i) = \text{SUBST}(\theta, p_i)$ per tutti gli i ,

$$\frac{p'_1, p'_2, p'_n, (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}.$$

Questa regola ha $n + 1$ premesse: le n formule atomiche p'_i e la singola implicazione. La conclusione è il risultato dell'applicazione della sostituzione θ alla conseguenza q . Nel nostro esempio:

$$\begin{aligned} p'_1 &\text{ è } \text{Re}(\text{Giovanni}) \\ p'_2 &\text{ è } \text{Avido}(y) \\ \theta &\text{ è } \{x/\text{Giovanni}, y/\text{Giovanni}\} \\ \text{SUBST}(\theta, q) &\text{ è } \text{Malvagio}(\text{Giovanni}) . \end{aligned}$$

p'_1 è $\text{Re}(x)$	\rightarrow
p'_2 è $\text{Avido}(x)$	\rightarrow
q è $\text{Malvagio}(x)$	\rightarrow

Re(x)
 Avido(x)
 Malvagio(x)

È facile dimostrare che il Modus Ponens generalizzato è una regola di inferenza corretta. Prima di tutto osserviamo che per ogni formula p (le cui variabili si presumono universalmente quantificate) e per ogni sostituzione θ ,

$$p \models \text{SUBST}(\theta, p).$$

Questo vale per la stessa ragione per cui vale la regola di istanziazione universale, e in particolare per una θ che soddisfa le condizioni del Modus Ponens generalizzato. Così, da p'_1, \dots, p'_n possiamo inferire

$$\text{SUBST}(\theta, p'_1) \wedge \dots \wedge \text{SUBST}(\theta, p'_n)$$

E dall'implicazione $p_1 \wedge \dots \wedge p_n \Rightarrow q$ possiamo inferire

$$\text{SUBST}(\theta, p_1) \wedge \dots \wedge \text{SUBST}(\theta, p_n) \Rightarrow \text{SUBST}(\theta, q).$$

Ora, nel Modus Ponens generalizzato θ è definita in modo tale che $\text{SUBST}(\theta, p'_i) = \text{SUBST}(\theta, p_i)$ per tutti gli i ; ne consegue che la prima di queste due formule corrisponde esattamente alla premessa della seconda. Quindi $\text{SUBST}(\theta, q)$ segue per Modus Ponens.

Il Modus Ponens generalizzato è ottenuto dal Modus Ponens che già conosciamo attraverso un processo noto come **lifting** (letteralmente "sollevamento"). Il termine deriva dal fatto che abbiamo "sollevato" il **Modus Ponens**, portandolo dalla logica proposizionale a quella del primo ordine. Nel resto del capitolo vedremo che attraverso il lifting si possono sviluppare versioni per la logica del primo ordine della concatenazione in avanti, di quella all'indietro e dell'algoritmo di risoluzione, concetti che abbiamo già presentato nel Capitolo 7. Il vantaggio principale dell'uso di regole di inferenza "sollevate", rispetto alla proposizionalizzazione, sta nel fatto che le prime effettuano solo le sostituzioni effettivamente necessarie per portare avanti le inferenze. C'è un punto che potrebbe generare confusione, in quanto il Modus Ponens generalizzato è effettivamente meno generale del Modus

lifting

Ponens che abbiamo definito a pag. 272: quest'ultimo infatti permette che nella parte sinistra dell'implicazione vi sia una formula qualsiasi, mentre il Modus Ponens generalizzato richiede che essa sia espressa in un formato speciale. La sua "generalità" si riferisce al fatto che permette un numero arbitrario di P_i .

Unificazione

Le regole di inferenza ottenute attraverso il lifting richiedono di trovare sostituzioni che rendono identiche espressioni logiche diverse. Questo processo è chiamato **unificazione** ed è un componente chiave di tutti gli algoritmi di inferenza del primo ordine. L'algoritmo UNIFY prende due formule e, se esiste, restituisce un loro unificatore:

$$\text{UNIFY}(p, q) = \theta \text{ ove } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

Vediamo qualche esempio di come si dovrebbe comportare UNIFY. Supponiamo di avere una query *Conosce(Giovanni, x)*: quali persone conosce Giovanni? Si possono trovare delle risposte a quest'interrogazione cercando tutte le formule nella base di conoscenza che risultano dall'applicazione di UNIFY con *Conosce(Giovanni, x)*. Ecco il risultato dell'unificazione con quattro formule diverse che potrebbero far parte della base di conoscenza:

$$\begin{array}{ll} \text{UNIFY}(\text{Conosce}(\text{Giovanni}, x), \text{Conosce}(\text{Giovanni}, \text{Giacomina})) = \{x/\text{Giacomina}\} \\ \text{UNIFY}(\text{Conosce}(\text{Giovanni}, x), \text{Conosce}(y, \text{Guglielmo})) = \{x/\text{Guglielmo}, y/\text{Giovanni}\} \\ \text{UNIFY}(\text{Conosce}(\text{Giovanni}, x), \text{Conosce}(y, \text{Madre}(y))) = \{y/\text{Giovanni}, x/\text{Madre}(\text{Giovanni})\} \\ \text{UNIFY}(\text{Conosce}(\text{Giovanni}, x), \text{Conosce}(x, \text{Elisabetta})) = \text{fallimento}. \end{array}$$

L'ultima unificazione fallisce, perché x non può assumere contemporaneamente i valori *Giovanni* ed *Elisabetta*. Ricordate però che *Conosce(x, Elisabetta)* significa "tutti conoscono Elisabetta", per cui dovremmo essere in grado di inferire che anche Giovanni la conosce. Il problema si verifica unicamente perché le due formule usano lo stesso nome per la variabile x . Tutto ciò può essere evitato rinominando le variabili di una delle formule per evitare collisioni, un'operazione che prende il nome **standardizzazione separata** (*standardizing apart*). Potremmo ad esempio rinominare la x di *Conosce(x, Elisabetta)* in z_{17} (un nuovo nome) senza modificare il suo significato. Ora l'unificazione funzionerà:

$$\text{UNIFY}(\text{Conosce}(\text{Giovanni}, x), \text{Conosce}(z_{17}, \text{Elisabetta})) = \{x/\text{Elisabetta}, z_{17}/\text{Giovanni}\}.$$

L'Esercizio 9.7 esaminerà più a fondo questo tipo di operazione.

C'è però un'ulteriore complicazione: abbiamo detto che UNIFY dovrebbe restituire una sostituzione tale che i due argomenti appaiano identici; ma potrebbe esistere anche più di un unificatore. Ad esempio, *UNIFY(Conosce(Giovanni, x), Conosce(y, z))* potrebbe restituire $\{y/\text{Giovanni}, x/z\}$ o $\{y/\text{Giovanni}, x/\text{Giovanni}, z/\text{Giovanni}\}$. Il primo unificatore dà come risultato *Conosce(Giovanni, z)*, mentre il secondo dà *Conosce(Giovanni, Giovanni)*. Il secondo risultato potrebbe essere an-

unificazione

standardizzazione separata

unificatore più generale

controllo di occorrenza

che ottenuto dal primo con la sostituzione aggiuntiva $\{z/Giovanni\}$; in questo caso diciamo che il primo unificatore è più generale del secondo, dato che impone meno restrizioni sul valore delle variabili. Risulta che, per ogni coppia di espressioni unificabili, esiste un singolo unificatore più generale (o MGU, dall'acronimo inglese), distinto da tutti gli altri (a meno della rinominazione delle variabili). In questo caso il MGU è $\{y/Giovanni, x/z\}$.

La Figura 9.1 presenta un algoritmo per il calcolo degli MGU. Il processo è molto semplice: si esplorano simultaneamente le due espressioni “fianco a fianco” in modo ricorsivo, costruendo contemporaneamente un unificatore, e restituendo un risultato di fallimento se si incontrano nella loro struttura due punti non corrispondenti. L'algoritmo include un passo di costo computazionale elevato: quando si cerca una corrispondenza tra una variabile e un termine complesso, è necessario controllare se la variabile stessa è presente all'interno dello stesso termine; in tal caso la corrispondenza fallisce perché non è possibile costruire un unificatore consistente. Questo cosiddetto controllo di occorrenza fa sì che la complessità dell'intero algoritmo risulti quadratica nella dimensione delle espressioni da unificare. Alcuni sistemi, tra cui tutti quelli di programmazione logica, si limitano ad omettere completamente il controllo di occorrenza; di conseguenza può capitare che effettuino inferenze non corrette. Altri sistemi utilizzano algoritmi più complessi che hanno complessità lineare.

Memorizzazione e recupero di informazioni

Alla base delle funzioni TELL e ASK, usate per informare e interrogare una base di conoscenza, ci sono le funzioni primitive STORE e FETCH. STORE(s) memorizza la formula s nella base di conoscenza, FETCH(q) restituisce tutti gli unificatori tali che la query q possa unificare con una formula presente nella KB. Il problema che abbiamo proposto qui sopra, di trovare tutti i fatti che unificano con $Conosce(Giovanni, x)$, è un esempio di FETCH.

Il modo più semplice di implementare STORE e FETCH è inserire tutti i fatti presenti nella KB in una lunga lista; a quel punto, data la query q , basterebbe chiamare UNIFY(q, s) per ogni formula s presente in tale lista. Un processo simile è inefficiente ma funziona, e per comprendere il resto del capitolo non è necessario sapere altro. Ora accenneremo sommariamente alle tecniche utilizzate per rendere il recupero delle informazioni più efficiente: questa parte può essere saltata a una prima lettura.

Possiamo rendere FETCH più efficiente assicurandoci che cerchi di effettuare l'unificazione solo quando le formule assicurano che esista almeno una qualche possibilità di riuscita. Ad esempio, è del tutto inutile cercare di unificare $Conosce(Giovanni, x)$ con $Fratello(Riccardo, Giovanni)$. Possiamo evitare tali unificazioni indicizzando i fatti nella base di conoscenza. Un semplice schema chiamato indicizzazione dei predicati pone tutti i fatti $Conosce$ in un bucket (termine tecnico

indicizzando
indicizzazione dei
predicati

```

function UNIFY( $x, y, \theta$ ) returns una sostituzione che rende  $x$  e  $y$  identici
  inputs:  $x$ , una variabile, costante, lista o espressione composta
           $y$ , una variabile, costante, lista o espressione composta
           $\theta$ , la sostituzione costruita sin qui (opzionale, per default è vuota)

  if  $\theta$  = fallimento then return fallimento
  else if  $x = y$  then return  $\theta$ 
  else if VARIABILE?( $x$ ) then return UNIFY-VAR( $x, y, \theta$ )
  else if VARIABILE?( $y$ ) then return UNIFY-VAR( $y, x, \theta$ )
  else if COMPOSTA?( $x$ ) and COMPOSTA?( $y$ ) then
    return UNIFY(ARGS[ $x$ ], ARGS[ $y$ ], UNIFY(OP[ $x$ ], OP[ $y$ ],  $\theta$ ))
  else if LISTA?( $x$ ) and LISTA?( $y$ ) then
    return UNIFY(RESTO[ $x$ ], RESTO[ $y$ ], UNIFY(PRIMO[ $x$ ], PRIMO[ $y$ ],  $\theta$ ))
  else return fallimento

```

```

function UNIFY-VAR( $var, x, \theta$ ) returns una sostituzione
  inputs:  $var$ , una variabile
           $x$ , una qualsiasi espressione
           $\theta$ , la sostituzione costruita sin qui

  if  $\{var/val\} \in \theta$  then return UNIFY( $val, x, \theta$ )
  else if  $\{x/val\} \in \theta$  then return UNIFY( $var, val, \theta$ )
  else if CONTROLLA-OCCORRENZA?( $var, x$ ) then return fallimento
  else return aggiungi  $\{var/x\}$  a  $\theta$ 

```

Figura 9.1 L'algoritmo di unificazione. L'algoritmo confronta le strutture degli input, elemento per elemento. La sostituzione θ , argomento di UNIFY, è costruita via via e usata per assicurarsi che i confronti successivi siano consistenti con i legami già stabiliti. In un'espressione composta, come $F(A, B)$, la funzione OP estrae il simbolo di funzione F e la funzione ARGS estrae la lista di argomenti (A, B).

che letteralmente significa “secchio”) e tutti i fatti *Fratello* in un altro. I bucket possono essere memorizzati in una tabella di hash per assicurare un accesso efficiente.²

L'indicizzazione dei predicati è utile quando ci sono molti simboli di predicationi ma solo poche clausole per ognuno di essi. In alcune applicazioni, tuttavia, a

² Una tabella di hash è una struttura dati usata per memorizzare e recuperare informazione indirizzata da chiavi prefissate. In pratica si può considerare che una tabella di hash abbia tempi di inserimento e di lettura dei dati costanti, anche quando contiene un numero di elementi molto grande.

ogni simbolo di predicato sono associate molte clausole. Supponiamo ad esempio che il fisco voglia tener traccia di chi lavora per ogni azienda con il predicato $Dipendente(x, y)$. Questo darebbe origine a un bucket molto esteso, con milioni di datori di lavoro e decine di milioni di dipendenti, e rispondere a una semplice query come $Dipendente(x, Riccardo)$ attraverso l'indicizzazione dei prediciati richiederebbe la scansione dell'intero bucket.

Per questa particolare interrogazione, un grande aiuto sarebbe rappresentato dall'indicizzazione dei fatti mediante sia il predicato che il secondo argomento, magari usando una chiave di hash combinata. A quel punto si potrebbe semplicemente costruire la chiave dalla query ed estrarre esattamente i fatti che unificano con la query stessa. Per altri tipi di interrogazione, come $Dipendente(AIMA.org, y)$, dovremmo indicizzare i fatti combinando il predicato e il primo argomento. I fatti quindi possono essere memorizzati in base a chiavi d'accesso multiple, rendendoli così istantaneamente disponibili alle varie query con cui potrebbero unificare.

Data una formula da memorizzare, è possibile costruire indici per *tutte le possibili* query che dovranno unificare con essa. Per il fatto $Dipendente(AIMA.org, Riccardo)$, le interrogazioni sono:

$Dipendente(AIMA.org, Riccardo)$	La AIMA.org dà lavoro a Riccardo?
$Dipendente(x, Riccardo)$	Per chi lavora Riccardo?
$Dipendente(AIMA.org, y)$	A chi dà lavoro la AIMA.org?
$Dipendente(x, y)$	Chi dà lavoro a chi?

reticolo di sussunzione

Queste query formano un **reticolo di sussunzione**, mostrato nella Figura 9.2(a). Il reticolo ha alcune caratteristiche interessanti: ad esempio, il figlio di ogni nodo si ottiene dal padre con una sola sostituzione e il “più alto” descendente comune di due nodi qualsiasi è il risultato dell’applicazione del loro unificatore più generale. La parte del reticolo che sta sopra i fatti ground può essere costruita in modo sistematico (Esercizio 9.5). Una formula che comprende costanti ripetute ha un reticolo leggermente diverso, mostrato nella Figura 9.2(b). La presenza di simboli di funzione e di variabili introduce ancora altre interessanti strutture.

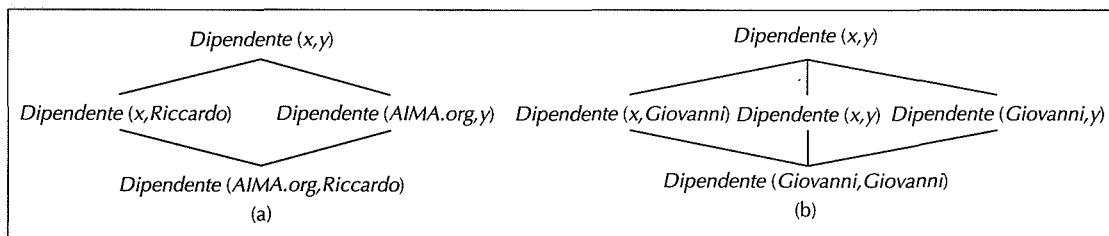


Figura 9.2 (a) Il reticolo di sussunzione il cui nodo più basso contiene la formula $Dipendente(ALMA.org, Riccardo)$. (b) Il reticolo di sussunzione per la formula $Dipendente(Giovanni, Giovanni)$.

Lo schema che abbiamo descritto funziona molto bene quando il reticolo contiene un piccolo numero di nodi: se il predicato ha n argomenti, il reticolo corrispondente comprende $O(2^n)$ nodi. Se è consentito l'uso di simboli di funzione, il numero di nodi è anche esponenziale nella dimensione dei termini della formula da memorizzare: questo può condurre a un'esplosione del numero di indici. A un certo punto, i benefici dell'indicizzazione sono superati dai costi di memorizzazione e gestione di tutti gli indici. Si può adottare una politica prefissata, come quella di memorizzare gli indici solo per le chiavi composte da un predicato più ogni singolo argomento, o una strategia adattiva che crea gli indici più adatti per soddisfare la particolare tipologia di interrogazioni proposte. Nella maggior parte dei sistemi intelligenti, il numero di fatti da memorizzare è sufficientemente piccolo da permettere un'indicizzazione efficiente. Il problema ha goduto del notevole sviluppo tecnologico delle basi di dati industriali e commerciali.

9.3 Concatenazione in avanti

Un algoritmo di concatenazione in avanti per clausole definite proposizionali è stato presentato nel Paragrafo 7.5. L'idea è semplice: si parte con le formule atomiche nella base di conoscenza e si applica Modus Ponens in avanti, aggiungendo nuove formule atomiche, finché non è più possibile compiere altre inferenze. Qui spiegheremo come si può applicare l'algoritmo a clausole definite del primo ordine e come implementarlo in modo efficiente. Clausole definite come *Situazione* \Rightarrow *Risposta* sono particolarmente utili per i sistemi che effettuano inferenze non appena ricevono nuove informazioni. Molti sistemi possono essere definiti in questi termini, e il ragionamento mediante concatenazione in avanti può essere molto più efficiente della dimostrazione di teoremi con la risoluzione. Di conseguenza vale spesso la pena di cercare di costruire una base di conoscenza usando solo clausole definite, così da evitare il costo computazionale della risoluzione.)

Clausole definite del primo ordine

Le clausole definite del primo ordine sono molto simili a quelle proposizionali che abbiamo presentato a pag. 278: sono disgiunzioni di letterali esattamente uno dei quali è positivo. Una clausola definita è una formula atomica, oppure si può scrivere come implicazione il cui antecedente è una congiunzione di letterali positivi e la cui conseguenza è un singolo letterale positivo. Quelle che seguono sono clausole definite del primo ordine:

$$Re(x) \wedge Avido(x) \Rightarrow Malvagio(x).$$

$$Re(Giovanni).$$

$$Avido(y).$$

TP V 7B VK

A differenza di quelle proposizionali, le clausole del primo ordine possono includere variabili: in tal caso le variabili stesse si considerano sempre quantificate universalmente, anche se normalmente il quantificatore universale viene omesso. Le clausole definite sono una forma normale adatta all'applicazione del Modus Ponens generalizzato.

Data la restrizione sulla presenza di un solo letterale positivo, non tutte le basi di conoscenza possono essere convertite in un insieme di clausole definite; tuttavia questo è possibile in molti casi. Considerate il seguente problema:

La legge americana afferma che per un cittadino è un crimine vendere armi a una nazione ostile. Lo stato di Nono, un nemico dell'America, possiede dei missili, e gli sono stati venduti tutti dal Colonnello West, un americano.

Dimostreremo che West è un criminale. Prima di tutto dobbiamo rappresentare i fatti sotto forma di clausole definite della logica del primo ordine. Nel prossimo paragrafo vedremo come risolvere il problema mediante l'algoritmo di concatenazione in avanti.

“Per un americano è un crimine vendere armi a nazioni ostili”:

$$\text{Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Ostile}(z) \Rightarrow \text{Criminale}(x) . \quad (9.3)$$

“Nono... possiede dei missili”. La formula $\exists x \text{ Possiede}(\text{Nono}, x) \wedge \text{Missile}(x)$ si può trasformare in due clausole definite mediante l'eliminazione dell'esistenziale, introducendo una nuova costante M_1 :

$$\text{Possiede}(\text{Nono}, M_1) \quad (9.4)$$

$$\text{Missile}(M_1) . \quad (9.5)$$

“Tutti i missili sono stati venduti dal Colonnello West”:

$$\text{Missile}(x) \wedge \text{Possiede}(\text{Nono}, x) \Rightarrow \text{Vende}(\text{West}, x, \text{Nono}) . \quad (9.6)$$

Dobbiamo anche sapere che i missili sono armi:

$$\text{Missile}(x) \Rightarrow \text{Arma}(x) . \quad (9.7)$$

E anche che un nemico dell'America viene considerato ostile:

$$\text{Nemico}(x, \text{America}) \Rightarrow \text{Ostile}(x) . \quad (9.8)$$

“West è americano”:

$$\text{Americano}(\text{West}) . \quad (9.9)$$

“Lo stato di Nono è un nemico dell'America”:

$$\text{Nemico}(\text{Nono}, \text{America}) . \quad (9.10)$$

Questa base di conoscenza non contiene simboli di funzione ed è quindi un esempio di quelle KB che prendono il nome di Datalog: sono Datalog gli insiemi di clausole definite del primo ordine senza simboli di funzioni. Come vedremo, l'assenza di funzioni rende l'inferenza molto più facile.

function FOL-FC-ASK(*KB*, α) **returns** una sostituzione oppure *false*

inputs: *KB*, la base di conoscenza, un insieme di clausole del primo ordine

α , la query, una formula atomica

variabili locali: *new*, le nuove formule inferite a ogni iterazione

repeat until *new* è vuoto

new $\leftarrow \{\}$

for each formula *r* in *KB* **do**

$(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZZA-SEPARATAMENTE}(r)$

for each θ tale che $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$

per qualche p_1, \dots, p_n nella *KB*

$q' \leftarrow \text{SUBST}(\theta, q)$

if q' non è la rinominazione di una formula già presente nella *KB*

o nell'insieme *new* **then do**

aggiungi q' a *new*

$\phi \leftarrow \text{UNIFY}(q', \alpha)$

if ϕ non è fallimento **then return** ϕ

aggiungi *new* alla *KB*

return *false*

$$\forall f \in KB (f_1, f_2, f_3, \dots, f_n)$$

Figura 9.3 Un algoritmo di concatenazione in avanti intuitivo, ma molto inefficiente. A ogni iterazione l'algoritmo aggiunge alla *KB* tutte le formule atomiche che possono essere inferite in un passo dalle formule di implicazione e dalle formule atomiche già presenti nella *KB*.

Un semplice algoritmo di concatenazione in avanti

Il primo algoritmo di concatenazione in avanti che consideriamo è molto semplice, come si vede nella Figura 9.3: partendo dai fatti noti, si fanno scattare tutte le regole le cui premesse sono soddisfatte, aggiungendo le relative conclusioni ai fatti noti. Il processo si ripete finché si trova una risposta (supponendo che ne basti una), oppure non è più possibile aggiungere nuovi fatti. Notate che un fatto non è "nuovo" se consiste solo nella rinominazione di uno noto. Una formula è la rinominazione di un'altra se tra le due cambiano solo i nomi delle variabili. Ad esempio, *Gradisce(x, Gelato)* e *Gradisce(y, Gelato)* sono una la rinominazione dell'altra, dato che differiscono solo nella scelta di *x o y*: il loro significato è lo stesso, e cioè che a tutti piace il gelato.

Per illustrare il funzionamento di FOL-FC-ASK useremo il nostro esempio del colonnello criminale. Le formule di implicazione sono (9.3), (9.6), (9.7) e (9.8).

rinominazione

Sono necessarie due iterazioni.

- ◆ Nella prima iterazione, la regola (9.3) ha delle premesse non soddisfatte.
La regola (9.6) è soddisfatta con $\{x/M_1\}$, e si può aggiungere alla base di conoscenza il fatto $Vende(West, M_1, Nono)$.
La regola (9.7) è soddisfatta con $\{x/M_1\}$, e si aggiunge $Arma(M_1)$.
La regola (9.8) è soddisfatta con $\{x/Nono\}$, e si aggiunge $Ostile(Nono)$.
- ◆ Nella seconda iterazione, la regola (9.3) è soddisfatta con $\{x/West, y/M_1, z/Nono\}$, e si può aggiungere $Criminale(West)$.

La Figura 9.4 riporta l'albero di dimostrazione generato. Notate che a questo punto non è più possibile effettuare alcuna nuova inferenza, perché ogni formula che si potrebbe derivare mediante la concatenazione in avanti è già esplicitamente presente nella KB. Una base di conoscenza in questa situazione prende il nome di punto fisso del processo inferenziale. I punti fissi raggiunti dalla concatenazione in avanti con clausole definite del primo ordine sono simili a quelli del caso proposizionale, che abbiamo incontrato a pag. 282; la differenza principale sta nel fatto che un punto fisso del primo ordine può includere formule atomiche universalmente quantificate.

FOL-FC-ASK è facile da analizzare: per prima cosa è corretto, perché ogni inferenza è un'applicazione del Modus Ponens generalizzato. Inoltre è completo per le basi di conoscenza di clausole definite; ovvero, è in grado di rispondere a ogni interrogazione le cui soluzioni sono implicate da una KB composta da sole clausole definite. Nel caso di basi Datalog, che non contengono simboli di funzione, la dimostrazione della completezza è abbastanza facile. Cominciamo con il contare il numero di fatti possibili che possono essere aggiunti, che determina il numero massimo di iterazioni. Sia k la massima arità (numero di argomenti) tra tutti i predicati, p il numero dei predicati ed n quello dei simboli di costante. Palesemente non possono esserci più di pn^k fatti ground distinti, ragion per cui dopo quel numero di iterazioni l'algoritmo deve aver raggiunto un punto fisso. A questo punto possiamo dimostrare la completezza in modo molto simile alla concatenazione in avanti proposizionale (v. pag. 282). I dettagli sul passaggio dalla completezza proposizionale a quella del primo ordine sono forniti per l'algoritmo di risoluzione nel Paragrafo 9.5.

Se le clausole definite includono simboli di funzione, FOL-FC-ASK può generare un numero infinito di fatti, per cui dobbiamo stare attenti. nel caso in cui la risposta alla query q sia implicata, possiamo ricorrere al teorema di Herbrand per asserire che l'algoritmo terminerà con successo (v. il Paragrafo 9.5 per una discussione, applicata al caso della risoluzione). Se la query non ha risposta, tuttavia, l'algoritmo potrebbe non terminare mai. Ad esempio, se la base di conoscenza contiene gli assiomi di Peano,

$$NatNum(0)$$

$$\forall n \ NatNum(n) \Rightarrow NatNum(S(n))$$

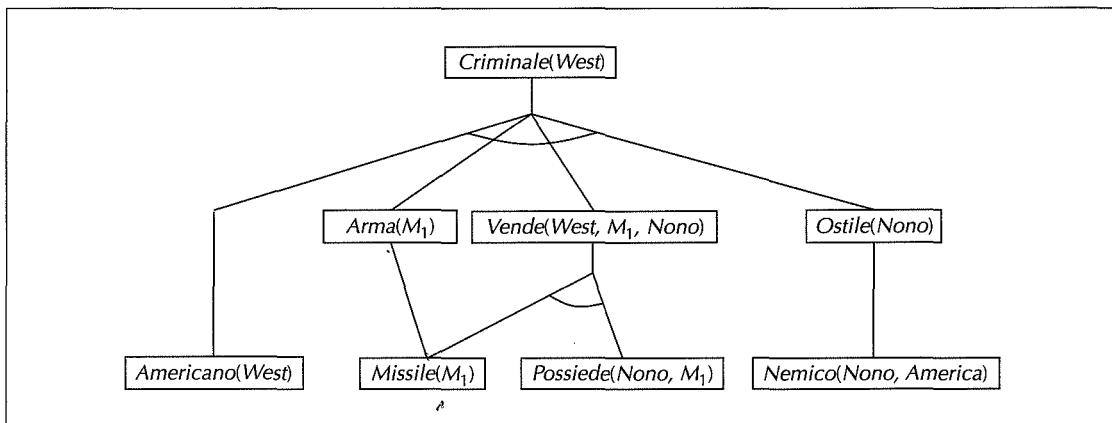


Figura 9.4 L'albero di dimostrazione generato dalla concatenazione in avanti sull'esempio del colonnello criminale. I fatti iniziali si trovano al livello più basso, quelli inferiti nella prima iterazione al livello intermedio, quelli inferiti nella seconda iterazione al livello più alto.

la concatenazione in avanti aggiungerà $\text{NatNum}(S(0))$, $\text{NatNum}(S(S(0)))$, $\text{NatNum}(S(S(S(0))))$ e così via. Questo non si può evitare in alcun modo: come nel caso generale della logica del primo ordine, anche restringendo il campo alle sole clausole definite il problema dell'implicazione rimane semidecidibile.

Concatenazione in avanti efficiente

L'algoritmo di concatenazione in avanti riportato nella Figura 9.3 è progettato pensando alla facilità di comprendere e non all'efficienza. Le possibili fonti di complessità sono tre: prima di tutto, il ciclo interno dell'algoritmo richiede di trovare tutti gli unificatori tali che la premessa di una regola possa unificare con un insieme adeguato di fatti presenti nella base di conoscenza. Questa operazione, spesso chiamata pattern matching, può essere molto costosa computazionalmente. In secondo luogo, a ogni iterazione l'algoritmo deve ricontrolare tutte le regole per vedere se le loro premesse sono soddisfatte, anche se la base di conoscenza è cambiata molto poco. Infine, l'algoritmo potrebbe generare molti fatti che sono del tutto irrilevanti per il suo obiettivo. Tratteremo uno per volta ognuno di questi tre aspetti.

pattern matching

Corrispondenze tra regole e fatti conosciuti

Trovare le corrispondenze tra le premesse di una regola e i fatti presenti nella base di conoscenza potrebbe sembrare un problema alquanto semplice. Supponiamo ad esempio di voler applicare la regola

$$\text{Missile}(x) \Rightarrow \text{Arma}(x)$$

ordinamento dei congiunti



complessità dei dati

Tutto quello che dobbiamo fare è trovare tutti i fatti che unificano con $\text{Missile}(x)$; in una base di conoscenza opportunamente indicizzata questo può essere fatto in un tempo costante per ogni fatto. Ora considerate un regola come

$$\text{Missile}(x) \wedge \text{Possiede}(\text{Nono}, x) \Rightarrow \text{Vende}(\text{West}, x, \text{Nono}) .$$

Ancora una volta possiamo trovare tutti gli oggetti posseduti da Nono in un tempo costante per ogni oggetto; quindi per ognuno di essi potremo verificare se è un missile. Se la base di conoscenza contiene molti oggetti posseduti da Nono e pochi missili, comunque, sarebbe più efficiente trovare innanzitutto questi ultimi e dopo controllare se appartengono a Nono. Questo è il problema dell'ordinamento dei congiunti: ordinando la risoluzione dei congiunti nella premessa di una regola se ne può minimizzare il costo. Purtroppo trovare l'ordinamento ottimo è un problema NP-difficile, ma sono disponibili buone euristiche: ad esempio quella della variabile più vincolata, che abbiamo applicato ai CSP nel Capitolo 5, nel caso ci fossero più oggetti posseduti da Nono che missili ci suggerirebbe di ordinare i congiunti in modo da cercare innanzitutto i missili.

In effetti, la relazione tra pattern matching e soddisfacimento dei vincoli è moltissima. Possiamo considerare ogni congiunto come un vincolo sulle variabili contenute: ad esempio, $\text{Missile}(x)$ è un vincolo unario su x . Estendendo quest'idea, possiamo esprimere ogni CSP di dominio finito con una singola clausola e alcuni fatti ground a essa associati. Considerate il problema di coloratura di una mappa della Figura 5.1, che abbiamo riportato nella Figura 9.5(a): la Figura 9.5(b) fornisce una formulazione equivalente sotto forma di una singola clausola definita. È chiaro che la conclusione Colorabile() potrà essere inferita solo se il CSP ha una soluzione. Dato che i CSP includono come casi speciali i problemi 3SAT, possiamo concludere che cercare il matching tra una clausola definita e un insieme di fatti è un problema NP-difficile.

Potrebbe sembrarvi alquanto deprimente che la concatenazione in avanti contenga come ciclo interno un problema NP-difficile. Ci sono tre modi di rallegrarsi.

- ◆ Possiamo ricordarci che la maggior parte delle regole, nelle basi di conoscenza reali, sono piccole e semplici (come quelle del nostro esempio del colonnello criminale) e non grandi e complesse (come la formulazione di CSP della Figura 9.5). Nel mondo dei database spesso si dà per scontato che la dimensione delle regole e l'arità dei predicati siano entrambe limitate da una costante e ci si preoccupa solo della complessità dei dati, ovvero della complessità dell'inferenza in funzione del numero di fatti ground nel database. È facile dimostrare che la complessità dei dati, per la concatenazione in avanti, è solo polinomiale.
- ◆ Possiamo considerare delle sottoclasse di regole la cui struttura faciliti il matching. Essenzialmente si può dire che ogni clausola Datalog definisca un CSP, ragion per cui il matching sarà trattabile quando lo sarà il CSP corrispondente. Nel Capitolo 5 abbiamo descritto diverse famiglie di CSP tratta-

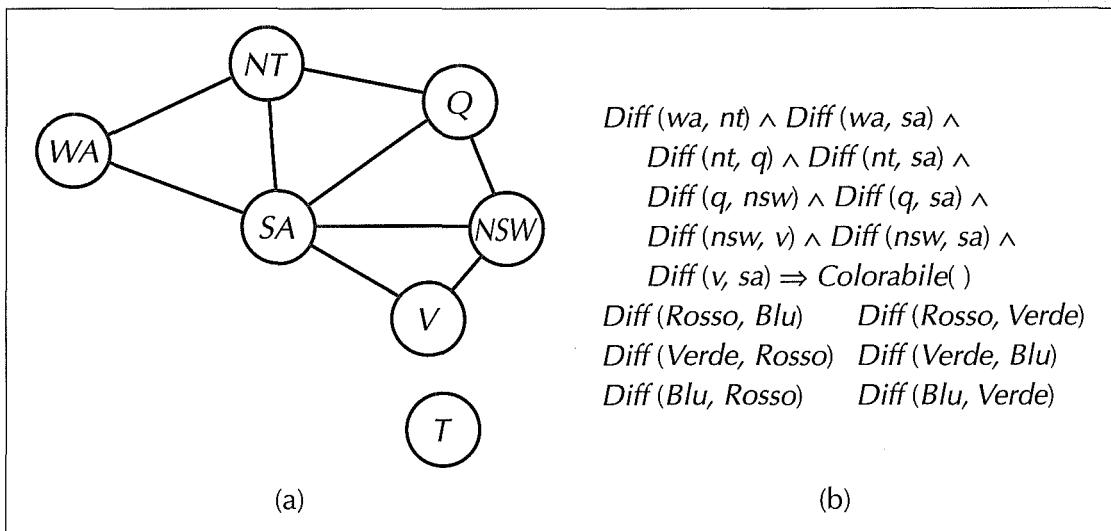


Figura 9.5 (a) Il grafo dei vincoli per la coloratura della mappa dell'Australia (dalla Figura 5.1). (b) Il CSP della coloratura della mappa espresso con un'unica clausola definita. Notate che i domini delle variabili sono definiti implicitamente dalle costanti fornite nei fatti ground per *Diff*.

bili: ad esempio se il grafo dei vincoli (quello i cui nodi rappresentano variabili e i cui archi rappresentano vincoli) si riduce a un albero, il CSP può essere risolto in un tempo lineare. Lo stesso identico risultato vale per il matching delle regole; se ad esempio eliminiamo “South Australia” dalla mappa della Figura 9.5, la clausola risultante è

$$\text{Diff}(wa, nt) \wedge \text{Diff}(nt, q) \wedge \text{Diff}(q, nsw) \wedge \text{Diff}(nsw, v) \Rightarrow \text{Colorabile}()$$

Che corrisponde al CSP ridotto della Figura 5.11. Gli algoritmi di risoluzione di CSP strutturati da albero possono essere applicati direttamente al matching delle regole.

- ◆ Possiamo sforzarci di eliminare tentativi di matching ridondanti nell'algoritmo di concatenazione in avanti, ciò che costituisce il nostro prossimo argomento.

Concatenazione in avanti incrementale

Quando abbiamo mostrato il funzionamento della concatenazione in avanti sull'esempio del colonnello criminale, abbiamo barato; in particolare, abbiamo omesso una parte del matching effettuato dall'algoritmo della Figura 9.3. Nella seconda iterazione, ad esempio, la regola

$$\text{Missile}(x) \Rightarrow \text{Arma}(x)$$

ha ancora una corrispondenza in $\text{Missile}(M_1)$, ma naturalmente la conclusione $\text{Arma}(M_1)$ è già nota e non accade nulla. Questo matching ridondante può essere evitato grazie alla seguente osservazione: *ogni nuovo fatto inferito durante l'iterazione t deve derivare da almeno un fatto nuovo inferito nell'iterazione t - 1*. È palese infatti che ogni inferenza che non richiede un fatto nuovo, precedentemente non disponibile, sarà stata già effettuata nell'iterazione precedente.

Quest'osservazione conduce naturalmente alla formulazione di un algoritmo di concatenazione in avanti incrementale ove, nell'iterazione t , saranno controllate solo le regole le cui premesse includono un congiunto p_i che unifica con un fatto p'_i inferito nell'iterazione $t - 1$. Il passo di matching allora obbligherà p_i a corrispondere a p'_i , ma permetterà agli altri congiunti della stessa regola di corrispondere a fatti ottenuti in qualsiasi iterazione precedente. Quest'algoritmo genera a ogni iterazione esattamente gli stessi fatti di quello della Figura 9.3, ma è molto più efficiente.

Con un'indicizzazione adeguata è facile identificare tutte le regole che possono essere attivate da ogni fatto, e in effetti molti sistemi operano con una modalità di "aggiornamento" in cui la concatenazione in avanti scatta in risposta a ogni nuovo fatto comunicato al sistema. Le inferenze si susseguono a cascata in tutto l'insieme di regole finché non viene raggiunto un punto fisso, e il processo riparte ogni volta che giunge un fatto nuovo.

Tipicamente, l'aggiunta di un fatto provoca l'attivazione di una piccola parte delle regole nella base di conoscenza. Questo significa che un sacco di lavoro viene sprecato nella costruzione ripetuta di corrispondenze parziali che hanno qualche premessa non soddisfatta. Il nostro esempio del colonnello criminale è un po' troppo piccolo per mostrarlo efficacemente, ma notate comunque la costruzione di una corrispondenza parziale nella prima iterazione tra la regola

$$\text{Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Ostile}(z) \Rightarrow \text{Criminale}(x)$$

e il fatto $\text{Americano}(\text{West})$. Questa corrispondenza parziale viene scartata e ricostruita daccapo nella seconda iterazione (nella quale viene soddisfatta con successo). Sarebbe meglio mantenere in memoria le corrispondenze parziali anziché scarfarle, completandole gradualmente man mano che giungono nuovi fatti.

L'algoritmo *rete*³ è stato il primo a occuparsi seriamente di questo problema. L'algoritmo-pre-processa l'insieme di regole nella base di conoscenza per costruire una specie di rete dataflow in cui ogni nodo è un letterale di una premessa di una regola. I legami delle variabili attraversano la rete e sono filtrati via quando non

³ Il nome deriva dalla parola latina, che ha lo stesso significato di quella italiana.

corrispondono ad alcun letterale. Se due letterali in una regola condividono una variabile, come *Vende(x, y, z) \wedge Ostile(z)* nel nostro esempio, i legami di ogni letterale sono filtrati attraverso un nodo di uguaglianza. Un legame di variabile che raggiunge il nodo di un letterale *n*-ario come *Vende(x, y, z)* potrebbe dover aspettare che le altre variabili siano legate prima che il processo possa continuare. In ogni istante, lo stato della rete contiene tutte le corrispondenze parziali raggiunte, permettendo di evitare una gran quantità di calcoli ripetuti.

I sistemi a rete come quello descritto, e i molti miglioramenti che vi sono stati apportati, sono stati un componente chiave dei cosiddetti sistemi di produzioni, che sono stati tra i primi sistemi di concatenazione in avanti di grande diffusione.⁴ Il sistema XCON (originariamente denominato R1, McDermott, 1982) era basato su un sistema di produzioni. XCON conteneva diverse migliaia di regole per la progettazione di configurazioni di componenti per computer per i clienti della Digital Equipment Corporation. Fu uno dei primi chiari successi commerciali nel campo emergente dei sistemi esperti. In seguito sono stati costruiti molti sistemi basati sulla stessa tecnologia, che è stata implementata nel linguaggio di uso generale OPS-5.

sistemi di produzioni

I sistemi di produzioni sono anche popolari nel campo delle architetture cognitive (che si ripropongono cioè di modellare il ragionamento umano), come ACT (Anderson, 1983) e SOAR (Laird et al., 1987). In tali sistemi la “memoria di lavoro” rappresenta quella umana a breve termine, mentre le produzioni fanno parte della rappresentazione della memoria a lungo termine. A ogni ciclo si cercano corrispondenze tra le produzioni e i fatti contenuti nella memoria di lavoro. Una produzione le cui condizioni sono soddisfatte può aggiungere o cancellare fatti dalla memoria di lavoro. A differenza di quello che accade di solito con i database, i sistemi di produzioni hanno tipicamente molte regole e un numero relativamente piccolo di fatti. Sfruttando tecniche di matching adeguatamente ottimizzate, alcuni sistemi moderni possono operare in tempo reale con più di un milione di regole.

architetture cognitive

Fatti irrilevanti

L’ultima fonte di inefficienza nella concatenazione in avanti sembra intrinseca nello stesso approccio, e si manifesta anche in un contesto proposizionale (v. Paragrafo 7.5). La concatenazione in avanti effettua in ogni momento tutte le inferenze consentite dai fatti conosciuti, anche se sono del tutto irrilevanti per raggiungere l’obiettivo. Nel nostro esempio del colonnello criminale il problema non è saltato

⁴ Nei sistemi di produzioni, la parola produzione indica una regola condizione-azione.

magic set

all'occhio perché non c'erano regole da cui potessero essere tratte conclusioni irrilevanti. In altri casi (se ad esempio la base di conoscenza dovesse mescolare informazioni sui missili con una gran quantità di particolari sulle abitudini alimentari degli americani), FOL-FC-ASK finirà con il generare una gran quantità di conclusioni inutili.

Un modo di evitare tutto ciò è usare la concatenazione all'indietro, che descriveremo tra poco. Un'altra soluzione è restringere la concatenazione in avanti a un insieme selezionato di regole; quest'approccio è già stato discusso nel contesto proposizionale. Un terzo metodo è stato sviluppato dalla comunità dei database deduttivi, in cui la concatenazione in avanti è comunemente usata. L'idea è riscrivere l'insieme di regole utilizzando informazione presa dall'obiettivo in modo tale che solo i legami rilevanti – quelli che appartengono al cosiddetto “insieme magico” o **magic set** – siano presi in considerazione durante l'inferenza in avanti. Ad esempio, se l'obiettivo è *Criminale(West)*, la regola che ha come conclusione *Criminale(x)* sarà riscritta in modo da includere un congiunto addizionale che vincola il valore di *x*:

$$\text{Magic}(x) \wedge \text{Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Ostile}(z) \Rightarrow \text{Criminale}(x).$$

Naturalmente, alla KB bisognerà aggiungere il fatto *Magic(West)*. In questo modo, anche se la base di conoscenza contiene fatti che riguardano milioni di americani, durante il processo di inferenza in avanti sarà considerato il solo colonnello West. Il processo completo per definire gli insiemi magici e riscrivere la base di conoscenza è troppo complesso per approfondirlo qui ma l'idea, a grandi linee, consiste nell'eseguire una specie di inferenza all'indietro “generica” partendo dall'obiettivo, per determinare quali legami di variabili devono essere vincolati. La soluzione dei magic set può quindi essere considerata una specie di approccio ibrido che sfrutta l'inferenza in avanti unita a una fase di pre-processo all'indietro.

9.4

Concatenazione all'indietro

La seconda grande famiglia di algoritmi di inferenza logica utilizza l'approccio della concatenazione all'indietro, che abbiamo presentato nel Paragrafo 7.5. Questi algoritmi procedono dall'obiettivo seguendo le regole a ritroso per trovare fatti noti che supportino la dimostrazione. Noi presenteremo l'algoritmo base, descrivendo poi il suo utilizzo all'interno della programmazione logica, che è la forma più diffusa di ragionamento automatico. Vedremo anche che la concatenazione all'indietro presenta degli svantaggi rispetto a quella in avanti, e considereremo alcune possibili soluzioni. Infine, prenderemo in esame la stretta correlazione tra la programmazione logica e i problemi di soddisfacimento di vincoli.

9.4 Concatenazione all'indietro

Sottosezione

function FOL-BC-ASK(*KB*, *obiettivi*, θ) **returns** un insieme di sostituzioni

inputs: *KB*, una base di conoscenza

obiettivi, una lista di congiunti che formano una query (θ già applicato)

θ , la sostituzione corrente, inizialmente la sostituzione vuota { }

variabili locali: *risposte*, un insieme di sostituzioni, inizialmente vuoto

if *obiettivi* è vuoto **then return** { θ }

$q' \leftarrow \text{SUBST}(\theta, \text{PRIMO}(\text{obiettivi}))$

for each formula *r* **in** *KB* **dove** STANDARDIZZA-SEPARATAMENTE(*r*) = $(p_1 \wedge \dots \wedge p_n \Rightarrow q)$

e $\theta' \leftarrow \text{UNIFY}(q, q')$ **ha successo**

nuovi_obiettivi $\leftarrow [p_1, \dots, p_n \mid \text{RESTO}(\text{obiettivi})]$

risposte $\leftarrow \text{FOL-BC-ASK}(\text{KB}, \text{nuovi_obiettivi}, \text{COMPOSE}(\theta', \theta)) \cup \text{risposte}$

return *risposte*

Figura 9.6 Un semplice algoritmo di concatenazione all'indietro.

Un algoritmo di concatenazione all'indietro

La Figura 9.6 mostra un semplice algoritmo di concatenazione all'indietro, FOL-BC-ASK. L'invocazione prende come parametro una lista di obiettivi che inizialmente contiene un solo elemento, la query originale, e restituisce l'insieme di tutte le sostituzioni che soddisfano la query. La lista può essere considerata una specie di "pila" a disposizione dell'algoritmo: se tutti gli obiettivi contenuti possono essere soddisfatti, il ramo corrente della dimostrazione ha avuto successo. L'algoritmo prende il primo obiettivo della lista e cerca tutte le clausole nella base di conoscenza il cui letterale positivo, o testa, unifica con esso. Ogni clausola così identificata fa scattare una chiamata ricorsiva in cui la sua premessa, o corpo della clausola, viene aggiunta alla pila di obiettivi. Ricordate che i fatti sono clausole che hanno una testa ma nessun corpo, per cui quando un obiettivo unifica con un fatto noto, non viene aggiunta alla pila alcun sotto-obiettivo e l'obiettivo originario è risolto. La Figura 9.7 riporta l'albero di dimostrazione che deriva Criminale(West) dalle formule che vanno da (9.3) a (9.10).

L'algoritmo utilizza la **composizione** delle sostituzioni. COMPOSE(θ_1, θ_2) è la sostituzione il cui effetto è identico a quello dell'applicazione di ogni sostituzione una dopo l'altra. In altre parole,

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p)).$$

composizione

Nell'algoritmo i legami di variabile correnti, memorizzati in θ , sono composti con quelli che risultano dall'unificazione dell'obiettivo con la testa di una clausola, fornendo un nuovo insieme di legami correnti per la chiamata ricorsiva.

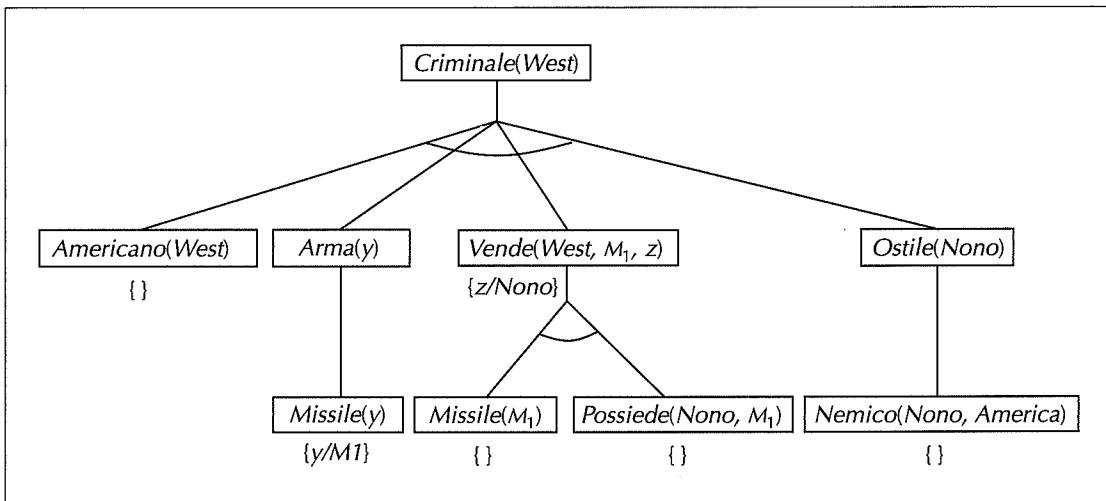


Figura 9.7 L'albero costruito con la concatenazione all'indietro per dimostrare che il Colonnello West è un criminale. L'albero va letto in profondità e da sinistra a destra. Per provare *Criminale(West)* dobbiamo dimostrare i quattro congiunti al livello inferiore. Alcuni di essi sono nella base di conoscenza, altri richiedono un'ulteriore concatenazione all'indietro. I legami per ogni unificazione che ha successo sono indicati accanto al sotto-oggettivo corrispondente. Notate che una volta che un sotto-oggettivo in una congiunzione ha successo, la sua sostituzione viene applicata anche ai sotto-oggettivi successivi. In questo modo quando FOL-BC-ASK arriva a considerare l'ultimo congiunto, che originariamente era *Ostile(z)*, *z* è già legata a *Nono*.

La concatenazione all'indietro, così come l'abbiamo scritta, è chiaramente un algoritmo di ricerca in profondità: questo significa che i suoi requisiti spaziali sono lineari con le dimensioni della dimostrazione (tralasciando, per ora, lo spazio richiesto per la memorizzazione delle soluzioni). Questo significa anche che, a differenza di quella in avanti, la concatenazione all'indietro soffre del problema degli stati ripetuti e dell'incompletezza. Discuteremo questi problemi e alcune possibili soluzioni, ma prima vediamo come la concatenazione all'indietro è applicata nei sistemi di programmazione logica.

Programmazione logica

La programmazione logica è uno strumento che arriva molto vicino ad avverare l'ideale dichiarativo che abbiamo illustrato nel Capitolo 7: la possibilità di costruire sistemi esprimendo semplicemente la conoscenza in un linguaggio formale e risolvendo i problemi attraverso un processo di inferenza applicato a tale conoscenza. L'ideale è riassunto dall'equazione di Robert Kowalski,

$$\text{Algoritmi} = \text{Logica} + \text{Controllo} .$$

Il linguaggio di programmazione logica più usato è di gran lunga **Prolog**: i suoi utenti sono centinaia di migliaia. La sua applicazione è normalmente rivolta allo sviluppo rapido di prototipi e alle attività che richiedono manipolazione di simboli, come la scrittura di compilatori (Van Roy, 1990), ed è anche utilizzato per il riconoscimento sintattico del linguaggio naturale (Pereira e Warren, 1980). Sono stati scritti molti sistemi esperti in Prolog nei domini legale, medico, finanziario e molti altri.

Prolog

I programmi Prolog consistono in un insieme di clausole definite espresse con una notazione un po' differente da quella standard della logica del primo ordine. Le variabili sono scritte in maiuscolo, le costanti in minuscolo. Nelle clausole la testa precede il corpo, l'operatore `:`- indica l'implicazione verso sinistra, i letterali del corpo sono separati da virgolette e la fine della formula è indicata da un punto:

```
criminale(X) :- americano(X), arma(Y), vende(X, Y, Z),
ostile(Z).
```

Prolog include "zucchero sintattico" per l'aritmetica e per rappresentare le liste. Come esempio, ecco un programma Prolog `concatena(X, Y, Z)`, che ha successo se la lista `Z` è il risultato della concatenazione delle liste `X` e `Y`:

```
concatena([], Y, Y).
concatena([A|X], Y, [A|Z]) :- concatena(X, Y, Z).
```

Possiamo leggere queste clausole come: (1) concatenare una lista vuota e `Y` produce la stessa lista `Y`; (2) `[A|Z]` è il risultato della concatenazione di `[A|X]` e `Y`, a patto che `Z` sia il risultato della concatenazione di `X` e `Y`. Questa definizione sembra molto simile a quella corrispondente in Lisp, ma in effetti è molto più potente. Se per esempio scriviamo la query `concatena(A, B, [1, 2])` – quali due liste possono essere concatenate per dare la lista `[1, 2]`? – otteniamo le tre soluzioni

```
A= []
B= [1, 2]
A= [1]
B= [2]
A= [1, 2]
B= []
```

L'esecuzione dei programmi Prolog è svolta mediante una concatenazione all'indietro in profondità, in cui le clausole vengono provate nell'ordine con cui sono state scritte nella base di conoscenza. Alcuni aspetti del Prolog differiscono dell'inferenza logica standard.

- È presente un insieme di funzioni aritmetiche predefinite. I letterali che usano questi simboli di funzione sono "dimostrati" eseguendo codice e non mediante ulteriori inferenze. Ad esempio, l'obiettivo "`X is 4+3`" risulta vero con `X` legata a 7. D'altra parte, l'obiettivo "`5 is X+Y`" fallisce sempre, perché le funzioni aritmetiche predefinite non risolvono equazioni arbitrarie.⁵

⁵ Notate che un programma Prolog a cui sono stati forniti gli assiomi di Peano può trovare le risposte di interrogazioni simili mediante l'inferenza.

- ◆ Alcuni predicati predefiniti hanno effetti collaterali. Tra questi ci sono quelli di input-output e quelli di asserzione/ritrattazione utilizzati per modificare la base di conoscenza. Predicati simili non hanno alcuna controparte nella logica e possono produrre effetti strani: ad esempio, se si asseriscono alcuni fatti in un ramo dell'albero di dimostrazione che a un certo punto fallisce.

- ◆ Prolog permette l'uso di una forma di negazione chiamata negazione come fallimento. Un obiettivo negato not P è considerato dimostrato se il sistema non riesce a dimostrare P. Così la formula

```
vivo(X) :- not morto(X).
```

può essere letta come “tutti sono vivi se non si può dimostrare che sono morti”.

- ◆ Prolog ha un operatore di uguaglianza, =, che però non ha la stessa potenza dell'uguaglianza logica. Un obiettivo che include l'uguaglianza è soddisfatto se i due termini sono unificabili, in caso contrario fallisce. Così X+Y=2+3 ha successo con X legata a 2 e Y legata a 3, mentre stellastellina=stellabrillante fallisce in ogni caso, mentre nella logica classica potrebbe essere vera o no. Non si possono asserire fatti o formulare regole che riguardano l'uguaglianza.
- ◆ L'algoritmo di unificazione del Prolog non include il controllo di occorrenza. Questo significa che potranno essere effettuate inferenze scorrette; raramente questo rappresenta un problema, tranne quando si usa Prolog per la dimostrazione di teoremi matematici.

Le decisioni prese durante la progettazione di Prolog rappresentano un compromesso tra la potenza dichiarativa e l'efficienza dell'esecuzione (per quanto poteva essere compresa in quei tempi). Torneremo sull'argomento dopo aver esaminato l'implementazione di Prolog.

Implementazione efficiente di programmi logici

Un programma Prolog può essere eseguito in due modi: interpretato o compilato. L'interpretazione essenzialmente consiste nell'esecuzione dell'algoritmo FOL-BC-ASK della Figura 9.6, con il programma stesso come base di conoscenza. Abbiamo detto “essenzialmente” perché gli interpreti Prolog contengono diverse migliorie progettate per massimizzare la velocità d'esecuzione: qui ne citeremo solo due.

Prima di tutto, invece di costruire la lista di tutte le possibili risposte per ogni sotto-obiettivo prima di considerare il successivo, gli interpreti Prolog generano una risposta e la “promessa” di generare le altre quando quella corrente sarà stata completamente esplorata. Questa promessa prende il nome di punto di scelta. Quando la ricerca in profondità completa la sua esplorazione delle soluzioni possibili derivate dalla risposta corrente e torna indietro al punto di scelta, quest'ultimo

è espanso per generare una nuova risposta del sotto-oggetto e un nuovo punto di scelta. Questo approccio consente di risparmiare tempo e spazio, e fornisce un'interfaccia molto semplice per il debugging dato che in qualsiasi istante il sistema considera un solo cammino.

In secondo luogo, la nostra semplice implementazione di FOL-BC-ASK passa gran parte del tempo generando e componendo sostituzioni. Prolog implementa le sostituzioni usando variabili logiche che possono ricordare il loro legame corrente: in ogni determinato istante, ogni variabile del programma può essere libera o legata a qualche valore. Prese insieme, le variabili e i loro valori definiscono implicitamente la sostituzione per il ramo corrente della dimostrazione. L'estensione del cammino può solo aggiungere nuovi legami di variabile, perché il tentativo di aggiungere un legame diverso a una variabile già legata causerà immancabilmente il fallimento dell'unificazione. Quando un cammino della ricerca fallisce Prolog ritorna all'ultimo punto di scelta, e in quel momento potrebbe essere necessario annullare il legame di qualche variabile. Per far questo il programma tiene traccia di tutte le variabili legate in una pila chiamata traccia (*trail*). Una nuova variabile viene spinta sulla traccia non appena UNIFY-VAR la lega a un valore. Quando un obiettivo fallisce e bisogna tornare a un punto di scelta precedente, le variabili rimosse dalla pila ridiventano libere.)

Anche gli interpreti Prolog più efficienti richiedono diverse migliaia di cicli macchina per ogni passo inferenziale in virtù del costo di accesso ai dati indicizzati, dell'unificazione e della costruzione di una pila di chiamate ricorsive. In effetti, l'interprete si comporta come se non avesse mai visto prima il programma; ad esempio deve trovare le clausole che corrispondono all'obiettivo. Un programma Prolog compilato, d'altra parte, rappresenta una procedura di inferenza per uno specifico insieme di clausole, per cui sa già quali clausole corrispondono. In pratica Prolog genera un mini-dimostratore di teoremi per ogni singolo predicato, eliminando gran parte del costo dell'interpretazione. È anche possibile codificare direttamente (*open code*) la procedura di unificazione di ogni chiamata, evitando così l'analisi esplicita della struttura dei termini. Per ulteriori dettagli sull'unificazione codificata direttamente si può leggere Warren et al. (1977).

Le istruzioni macchina dei computer odierni non si adattano bene alla semantica di Prolog, ragion per cui la maggior parte dei compilatori traduce in un linguaggio intermedio. Il più popolare è quello della Warren Abstract Machine, o WAM, così chiamata in onore di David H. D. Warren, uno degli implementatori del primo compilatore Prolog. WAM è un insieme di istruzioni astratte adatte al Prolog e può essere interpretato o tradotto a sua volta in codice macchina. Altri compilatori traducono invece il Prolog in un linguaggio ad alto livello come Lisp o C e si appoggiano poi ai compilatori esistenti per il passo finale di traduzione in linguaggio macchina. Per esempio, la definizione del predicato Concatena può essere compilata nel codice mostrato nella Figura 9.8. Ci sono diversi punti degni di nota.

traccia *Pi* *Co*
chiamata
tiene *no* *di*
tutte *le* *var*
legame.

codificare direttamente

procedure CONCATENA(*ax, y, az, continuazione*)

```

traccia ← PUNTATORE-GLOBALE-TRACCIA()
if ax = [] e UNIFY(y, az) then CALL(continuazione)
RESET-TRACCIA(traccia)
a ← NUOVA-VARIABILE(); x ← NUOVA-VARIABILE(); z ← NUOVA-VARIABILE()
if UNIFY(ax, [a|x]) e UNIFY(az, [a|z]) then CONCATENA(x, y, z, continuazione)

```

Figure 9.8 Pseudocodice che rappresenta il risultato della compilazione del predicato Concatena. La funzione NUOVA-VARIABILE restituisce appunto una nuova variabile, distinta da tutte quelle usate in precedenza. La procedura CALL(*continuazione*) prosegue l'esecuzione con la continuazione specificata.

- ◆ Non è più necessario cercare nella base di conoscenza le clausole Concatena, che sono diventate una procedura: le inferenze sono effettuate semplicemente invocando la procedura stessa.
- ◆ Come abbiamo detto poco fa, i legami correnti delle variabili sono memorizzati su una traccia. Il primo passo della procedura salva lo stato corrente della traccia, in modo che possa essere recuperato da RESET-TRACCIA se la prima clausola dovesse fallire. Questo scioglierà tutti i legami generati dalla prima chiamata a UNIFY.
- ◆ La parte più delicata è l'uso delle **continuazioni** per implementare i punti di scelta. Potete considerare una continuazione come una procedura "impacchettata" insieme a una lista di argomenti per definire cosa dev'essere fatto non appena l'obiettivo corrente ha successo. Da una procedura come CONCATENA non si può uscire semplicemente quando l'obiettivo ha successo, dato che ciò potrebbe avvenire in molti modi, che vanno esplorati tutti. La continuazione risolve questo problema perché può essere passata come argomento ogni volta che l'obiettivo ha successo. Nel codice di CONCATENA, se il primo argomento è vuoto il predicato CONCATENA ha avuto successo. A questo punto possiamo invocare (mediante CALL) la continuazione con i legami appropriati sulla traccia, per fare qualsiasi cosa occorra: ad esempio, se la chiamata a CONCATENA si trovasse al livello più alto, la continuazione stamperebbe i legami delle variabili.

Prima del lavoro svolto da Warren sulla compilazione delle inferenze in Prolog, la programmazione logica era troppo lenta per un uso generale. I compilatori sviluppati da Warren e altri hanno permesso al codice Prolog di raggiungere velocità paragonabili a quelle del C su molti benchmark standard (Van Roy, 1990). Naturalmente, il fatto che si possa scrivere un programma di pianificazione o un

riconoscitore di linguaggio naturale con poche decine di righe di Prolog lo rende più appropriato del C per la prototipazione della maggior parte dei progetti di ricerca di IA di piccola scala.

Un altro modo di ottenere grandi miglioramenti nelle prestazioni è l'uso del parallelismo. Ci sono due approcci principali: il primo, chiamato **OR-parallelismo**, deriva dalla possibilità che un obiettivo unifichi con più clausole diverse della base di conoscenza. Ognuna di esse dà origine a un ramo indipendente nello spazio di ricerca, e tutti i rami possono essere risolti in parallelo. Il secondo approccio, chiamato **AND-parallelismo**, deriva dalla possibilità di risolvere in parallelo ogni congiunto nel corpo di un'implicazione. L'AND-parallelismo è più difficile da ottenere, perché le soluzioni di intere congiunzioni richiedono che i legami delle variabili siano tutti consistenti: ogni ramo della congiunzione deve quindi comunicare con tutti gli altri per assicurare la correttezza della soluzione globale.

OR-parallelismo

AND-parallelismo

Inferenza ridondante e cicli infiniti

Passiamo ora ad occuparci del tallone d'Achille del Prolog: l'incompatibilità tra la ricerca in profondità e gli alberi che presentano stati ripetuti e cammini infiniti. Considerate il seguente programma logico che decide se esiste un cammino tra due punti di un grafo orientato:

```
cammino(X, Z) :- arco(X, Z).
cammino(X, Z) :- cammino(X, Y), arco(Y, Z).
```

La Figura 9.9(a) mostra un semplice grafo a tre nodi, descritto dai fatti `arco(a, b)` e `arco(b, c)`. Con il programma qui sopra, la query `cammino(a, c)` genera l'albero di dimostrazione riportato nella Figura 9.10(a). D'altra parte, se scriviamo le due clausole in ordine inverso

```
cammino(X, Z) :- cammino(X, Y), arco(Y, Z).
cammino(X, Z) :- arco(X, Z).
```

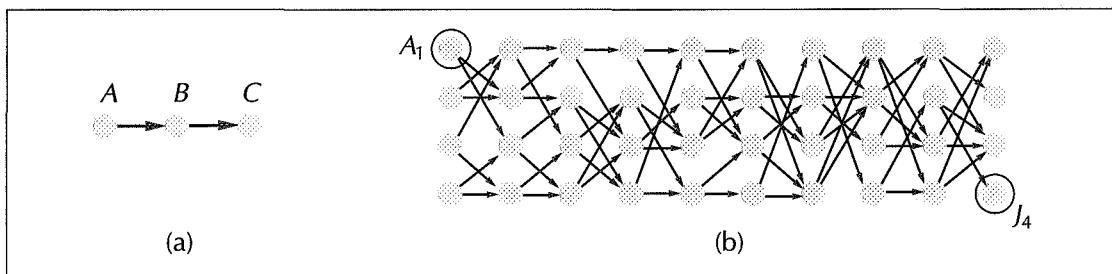


Figura 9.9 (a) Trovare un cammino da A a C può portare Prolog in un ciclo infinito. (b) Un grafo in cui ogni nodo è collegato a due successori casuali del livello successivo. Trovare un cammino da A_1 a J_4 richiede 877 inferenze.

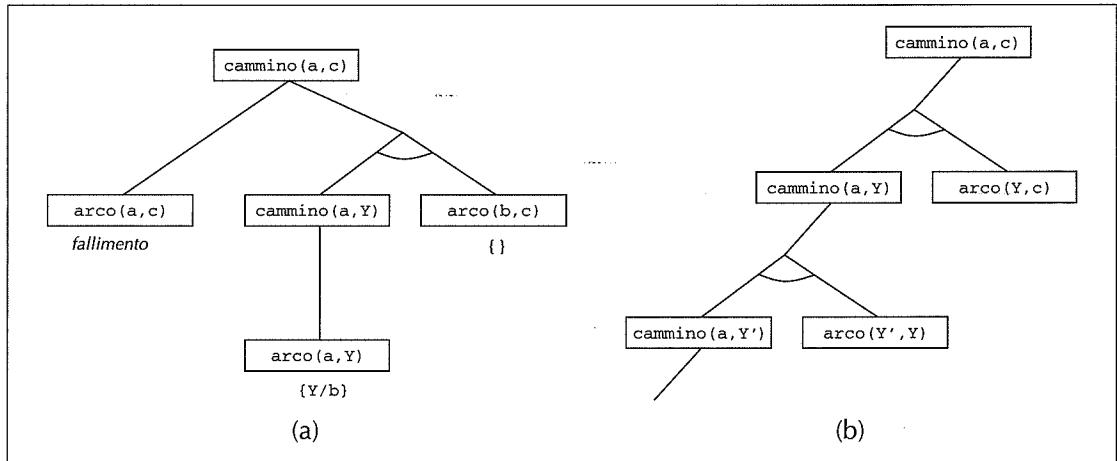
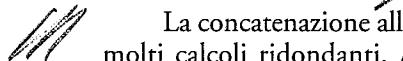


Figure 9.10 (a) Dimostrazione che esiste un cammino da A a C. (b) Quando le clausole sono in ordine "sbagliato", può essere generato un albero di dimostrazione infinito.

Prolog segue il cammino infinito mostrato nella Figura 9.10(b). Ne consegue che Prolog è incompleto come dimostratore di teoremi per clausole definite; l'esempio mostra che questo avviene anche nel caso di programmi Datalog. La ragione è che, per certe basi di conoscenza, Prolog non riesce a dimostrare formule effettivamente implicate. Notate che la concatenazione in avanti non ha questo problema: una volta inferite `cammino(a,b)`, `cammino(b,c)` e `cammino(a,c)`, la concatenazione in avanti si arresta.



La concatenazione all'indietro in profondità ha anche il problema di eseguire molti calcoli ridondanti. Ad esempio, per trovare il cammino da A_1 a J_4 nella Figura 9.9(b), Prolog esegue 877 inferenze, la maggior parte delle quali per cercare tutti i cammini possibili verso nodi da cui l'obiettivo è irraggiungibile. Questo problema è simile a quello degli stati ripetuti che abbiamo discusso nel Capitolo 3. Il numero totale di inferenze può crescere esponenzialmente con il numero di fatti ground generati. Applicando la concatenazione in avanti, invece, saranno generati al più n^2 fatti `cammino(X,Y)` che collegano n nodi. Per il problema della Figura 9.9(b), sono necessarie solo 62 inferenze.

La concatenazione in avanti applicata a problemi di ricerca su grafo è un esempio di programmazione dinamica, in cui le soluzioni ai sottoproblemi sono costruite incrementalmente partendo da quelle dei sottoproblemi più piccoli, e memorizzate in una cache per evitare di ricalcolarle. Possiamo ottenere un effetto simile in un sistema che utilizza la concatenazione all'indietro usando la tecnica della memoizzazione, che consiste nel memorizzare le soluzioni di tutti i sottoproblemi man mano che vengono trovate in modo da poterle riusare all'occorrenza.

anziché ricalcolarle. Questo è l'approccio adottato dai sistemi di programmazione logica con tabelle, che sfruttano meccanismi efficienti di salvataggio e recupero dati per eseguire la memoizzazione. La programmazione logica con tabelle è guidata dall'obiettivo come la concatenazione all'indietro, ma è efficiente come quella in avanti; inoltre è completa per programmi Datalog, il che significa che il programmatore deve preoccuparsi meno dei cicli infiniti.

programmazione
logica con tabelle

Programmazione logica con vincoli

Nella nostra discussione della concatenazione in avanti (v. Paragrafo 9.3) abbiamo mostrato che i problemi di soddisfacimento di vincoli (CSP) possono essere espressi sotto forma di clausole definite. Il Prolog standard risolve tali problemi esattamente come l'algoritmo con backtracking illustrato nella Figura 5.3.

Dato che il backtracking enumera il dominio delle variabili, è applicabile solo a CSP con dominio finito. Usando il gergo del Prolog, dev'esserci un numero finito di soluzioni per ogni obiettivo con variabili libere: ad esempio, l'obiettivo `diff(q, sa)`, che afferma che i colori di Queensland e South Australia devono essere diversi, con tre colori ha sei soluzioni. I CSP a dominio infinito, come quelli che prevedono variabili intere o reali, richiedono algoritmi molto diversi, come la propagazione dei vincoli o la programmazione lineare.

La clausola qui sotto è verificata se i tre numeri soddisfano la disugualanza triangolare:

```
triangolo(X, Y, Z) :-  
    X >= 0, Y >= 0, Z >= 0, X + Y >= Z, Y + Z >= X, X + Z >= Y.
```

Se sottoponiamo a Prolog la query `triangolo(3, 4, 5)`, il sistema funziona bene. Al contrario, l'interrogazione `triangolo(3, 4, Z)` non potrà avere alcuna soluzione, perché Prolog non può gestire il sotto-obiettivo `Z >= 0`. La difficoltà sta nel fatto che in Prolog le variabili devono necessariamente essere in uno di due stati: libere o legate a un particolare termine.

Legare una variabile a un particolare termine può essere considerata una forma estrema di vincolo, e precisamente un vincolo di uguaglianza. La programmazione logica con vincoli (o CLP, dall'acronimo inglese) permette di vincolare le variabili anziché legarle. La soluzione di un problema logico con vincoli è il più specifico insieme di vincoli sulle variabili della query che può essere derivato dalla base di conoscenza. Ad esempio, la soluzione dell'interrogazione `triangolo(3, 4, Z)` è il vincolo `7 >= Z >= 1`. I programmi logici standard a questo punto diventano un caso speciale di CLP che comprende solo vincoli di uguaglianza, cioè legami.

programmazione
logica con vincoli

I sistemi CLP incorporano diversi algoritmi per la risoluzione dei vincoli permessi nel linguaggio. Un sistema che permette la formulazione di disugualanze lineari su variabili reali, ad esempio, potrebbe includere un algoritmo specializzato di programmazione lineare per la loro soluzione. I sistemi CLP adottano anche un

approccio molto più flessibile alla risoluzione delle query standard della programmazione logica: ad esempio, invece di applicare sempre (poniamo) un algoritmo con backtracking in profondità da sinistra a destra, potrebbero utilizzare uno degli algoritmi più efficienti che abbiamo discusso nel Capitolo 5, come l'ordinamento euristico dei congiunti, il backjumping, il condizionamento con insieme di taglio etc. Si può dire quindi che uniscono elementi tipici degli algoritmi di soddisfacimento di vincoli, della programmazione logica e delle basi di dati deduttive.

I sistemi CLP possono anche avvantaggiarsi delle tecniche di ottimizzazione per la ricerca su CSP che abbiamo descritto nel Capitolo 5, come l'ordinamento delle variabili e dei valori, la verifica in avanti e il backtracking intelligente. Sono stati definiti molti sistemi che consentono al programmatore di esercitare un controllo sull'ordine di ricerca dell'inferenza. Ad esempio, il linguaggio MRS (Genesereth e Smith, 1981; Russell, 1985) permette al programmatore di scrivere metaregole per specificare quali congiunti provare per primi. L'utente potrebbe scrivere una regola per far sì che l'obiettivo con il minor numero di variabili sia il primo a essere considerato, o aggiungere regole specifiche del dominio per la gestione di particolari predicati.

metaregole

9.5 Risoluzione

L'ultima delle nostre tre famiglie di sistemi logici è basata sulla **risoluzione**. Abbiamo visto nel Capitolo 7 che la risoluzione è una procedura di inferenza completa per la refutazione nella logica proposizionale. In questo paragrafo estenderemo la sua applicazione alla logica del primo ordine.

La questione dell'esistenza di procedure di dimostrazione complete è di grande interesse per i matematici. Infatti se fosse possibile trovarne una per le asserzioni matematiche, ci sarebbero due conseguenze: prima di tutto, la verità di ogni congettura potrebbe essere determinata automaticamente; in secondo luogo, l'intera matematica potrebbe essere espressa come conseguenza logica di un insieme di assiomi fondamentali. La questione della completezza quindi ha stimolato alcune delle ricerche matematiche più importanti del XX secolo. Nel 1930, il matematico austriaco Kurt Gödel ha dimostrato il primo **teorema di completezza** per la logica del primo ordine, che afferma che ogni formula implicata ha una dimostrazione di lunghezza finita (naturalmente questo non significava disporre di una procedura *pratica* di dimostrazione; per questo si è dovuta aspettare la pubblicazione dell'algoritmo di risoluzione di J. A. Robinson, nel 1965). Nel 1931, Gödel dimostrò l'ancor più famoso **teorema di incompletezza**, che afferma che un sistema logico che include il principio di induzione (senza il quale si può costruire ben poca parte della matematica discreta) è necessariamente incompleto. Ne consegue che esistono formule vere per cui è impossibile costruire una prova finita all'interno del sistema. In altre parole, l'ago potrebbe anche trovarsi nel metaforico pagliaio, ma nessuna procedura può garantire il suo ritrovamento.

teorema di completezza

teorema di incompletezza

Nonostante il risultato di Gödel, i dimostratori di teoremi basati sulla risoluzione sono stati applicati ampiamente per derivare teoremi matematici, molti dei quali non avevano una dimostrazione preesistente; inoltre tra le altre cose sono stati utilizzati per verificare progetti hardware e generare programmi logicamente corretti.

Forma normale congiuntiva per la logica del primo ordine

Come nel caso proposizionale, anche la risoluzione del primo ordine richiede che le formule siano in **forma normale congiuntiva** (CNF), ovvero consistano in una congiunzione di clausole, ognuna delle quali è formata da una disgiunzione di letterali.⁶ I letterali possono contenere variabili, che sono sempre considerate universalmente quantificate. Per esempio, la formula

$$\forall x \text{ Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Ostile}(z) \Rightarrow \text{Criminale}(x)$$

diventa in CNF

$$\neg \text{Americano}(x) \vee \neg \text{Arma}(y) \vee \neg \text{Vende}(x, y, z) \vee \neg \text{Ostile}(z) \vee \text{Criminale}(x).$$

Ogni formula della logica del primo ordine può essere convertita in una formula CNF inferenzialmente equivalente. In particolare, la formula CNF non sarà soddisfacibile esattamente nei casi in cui non lo era quella originale: questo ci dà la possibilità di usare le formule CNF per effettuare dimostrazioni per assurdo.

La procedura per convertire una formula in CNF è molto simile a quella che abbiamo visto a pag. 277 per il caso proposizionale. La differenza principale sorge dalla necessità di eliminare i quantificatori esistenziali. Illustreremo la procedura traducendo la formula “chiunque ami tutti gli animali è amato da qualcuno”:

$$\forall x [\forall y \text{ Animale}(y) \Rightarrow \text{Ama}(x, y)] \Rightarrow [\exists y \text{ Ama}(y, x)].$$

I passi sono i seguenti.

- ♦ **Eliminazione delle implicazioni:**

$$\forall x [\neg \forall y \neg \text{Animale}(y) \vee \text{Ama}(x, y)] \vee [\exists y \text{ Ama}(y, x)].$$

- ♦ **Spostamento all'interno delle negazioni:** oltre alle regole consuete sui connettivi negati, ce ne servono altre per i quantificatori negati. Quindi

$$\begin{array}{lll} \neg \forall x p & \text{diventa} & \exists x \neg p \\ \neg \exists x p & \text{diventa} & \forall x \neg p. \end{array}$$

⁶ Come mostriamo nell'Esercizio 7.12, una clausola può anche essere rappresentata da un'implicazione formata da una congiunzione di atomi a sinistra e una disgiunzione di atomi a destra. Quando è scritta con un simbolo di implicazione che va da destra a sinistra questa prende il nome di **forma di Kowalski** (Kowalski, 1979b) e spesso è molto più facile da leggere.

La nostra formula passa attraverso le seguenti trasformazioni:

$$\forall x \ [\exists y \ \neg(\neg\text{Animale}(y) \vee \text{Ama}(x, y))] \vee [\exists y \ \text{Ama}(y, x)] .$$

$$\forall x \ [\exists y \ \neg\neg\text{Animale}(y) \wedge \neg\text{Ama}(x, y)] \vee [\exists y \ \text{Ama}(y, x)] .$$

$$\forall x \ [\exists y \ \text{Animale}(y) \wedge \neg\text{Ama}(x, y)] \vee [\exists y \ \text{Ama}(y, x)] .$$

Notate come il quantificatore universale ($\forall y$) nella premessa dell'implicazione sia diventato un quantificatore esistenziale. Ora la formula si legge "O esiste un animale che x non ama, oppure (se questo non è il caso) qualcuno ama x ". È chiaro che il significato originale della formula è stato preservato.

- ♦ **Standardizzazione delle variabili:** in formule come $(\forall x P(x)) \vee (\exists x Q(x))$, che usano due volte lo stesso nome di variabile, si deve cambiare il nome di una delle due. Questo ci permetterà di evitare confusione più tardi, quando dovremo eliminare i quantificatori. Risulta così

$$\forall x \ [\exists y \ \text{Animale}(y) \wedge \neg\text{Ama}(x, y)] \vee [\exists z \ \text{Ama}(z, x)] .$$

Skolemizzazione

- ♦ **Skolemizzazione:** è il processo di rimozione dei quantificatori esistenziali per eliminazione. Nel caso più semplice si riduce alla regola di istanziazione esistenziale del Paragrafo 9.1: $\exists x \ \tilde{P}(x)$ si traduce in $P(A)$, dove A è una nuova costante. Se applichiamo questa regola alla nostra formula, tuttavia, otteniamo

$$\forall x \ [\text{Animale}(A) \wedge \neg\text{Ama}(x, A)] \vee \text{Ama}(B, x)$$

che ha un significato totalmente sbagliato: infatti afferma che ogni individuo non ama un particolare animale A , oppure è a sua volta amato da una qualche specifica entità B . In realtà, la formula originale permetteva a ogni individuo di non amare un animale diverso, o di essere amata da una persona diversa. Quindi le entità di Skolem devono dipendere da x :

$$\forall x \ [\text{Animale}(F(x)) \wedge \neg\text{Ama}(x, F(x))] \vee \text{Ama}(G(x), x) .$$

funzioni di Skolem

F e G sono dette **funzioni di Skolem**. Come regola generale, gli argomenti di una funzione di Skolem sono tutte le variabili universalmente quantificate all'interno del cui campo d'azione appare un quantificatore esistenziale. Come nel caso della istanziazione esistenziale, la formula Skolemizzata è soddisfacibile esattamente nei casi in cui lo è quella originale.

- ♦ **Omissione dei quantificatori universali:** a questo punto, tutte le variabili restanti devono essere universalmente quantificate. Inoltre, la formula è equivalente a una in cui tutti i quantificatori universali sono stati spostati a sinistra. Possiamo quindi omettere i quantificatori:

$$[\text{Animale}(F(x)) \wedge \neg\text{Ama}(x, F(x))] \vee \text{Ama}(G(x), x) .$$

- ♦ **Distribuzione di \vee su \wedge :**

$$[\text{Animale}(F(x)) \vee \text{Ama}(G(x), x)] \wedge [\neg\text{Ama}(x, F(x)) \vee \text{Ama}(G(x), x)] .$$

Quest'ultimo passo potrebbe anche richiedere l'appiattimento di congiunzioni e disgiunzioni nidificate.

Ora la formula è in CNF ed è composta da due clausole. Notate che è abbastanza illeggibile (un primo aiuto consisterebbe nello specificare che la funzione di Skolem $F(x)$ si riferisce a un animale potenzialmente non amato da x , mentre $G(x)$ si riferisce a qualcuno che potrebbe amare x). Fortunatamente, gli esseri umani hanno raramente bisogno di esaminare direttamente le formule CNF; la traduzione si può facilmente automatizzare.

La risoluzione come regola di inferenza

La regola di risoluzione per le clausole del primo ordine è semplicemente una versione “sollevata” tramite lifting di quella che abbiamo già presentato a pag. 275 per il caso proposizionale. Due clausole, che grazie alla standardizzazione separata presumiamo sempre non avere alcuna variabile in comune, possono essere risolte se contengono letterali complementari. I letterali proposizionali sono complementari se uno è la negazione dell’altro; quelli di primo ordine lo sono se uno *unifica con* la negazione dell’altro. Quindi abbiamo

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m_1 \vee \dots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

dove $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. Ad esempio, possiamo risolvere le due clausole

$$[\text{Animale}(F(x)) \vee \text{Ama}(G(x), x)] \quad \text{e} \quad [\neg \text{Ama}(u, v) \vee \neg \text{Uccide}(u, v)]$$

eliminando i letterali complementari $\text{Ama}(G(x), x)$ e $\neg \text{Ama}(u, v)$, con unificatore $\theta = \{u/G(x), v/x\}$ e producendo la clausola **risolvente**

$$[\text{Animale}(F(x)) \vee \neg \text{Uccide}(G(x), x)] .$$

La regola che abbiamo appena formulato si chiama **risoluzione binaria**, perché risolve esattamente due letterali. La regola di risoluzione binaria, di per sé, non fornisce una procedura di inferenza completa: per questo si devono risolvere tutti i sottoinsiemi di letterali unificabili di ogni clausola. Un approccio alternativo consiste nell'estendere alla logica del primo ordine la **fattorizzazione**, ovvero la rimozione di letterali ridondanti. Nella fattorizzazione proposizionale due letterali sono ridotti a uno se sono *identici*; in quella del primo ordine i letterali sono ridotti se sono *unificabili*. L'unificatore dev'essere applicato all'intera clausola. La combinazione di risoluzione binaria e fattorizzazione è completa.

risoluzione binaria

Alcuni esempi di dimostrazione

La risoluzione dimostra che $KB \models \alpha$ provando che $KB \wedge \neg\alpha$ non è soddisfacibile: per far questo deve derivare la clausola vuota. L'approccio algoritmico è identico a quello del caso proposizionale, che abbiamo descritto nella Figura 7.12, ragion per cui non lo ripeteremo qui: al suo posto forniremo un paio di esempi. Il primo riguarda il colonnello criminale introdotto nel Paragrafo 9.3. Le formule in CNF sono

$$\begin{aligned} & \neg\text{Americano}(x) \vee \neg\text{Arma}(y) \vee \neg\text{Vende}(x, y, z) \vee \neg\text{Ostile}(z) \vee \text{Criminale}(x) . \\ & \neg\text{Missile}(x) \vee \neg\text{Possiede}(\text{Nono}, x) \vee \text{Vende}(\text{West}, x, \text{Nono}) . \\ & \neg\text{Nemico}(x, \text{America}) \vee \text{Ostile}(x) . \\ & \neg\text{Missile}(x) \vee \text{Arma}(x) . \\ & \text{Possiede}(\text{Nono}, M_1) . \quad \text{Missile}(M_1) . \\ & \text{Americano}(\text{West}) . \quad \text{Nemico}(\text{Nono}, \text{America}) . \end{aligned}$$

Abbiamo anche incluso l'obiettivo negato: $\neg\text{Criminale}(\text{West})$. La dimostrazione per risoluzione è mostrata nella Figura 9.11. Notate la sua struttura: una singola linea di inferenze che comincia con la clausola obiettivo e continua a risolverla con clausole presenti nella base di conoscenza finché non viene generata la clausola vuota. Questa è la struttura caratteristica della risoluzione quando la base di conoscenza è composta da clausole di Horn. In effetti, le clausole lungo la linea principale corrispondono *esattamente* ai valori assunti consecutivamente dalla variabile *obiettivo* nell'algoritmo di concatenazione all'indietro dalla Figura 9.6. Questo è dovuto al fatto che abbiamo sempre scelto di risolvere con una clausola il cui letterale positivo si unifica con quello più a sinistra della clausola "corrente" sulla linea

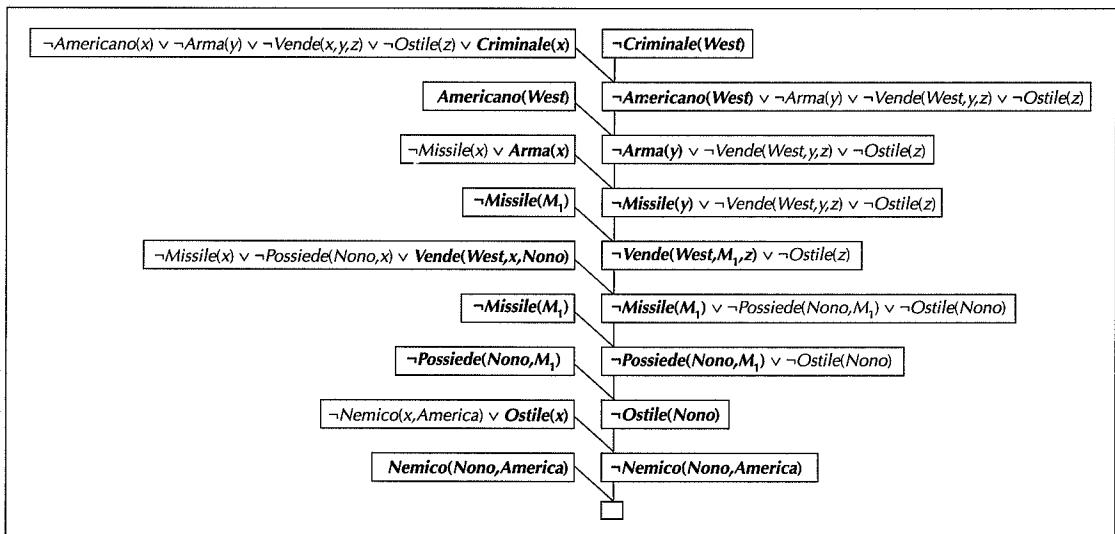


Figura 9.11 Una dimostrazione per risoluzione che il Colonnello West è un criminale.

principale della dimostrazione; questo è esattamente ciò che accade nella concatenazione all'indietro. Possiamo quindi dire che la stessa concatenazione all'indietro è di fatto solo un caso speciale di risoluzione che adotta una particolare strategia di controllo per decidere l'ordine di esecuzione delle risoluzioni.

Il nostro secondo esempio utilizza la Skolemizzazione e coinvolge clausole che non sono definite. Questo fa sì che la struttura della dimostrazione sia un po' più complessa. In linguaggio naturale il problema è il seguente:

- Chiunque ami tutti gli animali è amato da qualcuno.
- Chiunque uccida un animale non è amato da nessuno.
- Jack ama tutti gli animali.
- O Jack o la Curiosità hanno ucciso il gatto, che si chiama Tonno.
- La Curiosità ha ucciso il gatto?

Prima di tutto esprimiamo le formule originali, un po' di conoscenza iniziale e l'obiettivo negato G in logica del primo ordine:

- A. $\forall x [\forall y \text{Animale}(y) \Rightarrow \text{Ama}(x, y)] \Rightarrow [\exists y \text{Ama}(y, x)]$
- B. $\forall x [\exists y \text{Animale}(y) \wedge \text{Uccide}(x, y)] \Rightarrow [\forall z \neg \text{Ama}(z, x)]$
- C. $\forall x \text{Animale}(x) \Rightarrow \text{Ama}(\text{Jack}, x)$
- D. $\text{Uccide}(\text{Jack}, \text{Tonno}) \vee \text{Uccide}(\text{Curiosità}, \text{Tonno})$
- E. $\text{Gatto}(\text{Tonno})$
- F. $\forall x \text{Gatto}(x) \Rightarrow \text{Animale}(x)$
- $\neg G. \neg \text{Uccide}(\text{Curiosità}, \text{Tonno})$

Ora applichiamo la procedura per convertire ogni formula in CNF:

- A1. $\text{Animale}(F(x)) \vee \text{Ama}(G(x), x)$
- A2. $\neg \text{Ama}(x, F(x)) \vee \text{Ama}(G(x), x)$
- B. $\neg \text{Animale}(y) \vee \neg \text{Uccide}(x, y) \vee \neg \text{Ama}(z, x)$
- C. $\neg \text{Animale}(x) \vee \text{Ama}(\text{Jack}, x)$
- D. $\text{Uccide}(\text{Jack}, \text{Tonno}) \vee \text{Uccide}(\text{Curiosità}, \text{Tonno})$
- E. $\text{Gatto}(\text{Tonno})$
- F. $\neg \text{Gatto}(x) \vee \text{Animale}(x)$
- $\neg G. \neg \text{Uccide}(\text{Curiosità}, \text{Tonno})$

La dimostrazione per risoluzione che la Curiosità ha ucciso il gatto è fornita nella Figura 9.12. In linguaggio naturale si potrebbe parafrasarla così:

Supponiamo che la Curiosità non abbia ucciso Tonno. Sappiamo che Jack o la Curiosità l'hanno fatto; quindi dev'essere stato Jack. Ora, Tonno è un gatto e i gatti sono animali, quindi anche Tonno è un animale. Dato che chiunque uccida un animale non è amato da nessuno, sappiamo che nessuno ama Jack. D'altra parte, Jack ama tutti gli animali, per cui qualcuno lo ama; quindi abbiamo una contraddizione. Ne consegue che è stata la Curiosità a uccidere il gatto.

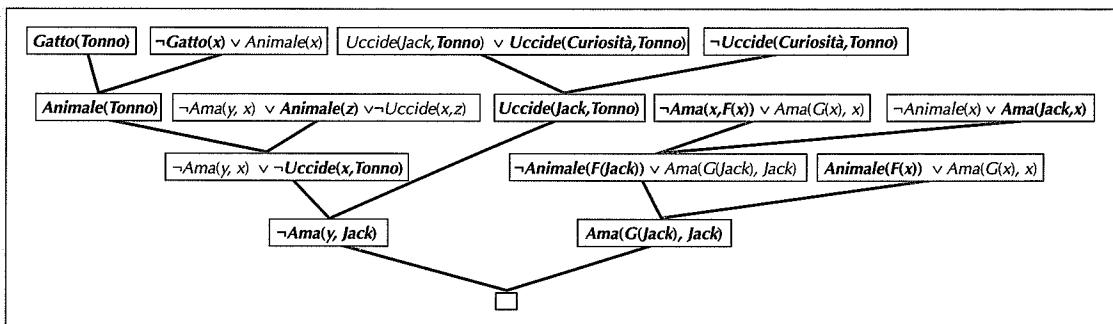


Figura 9.12 Una dimostrazione per risoluzione che la Curiosità ha ucciso il gatto. Notate l'uso della fattorizzazione nella derivazione della clausola $Ama(G(Jack), Jack)$.

Questa dimostrazione risponde alla domanda “la Curiosità ha ucciso il gatto?”, ma spesso vorremmo porre domande più generali, come “chi ha ucciso il gatto?”. La risoluzione può rispondere anche a queste interrogazioni, ma il compito è più faticoso. L’obiettivo è $\exists w Uccide(w, Tonno)$ che, una volta negato, diventa in CNF $\neg Uccide(w, Tonno)$. Ripetendo la dimostrazione della Figura 9.12 con il nuovo obiettivo negato otteniamo un albero simile, che comprende in uno dei passi la sostituzione $\{w/Curiosità\}$. In questo caso, quindi, appurare chi ha ucciso il gatto è solo questione di tener traccia dei legami delle variabili comprese nell’interrogazione.

Sfortunatamente, per obiettivi esistenziali la risoluzione può produrre **dimostrazioni non costruttive**. Ad esempio, $\neg Uccide(w, Tonno)$ risolve con $Uccide(Jack, Tonno) \vee Uccide(Curiosità, Tonno)$ per dare $Uccide(Jack, Tonno)$, che risolve ancora con $\neg Uccide(w, Tonno)$ per dare la clausola vuota. Notate che w nella dimostrazione ha due legami differenti; la risoluzione ci sta dicendo che sì, in effetti qualcuno ha ucciso Tonno: Jack o la Curiosità. Non è certo una gran sorpresa! Un modo di ovviare a questo inconveniente è restringere i passi di inferenza consentiti in modo che le variabili dell’interrogazione possano essere legate una sola volta in una data dimostrazione; a questo punto dovremo avere un meccanismo che ci permetta di fare il backtracking per annullare alcuni legami quando necessario. Un’altra possibilità è aggiungere uno speciale **letterale di risposta** all’obiettivo negato, che diventa quindi $\neg Uccide(w, Tonno) \vee Risposta(w)$. Ora il processo di risoluzione può fornire una soluzione non appena viene generata una clausola che contiene *un solo* letterale di risposta. Per la dimostrazione della Figura 9.12, questo sarebbe $Risposta(Curiosità) \vee Risposta(Jack)$, che non costituisce una soluzione.

dimostrazioni non costruttive

letterale di risposta

Completezza della risoluzione

In questo paragrafo dimostreremo la completezza della risoluzione. Se siete disposti a crederci sulla parola, potete saltarlo senza problemi.

Mostreremo che la procedura è **completa per la refutazione**, il che significa che se un insieme di formule è insoddisfacibile la risoluzione sarà sempre in grado di derivare una contraddizione. L'algoritmo non può generare tutte le conseguenze logiche di un insieme di formule, ma può sempre stabilire che una data formula è implicata da un insieme di formule. Di conseguenza può essere usato per trovare tutte le risposte a una specifica domanda, usando il metodo dell'obiettivo negato che abbiamo descritto qui sopra.

Considereremo assodato che ogni formula in logica del primo ordine (che non include l'uguaglianza) può essere riscritta come un insieme di clausole in CNF. Questo può essere dimostrato per induzione usando le formule atomiche come caso base (Davis e Putnam, 1960). Il nostro obiettivo quindi è dimostrare quanto segue: *se S è un insieme di clausole non soddisfacibile, l'applicazione su S di un numero finito di passi di risoluzione porterà a una contraddizione*.

Il nostro schema di dimostrazione segue quello originale di Robinson, con qualche semplificazione introdotta da Genesereth e Nilsson (1987). La struttura base della dimostrazione è mostrata nella Figura 9.13 e procede come segue.

1. Per prima cosa osserviamo che, se S non è soddisfacibile, esisterà un particolare insieme di *istanze ground* delle clausole di S anch'esso insoddisfacibile (teorema di Herbrand).

completa per la refutazione

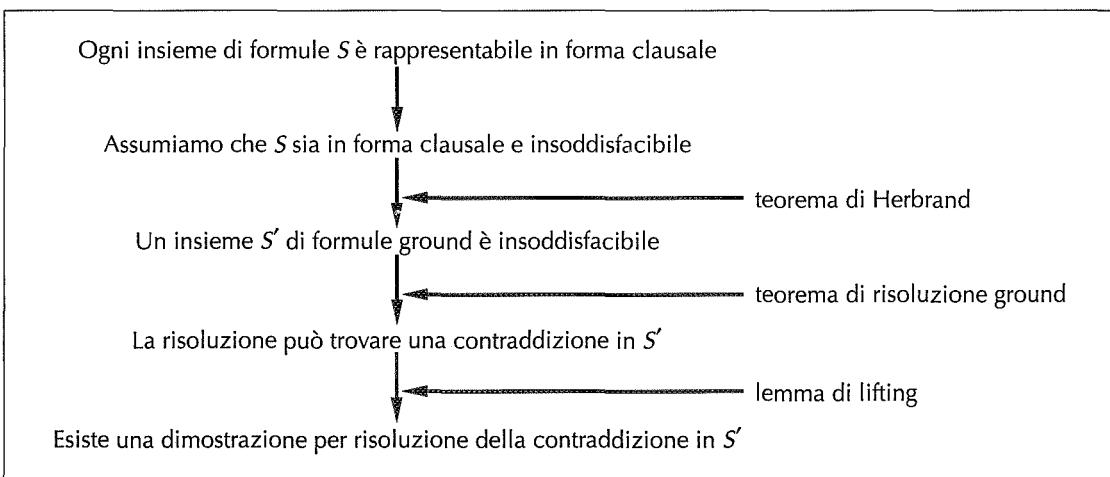


Figura 9.13 Struttura della dimostrazione di completezza della risoluzione.

2. Faremo poi ricorso al **teorema di risoluzione ground** già presentato nel Capitolo 7, che afferma che la risoluzione proposizionale è completa per le formule ground.
3. Applicheremo poi il **lemma di lifting** per mostrare che, per ogni dimostrazione per risoluzione proposizionale che usa un insieme di formule ground, esiste una corrispondente dimostrazione che usa le formule del primo ordine da cui sono state ricavate le formule ground originarie.

Per compiere il primo passo dobbiamo introdurre tre concetti nuovi.

universo di Herbrand

- ◆ **Universo di Herbrand:** se S è un insieme di clausole allora H_S , l'universo di Herbrand di S è costituito dall'insieme di tutti i termini ground che si possono costruire a partire da:

- a. I simboli di funzione in S , se ve ne sono.
- b. I simboli di costante in S , se ve ne sono; in caso contrario, il simbolo di costante A .

Per esempio, se S contiene solo la clausola $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, H_S è costituito dal seguente insieme infinito di termini ground:

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \dots\}.$$

saturazione

- ◆ **Saturazione:** se S è un insieme di clausole e P è un insieme di termini ground allora $P(S)$, la saturazione di S rispetto a P , è l'insieme delle clausole ground ottenute applicando tutte le possibili sostituzioni consistenti di termini ground in P alle variabili in S .

base di Herbrand

- ◆ **Base di Herbrand:** la saturazione di un insieme S di clausole rispetto al suo universo di Herbrand prende il nome di base di Herbrand di S e si scrive $H_S(S)$. Ad esempio, se S contiene solo la clausola che abbiamo scritto qui sopra, $H_S(S)$ sarà costituito dall'insieme infinito di clausole

$$\begin{aligned} &\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B), \\ &\quad \neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B), \\ &\quad \neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B), \\ &\quad \neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \dots\} \end{aligned}$$

teorema di Herbrand

Queste definizioni ci permettono di formulare una versione del **teorema di Herbrand** (Herbrand, 1930):

Se un insieme S di clausole è insoddisfacibile, allora esiste un sottoinsieme finito di $H_S(S)$ anch'esso insoddisfacibile.

Chiamiamo S' questo sottoinsieme finito di formule ground. Ora possiamo sfruttare il teorema di risoluzione ground (v. pag. 279) per dimostrare che la **chiusura della risoluzione** $RC(S')$ contiene la clausola vuota: questo significa che eseguire la risoluzione proposizionale su S' fino al completamento deriverà una contraddizione.

Ora che abbiamo stabilito che esiste sempre una dimostrazione per risoluzione che richiede un sottoinsieme finito della base di Herbrand di S , il passo successivo è mostrare che esiste una dimostrazione per risoluzione che usa le clausole di S stesso, che non sono necessariamente ground. Cominceremo con il considerare una singola applicazione della regola di risoluzione. Il lemma fondamentale di Robinson implica il seguente fatto:

Siano C_1 e C_2 due clausole senza variabili in comune, e siano C'_1 e C'_2 istanze ground di C_1 e C_2 . Se C' è risolvente di C'_1 e C'_2 , allora esiste una clausola C tale che (1) C è risolvente di C_1 e C_2 e (2) C' è un'istanza ground di C .

Questo è chiamato **lemma di lifting**, perché “solleva” un passo di dimostrazione da clausole ground a clausole generali del primo ordine. Per dimostrare il lemma di lifting, Robinson dovette inventare l'unificazione e derivare tutte le proprietà degli unificatori più generali. Invece di ripetere qui la dimostrazione, ci limiteremo a illustrare il lemma:

$$\begin{aligned} C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\ C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\ C'_1 &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C'_2 &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\ C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B). \end{aligned}$$

Vediamo che C' è effettivamente un'istanza ground di C . In generale, affinché C'_1 e C'_2 abbiano dei risolventi devono essere costruiti applicando per prima cosa a C_1 e C_2 l'unificatore più generale di una coppia di letterali complementari in C_1 e C_2 . Dal lemma di lifting è facile derivare un'asserzione simile riguardo a una sequenza qualsiasi di applicazioni della regola di risoluzione:

Per ogni clausola C' nella chiusura della risoluzione di S' c'è una clausola C nella chiusura della risoluzione di S tale che C' sia un'istanza ground di C e che la derivazione di C abbia la stessa lunghezza di quella di C' .

Da questo fatto segue che se la clausola vuota compare nella chiusura della risoluzione di S' deve comparire anche nella chiusura della risoluzione di S : infatti la clausola vuota non può essere un'istanza ground di alcuna altra clausola. Ricapitolando, abbiamo dimostrato che se S non è soddisfacibile, allora esiste una derivazione finita della clausola vuota mediante risoluzione.

lemma di lifting

Il lifting da clausole ground a clausole del primo ordine rappresenta un grande incremento di potenza nella dimostrazione dei teoremi. Questo deriva dal fatto che nella logica del primo ordine si devono istanziare variabili solo quando la dimostrazione lo richiede, mentre i metodi basati su clausole ground devono esaminare un numero enorme di istanziazioni arbitrarie.

Gestire l'uguaglianza

Nessuno dei metodi di inferenza descritti fin qui è in grado di gestire l'uguaglianza. Per far questo si possono adottare tre approcci distinti. Il primo consiste nell'assiomatizzare l'uguaglianza, aggiungendo alla base di conoscenza formule che la riguardano. Si deve specificare che è riflessiva, simmetrica e transitiva, e dobbiamo anche dire che in ogni predicato o funzione possiamo sostituire uguale per uguale. Sono necessari quindi tre assiomi base, più uno per ogni predicato e funzione:

$$\begin{aligned} & \forall x \ x = x \\ & \forall x, y \ x = y \Rightarrow y = x \\ & \forall x, y, z \ x = y \wedge y = z \Rightarrow x = z \\ & \forall x, y \ x = y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\ & \forall x, y \ x = y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\ & \vdots \\ & \forall w, x, y, z \ w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z)) \\ & \forall w, x, y, z \ w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z)) \\ & \vdots \end{aligned}$$

Date queste formule, una procedura di inferenza standard come la risoluzione può eseguire compiti che richiedono di ragionare sull'uguaglianza, come il calcolo di equazioni matematiche.

Un altro modo di gestire l'uguaglianza è aggiungere un'altra regola di inferenza. La più semplice è la **demodulazione**, che prende una clausola unitaria $x = y$ e sostituisce y con un qualsiasi termine che unifica con x in qualche altra clausola. Più formalmente, possiamo scrivere

- ◆ **demodulazione:** dati tre termini x, y e z , dove $\text{UNIFY}(x, z) = \theta$ e $m_n[z]$ è un letterale che contiene z :

$$\frac{x = y, \quad m_1 \vee \dots \vee m_n[z]}{m_1 \vee \dots \vee m_n[\text{SUBST}(\theta, y)]}.$$

demodulazione

Tipicamente la demodulazione è utilizzata per semplificare le espressioni usando collezioni di asserzioni come $x + 0 = x$, $x^1 = x$ e così via. La regola può essere estesa per gestire anche clausole non unitarie in cui appare un letterale di uguaglianza:

IL TEOREMA DI INCOMPLETEZZA DI GÖDEL

Estendendo leggermente il linguaggio della logica del primo ordine in modo da consentire l'uso dello **schema di induzione matematica** in aritmetica, Gödel è stato in grado di mostrare, nel suo **teorema di incompletezza**, che esistono formule aritmetiche vere che non possono essere dimostrate.

La dimostrazione del teorema di incompletezza va decisamente oltre lo scopo di questo libro, dato che è lunga almeno 30 pagine, ma possiamo farvi un accenno. Cominciamo dalla teoria dei numeri: in questa teoria c'è una sola costante, 0, e una sola funzione, S (la funzione successore). Nel modello inteso, $S(0)$ indica 1, $S(S(0))$ 2, e così via; il linguaggio quindi ha un nome per ogni numero naturale. Il vocabolario include anche i simboli di funzione $+$, \times e Expt (esponenziale) nonché il consueto insieme di connettivi logici e quantificatori. Il primo passo è notare che l'insieme di formule che possiamo scrivere può essere enumerato: immaginate infatti di ordinare alfabeticamente i simboli e poi scrivere in ordine alfabetico le formule di lunghezza 1, poi quelle di lunghezza 2 e così via. A questo punto sarà possibile numerare ogni formula α con un numero naturale distinto $\#\alpha$ (il **numero di Gödel**). Questo punto è cruciale: la teoria dei numeri contiene un nome per ognuna delle sue formule. In modo analogo possiamo numerare ogni possibile dimostrazione P con un numero di Gödel $G(P)$, perché una dimostrazione non è altro che una sequenza finita di formule.

Ora supponiamo di avere un insieme ricorsivamente enumerabile A di formule che rappresentano affermazioni vere riguardanti i numeri naturali. Ricordando che alle formule di A si può fare riferimento per mezzo di un insieme preciso di interi, possiamo immaginare di scrivere in linguaggio naturale una formula $\alpha(j, A)$ come la seguente:

$\forall i \ i$ non è il numero di Gödel di una dimostrazione della formula il cui numero di Gödel è j , ove la dimostrazione sfrutta solo premesse contenute in A .

Sia poi σ la formula $\alpha(\#\sigma, A)$, ovvero la formula che afferma la propria stessa indimostrabilità da A (questa formula esiste sempre, anche se la cosa può non apparire evidente).

Ora possiamo formulare la seguente, ingegnosa argomentazione: supponiamo che σ sia effettivamente dimostrabile partendo da A ; allora σ è falsa (dato che σ stessa afferma che non può essere dimostrata). Ma allora abbiamo una formula falsa dimostrabile da A , il che significa che A non può consistere solo di formule vere, cosa che costituisce una violazione della nostra premessa. Ne consegue che σ non è dimostrabile partendo da A . Ma questo è esattamente ciò che sostiene σ stessa; quindi σ è una formula vera.

In questo modo abbiamo dimostrato (a meno di altre 29 pagine e mezzo) che per ogni insieme di formule vere della teoria dei numeri, e in particolare ogni insieme di assiomi base, ci sono altre formule vere che non possono essere dimostrate partendo da tali assiomi. Questo significa, tra l'altro, che non potremo mai dimostrare tutti i teoremi della matematica *dato un qualsiasi sistema di assiomi*. Chiaramente questa scoperta è stata molto importante per la matematica: il suo significato per quanto riguarda l'IA è stato oggetto di un acceso dibattito, a partire da alcune riflessioni di Gödel stesso. Ci occuperemo dell'argomento nel Capitolo 26.

paramodulazione

- ♦ **paramodulazione:** dati tre termini x, y e z , dove $\text{UNIFY}(x, z) = \theta$,

$$\frac{\ell_1 \vee \dots \vee \ell_k \vee x = y, \quad m_1 \vee \dots \vee m_n[z]}{\text{SUBST}(\theta, \ell_1 \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_n[y])}.$$

A differenza della demodulazione, la paramodulazione rappresenta una procedura di inferenza completa e comprensiva di uguaglianza per la logica del primo ordine.

Un terzo approccio delega completamente il ragionamento sull'uguaglianza a un algoritmo esteso di unificazione. Questo significa che i termini sono unificabili se è *dimostrabile* che sono uguali sotto qualche sostituzione, dove la locuzione “dimostrabile” permette un certo grado di ragionamento sull'uguaglianza. Ad esempio, i termini $1 + 2$ e $2 + 1$ normalmente non sono unificabili, ma un algoritmo di unificazione che sa che $x + y = y + x$ li può unificare con una sostituzione vuota. Una **unificazione equazionale** di questo tipo può essere svolta mediante algoritmi efficienti progettati ad hoc per i particolari assiomi utilizzati (commutatività, associatività e così via) anziché ricorrere all'inferenza esplicita. I dimostratori di teoremi che sfruttano questa tecnica sono strettamente imparentati ai sistemi di programmazione logica che abbiamo descritto nel Paragrafo 9.4.

unificazione
equazionale

Strategie di risoluzione

Abbiamo assodato che una serie di applicazioni della regola di risoluzione porterà a scoprire una dimostrazione, se questa esiste. Ora esamineremo alcune strategie che aiutano a trovare una dimostrazione *in modo efficiente*.

Preferenza per le clausole unitarie

Questa strategia preferisce applicare la risoluzione quando una delle due formule è un letterale singolo (noto anche come **clausola unitaria**). Il concetto si basa sul fatto che stiamo comunque cercando di produrre una clausola vuota, ragion per cui è una buona idea preferire le inferenze che producono clausole corte. Risolvere una formula unitaria (come P) con qualsiasi altra formula (come $\neg P \vee \neg Q \vee R$) produce sempre una formula (in questo caso $\neg Q \vee R$) più corta di quella originale. La prima volta che fu applicata al caso proposizionale, nel 1964, la strategia di preferenza per le clausole unitarie portò a un incremento delle prestazioni davvero notevole, rendendo possibile dimostrare teoremi che in caso contrario non avrebbero potuto essere trattati. Di per sé, tuttavia, questa strategia non riduce il fattore di ramificazione di problemi di media grandezza abbastanza da renderli risolvibili mediante risoluzione. Rimane comunque un'euristica utile che può essere combinata con altre strategie.

risoluzione unitaria

La **risoluzione unitaria** è una forma ristretta di risoluzione in cui ogni passo deve obbligatoriamente coinvolgere una clausola unitaria. In generale non è completa, ma lo diventa per le basi di conoscenza di Horn: in quest'ultimo caso diventa molto simile a una concatenazione in avanti.

Insieme di supporto

Le euristiche che consentono di provare prima certe tipologie di risoluzione sono utili, ma in generale è più efficace cercare di eliminarne alcune del tutto: la strategia basata sull'insieme di supporto fa proprio questo. Per prima cosa si deve identificare un sottoinsieme di formule, che prende appunto il nome di **insieme di supporto**. Ogni risoluzione coinvolgerà una formula presa dall'insieme di supporto e un'altra e aggiungerà la clausola risolvente allo stesso insieme di supporto. Se l'insieme è piccolo rispetto all'intera base di conoscenza, lo spazio di ricerca risulterà notevolmente ridotto.

insieme di supporto

Quest'approccio richiede molta cautela, perché una scelta errata dell'insieme renderà l'algoritmo incompleto. Comunque, scegliere l'insieme di supporto S in modo tale che le formule restanti siano collettivamente soddisfacibili assicura una risoluzione completa. Un approccio diffuso è scegliere come insieme di supporto la query negata, partendo dall'assunto che la base di conoscenza originale sia consistente (del resto, se così non fosse sarebbe del tutto inutile dimostrare che implica la query). Questa strategia ha il vantaggio aggiuntivo di generare alberi di dimostrazione particolarmente leggibili per gli esseri umani, dato che sono guidati dall'obiettivo.

Risoluzione di input

Nella strategia di **risoluzione di input**, ogni risoluzione combina una formula di input (dalla KB o dalla query) con un'altra. La dimostrazione riportata nella Figura 9.11 usa solo risoluzioni di input e ha la caratteristica struttura formata da una "spina dorsale" a cui si attaccano singole formule. È chiaro che lo spazio degli alberi di dimostrazione che hanno questa struttura è più piccolo dello spazio di tutti i grafi di dimostrazione. Nelle basi di conoscenza di Horn, il Modus Ponens è un tipo di strategia di risoluzione di input, perché combina un'implicazione dalla KB originale con qualche altra formula. Non sorprende quindi che la risoluzione di input sia completa per le basi di conoscenza in forma di Horn, ma incompleta nel caso generale. La strategia di **risoluzione lineare** è una lieve generalizzazione che permette di risolvere insieme P e Q se P appartiene alla KB originale oppure se P è un antenato di Q nell'albero di dimostrazione. La risoluzione lineare è completa.

risoluzione di input

Sussunzione

Il metodo della **sussunzione** elimina tutte le formule che sono sussunte da (ovvero più specifiche di) una formula esistente nel KB. Ad esempio, se $P(x)$ è nel KB, non c'è alcuna ragione di aggiungere $P(A)$ e ha ancor meno senso aggiungere $P(A) \vee Q(B)$. La sussunzione aiuta a mantenere piccola la KB e lo spazio di ricerca.

sussunzione

risoluzione lineare

Dimostratori di teoremi

I dimostratori di teoremi (conosciuti anche come sistemi automatici di ragionamento) differiscono dalla programmazione logica in due aspetti. Prima di tutto, la maggior parte dei linguaggi di programmazione logica accetta solo clausole di Horn, mentre i dimostratori di teoremi possono gestire l'intera logica del primo ordine. In secondo luogo, i programmi Prolog mescolano logica e controllo: infatti la scelta del programmatore di scrivere $A :- B, C$ invece di $A :- C$, B ha un impatto sull'esecuzione del programma. Nella maggior parte dei dimostratori di teoremi, la struttura sintattica delle formule non modifica i risultati. Per funzionare in modo efficiente i dimostratori di teoremi hanno ancora bisogno di informazione di controllo, che però viene normalmente tenuta distinta dalla base di conoscenza e non fa direttamente parte della sua rappresentazione. La maggior parte della ricerca nel campo dei dimostratori di teoremi riguarda lo studio di strategie di controllo che oltre ad aumentare la velocità di esecuzione sono utili in generale.

Progettazione di un dimostratore di teoremi

In questa sezione descriveremo il dimostratore di teoremi OTTER (acronimo di Organized Techniques for Theorem-proving and Effective Research, McCune, 1992), con particolare attenzione alla sua strategia di controllo. Preparando un problema per OTTER, l'utente deve dividere la conoscenza in quattro parti.

- ◆ Un insieme di clausole noto come **insieme di supporto** (*o ids*), che definisce i fatti importanti del problema. Ogni passo di risoluzione risolve un elemento dell'insieme di supporto con un altro assioma, in modo tale che la ricerca rimanga focalizzata sull'*ids*.
- ◆ Un insieme di **assiomi usabili** al di fuori dell'insieme di supporto, che forniscono conoscenza aggiuntiva del dominio del problema. Il confine tra ciò che è parte del problema (e quindi parte dell'*ids*) e le informazioni di contorno (quindi assioma usabile) è una decisione dell'utente.
- ◆ Un insieme di equazioni che prende il nome di **riscrittura** o **demodulatori**. Benché i demodulatori siano equazioni, sono sempre applicati da sinistra a destra. Di conseguenza definiscono una forma canonica in cui saranno semplificati tutti i termini. Ad esempio, il demodulatore $x + 0 = x$ afferma che ogni termine nella forma $x + 0$ dev'essere rimpiazzato dal termine x .
- ◆ Un insieme di parametri e clausole che definiscono la strategia di controllo. In particolare, l'utente specifica una funzione euristica per il controllo della ricerca e una funzione di filtraggio per eliminare i sotto-objettivi considerati non interessanti.

OTTER lavora risolvendo a ogni passo un elemento dell'insieme di supporto con uno degli assiomi usabili. A differenza di Prolog sfrutta una forma di ricerca best-first. La sua funzione euristica misura il “peso” di ogni clausola, prediligendo le clausole più leggere. La scelta esatta dell'euristica è lasciata all'utente, ma general-

mente il peso di una clausola è legato alla sua dimensione o difficoltà. Le clausole unitarie sono leggere; la ricerca può quindi essere considerata una generalizzazione della strategia che preferisce le clausole unitarie. A ogni passo, OTTER sposta la clausola “più leggera” dall’insieme di supporto alla lista di assiomi usabili e aggiunge all’insieme di supporto alcune delle conseguenze immediate della risoluzione della clausola più leggera con elementi della lista usabile. OTTER si arresta quando trova una refutazione, oppure quando non ci sono più clausole nell’insieme di supporto. L’algoritmo è mostrato più dettagliatamente nella Figura 9.14.

procedura OTTER (*ids, usabile*)

inputs: *ids*, un insieme di supporto; clausole che definiscono il problema (var. globale)
usabile, conoscenza di fondo potenzialmente rilevante per il problema

repeat

 clausola \leftarrow il membro più leggero *dell’ids*

 sposta *clausola* da *ids* a *usabile*

 PROCESSA(INFERISCI(*clausola, usabile*), *ids*)

until *ids* = [] or è stata trovata una refutazione

function INFERISCI(*clausola, usabile*) **returns** alcune clausole

 resolvi *clausola* con ogni membro di *usabile*

return le clausole risultanti dopo l’applicazione di FILTRA

procedura PROCESSA(*clausole, ids*)

for each *clausole* **in** *clausole* **do**

clausola \leftarrow SEMPLIFICA(*clausola*)

 fondi i letterali identici

 scarta la clausola se è una tautologia

ids \leftarrow [*clausola* | *ids*]

if *clausola* non ha letterali **then** è stata trovata una refutazione

if *clausola* ha un solo letterale **then** cerca di ottenere refutazione unitaria

Figura 9.14 Schema del dimostratore di teoremi OTTER. Per scegliere la clausola “più leggera” è applicato un controllo euristico, e la funzione FILTRA fa sì che le clausole non interessanti non siano prese in considerazione.

Estensioni di Prolog

Un modo alternativo di costruire un dimostratore di teoremi è partire da un compilatore Prolog ed estenderlo in modo da ottenere un sistema di ragionamento corretto e completo per la logica del primo ordine. Questo è stato l'approccio adottato dal Prolog Technology Theorem Prover, o PTTP (Stickel, 1988). PTTP ha apportato cinque importanti cambiamenti a Prolog per recuperare la completezza e l'espressività della logica del primo ordine.

- ◆ Il controllo di occorrenza è stato inserito nuovamente nella procedura di unificazione per renderla corretta.
- ◆ La ricerca in profondità è stata rimpiazzata da una ricerca ad approfondimento iterativo. La strategia di ricerca così facendo è stata resa completa, al solo costo di un fattore temporale costante.
- ◆ Sono permessi letterali negati come $\neg P(x)$. L'implementazione include due procedure separate, una che cerca di dimostrare P e l'altra $\neg P$.
- ◆ Una clausola con n atomi è memorizzata come n regole diverse. Ad esempio, $A \Leftarrow B \wedge C$ sarebbe memorizzata come $(\neg B \Leftarrow C \wedge \neg A)$ e $(\neg C \Leftarrow B \wedge \neg A)$. Questa tecnica, nota come **locking**, fa sì che l'obiettivo corrente debba essere unificato con la sola testa di ogni clausola, ma permette ancora una gestione corretta della negazione.
- ◆ L'inferenza è resa completa (anche per clausole non in forma di Horn) mediante l'aggiunta della regola di risoluzione di input lineare: se l'obiettivo corrente unifica con la negazione di uno degli obiettivi sulla pila, tale obiettivo può considerarsi soddisfatto. Questo è un tipo di ragionamento per assurdo: supponete che l'obiettivo originale sia P e che sia ridotto a $\neg P$ da una serie di inferenze. Questo dimostrerebbe che $\neg P \Rightarrow P$, ciò che è equivalente logicamente a P .

locking

Nonostante questi cambiamenti, PTTP conserva le caratteristiche che rendono Prolog veloce. Le unificazioni sono ancora effettuate modificando direttamente le variabili e sciogliendo i legami in fase di backtracking mediante lo "srotolamento" della traccia. La strategia di ricerca è ancora basata sulla risoluzione dell'input, il che significa che ogni risoluzione viene fatta con una delle clausole appartenenti al problema originale (e non una clausola derivata). Questo rende possibile la compilazione di tutte le clausole della formulazione originale del problema.

Il limite principale di PTTP sta nel fatto che l'utente deve rinunciare a ogni controllo sulla ricerca. Ogni regola di inferenza è applicata dal sistema sia nella forma originale che in quella contrappositiva: questo può portare a ricerche non intuitive. Ad esempio, considerate

$$(f(x, y) = f(a, b)) \Leftarrow (x=a) \wedge (y=b).$$

Come regola Prolog, questo è un modo ragionevole di dimostrare che due termini f sono uguali. Ma PTT genererebbe anche la contrapposita:

$$(x \neq a) \Leftarrow (f(x, y) \neq f(a, b)) \wedge (y = b)$$

che sembra un modo inutilmente barocco di dimostrare che due termini qualsiasi x e a sono diversi.

Dimostratori di teoremi come aiutanti

Fin qui abbiamo pensato che i sistemi di ragionamento dovessero funzionare come agenti indipendenti, prendendo decisioni e scegliendo da soli come agire. I dimostratori di teoremi possono essere usati anche come aiutanti, ad esempio fornendo supporto e suggerimenti a un matematico. In questa modalità di funzionamento il matematico funge da supervisore, delineando la strategia e decidendo il passo successivo e lasciando che il dimostratore di teoremi si occupi dei dettagli. Questa soluzione rappresenta una parziale soluzione al problema della semidecidibilità, perché quando una query prende troppo tempo il supervisore può sempre cancellarla e provare un altro approccio. Un dimostratore di teoremi può anche funzionare da **verificatore di dimostrazioni**: in questo caso un utente umano deve fornire la dimostrazione sotto forma di una serie di passi relativamente ampi; il compito del sistema è fornire le inferenze richieste per dimostrare che ogni passo è corretto.

Un **ragionatore socratico** è un dimostratore di teoremi la cui funzione ASK è incompleta, ma che può sempre arrivare a una soluzione a patto che gli venga posta una giusta sequenza di domande. I ragionatori socratici quindi possono essere dei buoni aiutanti, se il supervisore è in grado di fornire le appropriate chiamate di ASK. ONTIC (McAllester, 1989) è un sistema di ragionamento socratico per matematici.

verificatore di dimostrazioni

ragionatore socratico

Utilizzi pratici dei dimostratori di teoremi

I dimostratori di teoremi sono stati capaci di scoprire nuovi risultati matematici. Il programma SAM (*Semi-Automated Mathematics*) è stato il primo, dimostrando un teorema della teoria dei reticolati (Guard et al., 1969). Il programma AURA ha trovato la soluzione di alcuni problemi aperti in diverse aree della matematica (Wos e Winker, 1983). Il dimostratore di teoremi di Boyer-Moore (Boyer e Moore, 1979) è stato esteso per molti anni e utilizzato da Natarajan Shankar per fornire la prima dimostrazione rigorosamente formale del Teorema di Incompletezza di Gödel (Shankar, 1986). Il programma OTTER è uno dei dimostratori di teoremi più potenti; è stato utilizzato per risolvere molte questioni aperte di logica combinatoria, la più famosa delle quali riguarda l'**algebra di Robbins**. Nel 1933, Herbert Robbins propose un semplice insieme di assiomi che sembravano sufficienti a definire l'algebra di Boole, ma questo non poteva essere provato (nonostante una grande mo-

algebra di Robbins

verifica
sintesi

le di lavoro da parte di molti matematici, tra cui lo stesso Alfred Tarski). Il 10 ottobre 1996, dopo otto giorni di calcoli, EQP (una versione di OTTER) trovò una dimostrazione (McCune, 1997).

I dimostratori di teoremi possono essere applicati alla verifica e alla sintesi sia dell'hardware che del software, dato che entrambi i domini possono essere automatizzati correttamente. La ricerca sulla dimostrazione automatica di teoremi, quindi, oltre che all'IA si estende anche al campo della progettazione hardware, dei linguaggi di programmazione e dell'ingegneria del software. Nel caso del software, gli assiomi esprimono le proprietà di ogni elemento sintattico del linguaggio di programmazione; ragionare sui programmi è un'attività molto simile al ragionamento sulle azioni nel calcolo delle situazioni. Un algoritmo è verificato dimostrando che i suoi output rispettano le specifiche per tutti gli input. L'algoritmo di criptazione a chiave pubblica RSA e l'algoritmo di matching di stringhe di Boyer-Moore sono stati verificati in questo modo (Boyer e Moore, 1984). Nel caso dell'hardware, gli assiomi descrivono le interazioni tra segnali ed elementi del circuito (v. Capitolo 8 per un esempio). Il progetto di un sommatore a 16 bit è stato verificato da AURA (Wojcik, 1983). Sistemi di ragionamento logico progettati specificatamente sono stati in grado di verificare la correttezza di intere CPU, incluse le loro proprietà temporali (Sriwas e Bickford, 1990).

La sintesi formale di algoritmi fu una delle prime applicazioni dei dimostratori di teoremi, come scrive Cordell Green (1969a), che ha esteso idee precedenti di Simon (1963). L'idea è dimostrare un teorema della forma “esiste un programma p che soddisfa una determinata specifica”. Se la dimostrazione è costruttiva, è possibile estrarvi lo stesso programma. Benché una versione completamente automatizzata di quella che è stata chiamata **sintesi deduttiva** non sia ancora disponibile per la programmazione generale, con l'aiuto di utenti umani è stata in grado di progettare con successo diversi algoritmi sofisticati e innovativi. La sintesi di programmi di uso speciale è un'altra area in cui la ricerca è molto attiva. Nel campo della sintesi hardware, il dimostratore di teoremi AURA è stato applicato alla progettazione di circuiti più compatti di qualsiasi precedente soluzione (Wojciechowski e Wojcik, 1983). Per molti progetti di circuiti la logica proposizionale è sufficiente, dato che l'insieme di proposizioni interessanti è prefissato dagli elementi del circuito. Oggi l'applicazione dell'inferenza proposizionale alla sintesi hardware è una tecnica standard che include molti sistemi su grande scala (v. ad es. Nowick et al., 1993).

Queste stesse tecniche cominciano ora ad essere applicate anche alla verifica software, grazie a sistemi come il model checker SPIN (Holzmann, 1997). Il programma di controllo del Remote Agent spaziale, ad esempio, è stato sottoposto a verifica sia prima che dopo il volo (Havelund et al., 2000).

sintesi deduttiva

9.6 Riepilogo

Abbiamo presentato un'analisi dell'inferenza nella logica del primo ordine e diversi algoritmi per svolgerla.

- ◆ Un primo approccio consiste nell'usare regole di inferenza per istanziare i quantificatori al fine di proposizionalizzare il problema. Tipicamente questa tecnica risulta molto lenta.
- ◆ L'uso dell'unificazione per identificare le sostituzioni appropriate delle variabili elimina il passo di istanziazione nelle dimostrazioni del primo ordine, rendendo il processo molto più efficiente.
- ◆ Una versione del Modus Ponens sollevata attraverso il lifting usa l'unificazione per dare origine a una regola di inferenza intuitiva e potente, il Modus Ponens generalizzato. Gli algoritmi di concatenazione in avanti e concatenazione all'indietro applicano questa regola a insiemi di clausole definite.
- ◆ Il Modus Ponens generalizzato è completo per le clausole definite, benché il problema dell'implicazione rimanga semidecidibile. L'implicazione è comunque decidibile nel caso di programmi Datalog, che consistono di clausole definite prive di simboli di funzione.
- ◆ La concatenazione in avanti è usata nei database deduttivi, in cui può essere combinata con le operazioni classiche dei database relazionali. È utilizzata anche nei sistemi di produzioni, che eseguono aggiornamenti efficienti anche in presenza di insiemi di regole molto grandi.
- ◆ La concatenazione in avanti è completa per programmi Datalog e richiede un tempo di esecuzione polinomiale.
- ◆ La concatenazione all'indietro è usata nei sistemi di programmazione logica come Prolog, che sfruttano sofisticati compilatori per ottenere un'inferenza molto veloce.
- ◆ La concatenazione all'indietro soffre dei problemi di inferenze ridondanti e cicli infiniti; per alleviarli si può ricorrere alla memoizzazione.
- ◆ La regola di inferenza generalizzata della risoluzione rappresenta un sistema completo di dimostrazione per la logica del primo ordine, applicabile a basi di conoscenza in forma normale congiuntiva.
- ◆ Esistono molte strategie per ridurre lo spazio di ricerca di un sistema di risoluzione senza comprometterne la completezza. Dimostratori di teoremi efficienti sono stati utilizzati per dimostrare interessanti teoremi matematici e per verificare e sintetizzare software e hardware.

sillogismo

Note storiche e bibliografiche

L'inferenza logica è stata studiata estensivamente dai matematici greci. Il tipo di inferenza studiato più attentamente da Aristotele fu il **sillogismo**. I sillogismi di Aristotele includevano qualche elemento della logica del primo ordine, come la quantificazione, ma erano limitati a predicati unari. Erano classificati per "figure" e "modi" a seconda dell'ordine dei termini (che oggi chiameremo predicati) nelle formule, il grado di generalità (che oggi esprimeremo con i quantificatori) applicato a ogni termine e la negazione o meno di ogni termine. Il sillogismo più importante è quello di primo modo e prima figura:

Tutti gli *S* sono *M*.
Tutti gli *M* sono *P*.
Quindi, tutti gli *S* sono *P*.

Aristotele cercò di dimostrare la validità di altri sillogismi "riducendoli" a quelli della prima figura, ma fu molto meno preciso nel descrivere che cosa dovesse comportare questa "riduzione" rispetto a quanto non fosse nella caratterizzazione delle stesse figure e dei modi sillogistici.

Gottlob Frege, che sviluppò la logica del primo ordine nella sua prima versione completa nel 1879, basò il suo sistema di inferenza su una grande collezione di schemi logicamente validi più una singola regola di inferenza, il Modus Ponens. Frege sfruttò il fatto che l'effetto di una regola di inferenza di forma "da *P* inferisci *Q*" può essere simulato applicando il Modus Ponens a *P* insieme a uno schema logicamente valido $P \Rightarrow Q$. Questo stile di esposizione "assiomatico", che utilizza il Modus Ponens insieme a un certo numero di schemi logicamente validi, fu impiegato da diversi studiosi di logica dopo Frege: l'esempio più notevole è quello dei *Principia Mathematica* (Whitehead e Russell, 1910).

Le regole di inferenza, intese come qualcosa di diverso dagli schemi di assiomi, furono al centro dell'approccio della **deduzione naturale**, introdotta da Gerhard Gentzen (1934) e da Stanisław Jaskowski (1934). Questo tipo di deduzione fu chiamata "naturale" perché non richiedeva la conversione in una (illeggibile) forma normale e perché le sue regole di inferenza dovevano apparire naturali agli esseri umani. Prawitz (1965) ha dedicato un intero libro alla deduzione naturale. Gallier (1986) usa l'approccio di Gentzen per esporre le basi teoriche della deduzione automatica.

L'invenzione della forma clausale rappresentò un passo fondamentale nello sviluppo dell'analisi matematica della logica del primo ordine. Whitehead e Russell (1910) formularono le cosiddette *regole di passaggio* (il termine è preso da Herbrand (1930)) per spostare i quantificatori nella parte iniziale delle formule. Le costanti e le funzioni di Skolem furono introdotte, come prevedibile, da Thoralf Skolem (1920). La procedura generale di skolemizzazione è ancora dovuta a Skolem (1928), così come l'importante nozione di universo di Herbrand.

Il teorema di Herbrand, chiamato così in onore del logico francese Jacques Herbrand (1930), ha giocato un ruolo fondamentale nello sviluppo dei metodi di ragionamento automatico, sia prima che dopo l'introduzione della risoluzione da parte di Robinson. È per questo motivo che parliamo di "universo di Herbrand", anche se è stato Skolem a inventare il concetto. Herbrand può anche essere considerato l'inventore dell'unificazione. Gödel (1930) partì dalle idee di Skolem e Herbrand per dimostrare che la logica del primo ordine ha una procedura di dimostrazione completa. Alan Turing (1936) e Alonzo Church (1936) hanno dimostrato simultaneamente, usando metodi molto diversi, che nella logica del primo ordine la validità non è decidibile. L'eccellente volume di Enderton (1972) espone tutti questi risultati in modo rigoroso ma abbastanza comprensibile.

Benché McCarthy (1958) avesse già suggerito l'uso della logica del primo ordine per la rappresentazione e il ragionamento nell'IA, i primi sistemi furono sviluppati da logici interessati alla dimostrazione di teoremi matematici. Fu Abraham Robinson a proporre l'uso della proposizionalizzazione e il teorema di Herbrand, mentre Gilmore (1960) scrisse il primo programma basato su questo approccio. Davis e Putnam (1960) usarono la forma clausale, producendo un programma che tentava di trovare refutazioni sostituendo membri dell'universo di Herbrand alle variabili per produrre clausole ground e poi cercando inconsistenze proposizionali tra di esse. Prawitz (1960) sviluppò l'idea chiave di generare termini dall'universo di Herbrand solo quando era necessario per stabilire un'inconsistenza proposizionale, e lasciare che questo guidasse il processo di ricerca. Dopo ulteriori sviluppi da parte di altri ricercatori, quest'idea portò J. A. Robinson (nessuna parentela) a sviluppare il metodo di risoluzione (Robinson, 1965). Il cosiddetto "metodo inverso", sviluppato più o meno contemporaneamente dal ricercatore sovietico S. Maslov (1964, 1967), è basato su principî leggermente diversi, ma offre analoghi vantaggi computazionali rispetto alla proposizionalizzazione. Il **metodo di connessione** di Wolfgang Bibel (1981) può essere considerato un'estensione di quest'approccio.

Dopo lo sviluppo della risoluzione, la ricerca sull'inferenza del primo ordine proseguì in molte direzioni diverse. Nell'IA, la risoluzione fu adottata per sistemi di risposta a interrogazioni da Cordell Green e Bertram Raphael (1968). Un approccio meno formale fu quello di Carl Hewitt (1969): il suo linguaggio PLANNER, benché non sia stato mai implementato completamente, fu un precursore della programmazione logica e includeva direttive per la concatenazione in avanti e all'indietro e per la negazione come fallimento. Un sottoinsieme noto come MICRO-PLANNER (Sussman e Winograd, 1970) fu implementato e utilizzato nel sistema di comprensione di linguaggio naturale SHRDLU (Winograd, 1972). Le prime implementazioni nel campo dell'IA dedicarono molti sforzi alla creazione di strutture dati che permettessero un recupero efficiente dei fatti; i frutti di questo lavoro sono presentati nei testi di programmazione per l'IA (Charniak et al., 1987; Norvig, 1992; Forbus e de Kleer, 1993).

All'inizio degli anni '70 la **concatenazione in avanti** si era affermata come un'alternativa alla risoluzione di più facile comprensione. Una grande varietà di sistemi la utilizzava, dal dimostratore di teoremi geometrici di Nevins (Nevins, 1975) al sistema esperto R1 per la configurazione VAX (McDermott, 1982). Solitamente le applicazioni di IA prevedevano un grande numero di regole, cosa che rendeva importante disporre di una tecnologia di matching efficiente, in particolar modo per gli aggiornamenti incrementali: per supportare applicazioni simili furono sviluppati i **sistemi di produzioni**. Il linguaggio per sistemi di produzioni OPS-5 (Forgy, 1981; Brownston et al., 1985) fu utilizzato per R1 e per l'architettura cognitiva SOAR (Laird et al., 1987). OPS-5 incorporava il processo di matching "rete" (Forgy, 1982). SOAR, che per memorizzare i risultati delle elaborazioni già svolte genera nuove regole, può arrivare ad averne un numero molto grande: nel caso del sistema TACAIR-SOAR per il controllo di velivoli da combattimento simulati, oltre 1.000.000 (Jones et al., 1998). CLIPS (Wygant, 1989) è un linguaggio per sistemi di produzioni sviluppato alla NASA: basato sul C, permette un'integrazione più stretta con altri sistemi software, hardware e sensori. È stato utilizzato per l'automazione dei veicoli spaziali e in molte applicazioni militari.

Anche l'area di ricerca che va sotto il nome di **database deduttivi** ha contribuito molto alla nostra comprensione dell'inferenza in avanti. La disciplina ha avuto inizio con un workshop a Tolosa nel 1977, organizzato da Jack Minker, che riunì diversi esperti di inferenza logica e di basi di dati (Gallaire e Minker, 1978). In una recente panoramica storica (Ramakrishnan and Ullman, 1995) si legge: "i [database] deduttivi sono un tentativo di adattare Prolog, che ha una visione del mondo basata su 'dati piccoli', a un mondo di 'dati grandi' ". Il suo scopo quindi è di fondere la tecnologia dei database relazionali, progettata per recuperare grandi *insiemi* di fatti, con la tecnologia di inferenza del Prolog, che tipicamente recupera un fatto alla volta. I testi sui database deduttivi includono Ullman (1989) e Ceri et al. (1990).

Importanti lavori di Chandra e Harel (1980) e di Ullman (1985) portarono all'adozione di Datalog come linguaggio standard per i database deduttivi. Divenne standard anche l'inferenza "bottom-up" dal basso in alto, ovvero la concatenazione in avanti, in parte per evitare i problemi di non terminazione e di calcolo ridondante legati alla concatenazione all'indietro, in parte per la sua naturale implementazione in termini di operazioni relazionali su database. Lo sviluppo, da parte di Bancilhon et al. (1986), della tecnica degli *insiemi magici* per la riscrittura di regole permise poi alla concatenazione in avanti di "prendere in prestito" da quella all'indietro il vantaggio di essere guidata dall'obiettivo. In favore di quest'ultima entrarono in gioco i metodi di programmazione logica con tabelle (v. sotto), che prendono dalla concatenazione in avanti il vantaggio della programmazione dinamica.

Gran parte della nostra comprensione della complessità dell'inferenza logica proviene dalla comunità dei database deduttivi. Chandra e Merlin (1977) hanno dimostrato per primi che il matching di una singola regola non ricorsiva (nella ter-

minologia dei database, una **query congiuntiva**) può essere NP-difficile. Kuper e Vardi (1993) hanno proposto la nozione di **complessità dei dati**, ovvero un calcolo della complessità in funzione della dimensione del database tenendo costante la dimensione delle regole, come misura adeguata per indicare quanto è difficile rispondere alle interrogazioni. Gottlob et al. (1999b) hanno discusso la relazione tra le query congiuntive e il soddisfacimento di vincoli, mostrando come la scomposizione di iperalbero può ottimizzare il processo di matching.

Come abbiamo già menzionato, la **concatenazione all'indietro** fu applicata all'inferenza logica nel linguaggio PLANNER di Hewitt (1969). La programmazione logica in sé, tuttavia, fu sviluppata indipendentemente. Una forma ristretta di risoluzione lineare, chiamata **risoluzione SL**, fu sviluppata da Kowalski e Kuehner (1971) partendo dalla tecnica di **eliminazione dei modelli** di Loveland (1968); applicata alle clausole definite prende il nome di **risoluzione SLD**, che si presta all'interpretazione di clausole definite come fossero programmi (Kowalski, 1974, 1979a, 1979b). Intanto, nel 1972, il ricercatore francese Alain Colmerauer aveva sviluppato e implementato **Prolog** con lo scopo di riconoscere il linguaggio naturale: in effetti le clausole Prolog inizialmente rappresentavano regole di una grammatica non contestuale (Roussel, 1975; Colmerauer et al., 1973). Gran parte dei fondamenti teorici della programmazione logica furono sviluppati da Kowalski insieme a Colmerauer. La definizione della semantica che usa i minimi punti fissi è dovuta a Van Emden e Kowalski (1976). Kowalski (1988) e Cohen (1988) forniscono buone panoramiche storiche delle origini del Prolog. *Foundations of Logic Programming* (Lloyd, 1987) è un'analisi teorica dei meccanismi alla base di Prolog e di altri linguaggi di programmazione logica.

I compilatori Prolog più efficienti sono generalmente basati sul modello della Warren Abstract Machine (WAM), sviluppato da David H. D. Warren (1983). Van Roy (1990) ha mostrato che l'applicazione di tecniche aggiuntive di compilazione, come l'inferenza dei tipi, rendono la velocità dei programmi Prolog competitiva con il C. Il progetto giapponese della Quinta Generazione, uno sforzo decennale cominciato nel 1982, si basava completamente su Prolog per sviluppare sistemi intelligenti.

Metodi per evitare cicli superflui nei programmi logici ricorsivi furono sviluppati indipendentemente da Smith et al. (1986) e Tamaki e Sato (1986). Quest'ultimo lavoro includeva anche il concetto di memoizzazione per i programmi logici, in seguito sviluppato estensivamente da David S. Warren sotto il nome di **programmazione logica con tabelle**. Swift e Warren (1994) mostrarono come estendere la WAM per gestire le tabelle, permettendo ai programmi Datalog di girare un ordine di grandezza più velocemente dei database deduttivi basati sulla concatenazione in avanti.

Le prime ricerche teoriche sulla programmazione logica con vincoli furono svolte da Jaffar e Lassez (1987). Jaffar et al. (1992a) descrive lo sviluppo del sistema CLP(R) per gestire vincoli a valori reali. Jaffar et al. (1992b) generalizza la WAM producendo la CLAM (Constraint Logic Abstract Machine), usata per specificare implementazioni di CLP. Ait-Kaci e Podelski (1993) descrivono un sofisti-

query congiuntiva
complessità dei dati

risoluzione SL

risoluzione SLD

cato linguaggio di nome LIFE, che fonde CLP con la programmazione funzionale e il ragionamento sull'ereditarietà. Kohn (1991) descrive un ambizioso progetto che usa la programmazione logica con vincoli come base per un'architettura di controllo in tempo reale, applicabile alla costruzione di piloti completamente automatizzati.

Sulla programmazione logica e il Prolog esistono molti libri: *Logic for Problem Solving* (Kowalski, 1979b) è un primo volume sulla programmazione logica in generale. Testi specifici sul Prolog includono Clocksin e Mellish (1994), Shoham (1994) e Bratko (2001). Marriott e Stuckey (1998) forniscono un'eccellente trattazione di CLP. Fino alla sua chiusura, avvenuta nel 2000, la rivista più importante era il *Journal of Logic Programming*; oggi è stata rimpiazzata da *Theory and Practice of Logic Programming*. I congressi sulla programmazione logica includono l'International Conference on Logic Programming (ICLP) e l'International Logic Programming Symposium (ILPS).

La ricerca nel campo della dimostrazione di teoremi matematici era cominciata ancor prima che fossero sviluppati i primi sistemi completi del primo ordine. Il Geometry Theorem Prover di Herbert Gelernter (Gelernter, 1959) usava metodi di ricerca euristica uniti a diagrammi per la potatura di sotto-oggettivi falsi e fu in grado di dimostrare risultati alquanto complessi di geometria euclidea. Da allora, comunque, non c'è stata molta interazione tra l'intelligenza artificiale e la dimostrazione di teoremi.

All'inizio l'attenzione si concentrò sulla completezza. Dopo il fondamentale articolo di Robinson, le regole di demodulazione e paramodulazione per il ragionamento sull'uguaglianza furono introdotte rispettivamente da Wos et al. (1967) e Wos e Robinson (1968). Queste regole furono anche sviluppate indipendentemente nel contesto dei sistemi di riscrittura di termini (Knuth e Bendix, 1970). L'incorporazione del ragionamento sull'uguaglianza nell'algoritmo di unificazione è dovuta a Gordon Plotkin (1972); è anche una delle caratteristiche di QLISP (Sacerdoti et al., 1976). Jouannaud e Kirchner (1991) forniscono una panoramica sull'unificazione equazionale dal punto di vista della riscrittura di termini. Algoritmi efficienti per l'unificazione standard furono sviluppati da Martelli e Montanari (1976) e Paterson e Wegman (1978).

Oltre al ragionamento sull'uguaglianza, i dimostratori di teoremi hanno incorporato diverse procedure di decisione specializzate. Nelson e Oppen (1979) hanno proposto uno schema importante per integrare tali procedure in un sistema generale di ragionamento; tra gli altri schemi citiamo la "theory resolution" di Stickel (1985) e le "relazioni speciali" di Manna e Waldinger (1986).

Per la risoluzione sono state proposte molte strategie di controllo, a partire da quella che preferisce le clausole unitarie (Wos et al., 1964). La strategia basata su un insieme di supporto è stata proposta da Wos et al. (1965) in modo da permettere alla risoluzione di essere guidata in una certa misura dall'obiettivo. La risoluzione lineare è apparsa per la prima volta in Loveland (1970). Genesereth e Nilsson (1987, Capitolo 5) offrono una breve ma completa analisi di una grande varietà di strategie di controllo.

Guard et al. (1969) descrive il primo dimostratore di teoremi SAM, che aiutò a risolvere un problema aperto nella teoria dei reticolati. Wos e Winker (1983) forniscono una panoramica dei contributi del dimostratore di teoremi AURA alla soluzione di problemi aperti in diverse aree della matematica e della logica. McCune (1992) prosegue il discorso riportando i risultati ottenuti dal successore di AURA, OTTER. Weidenbach (2001) descrive SPASS, uno dei dimostratori di teoremi contemporanei più potenti. *A Computational Logic* (Boyer e Moore, 1979) è il riferimento principale sul dimostratore di teoremi di Boyer-Moore. Stickel (1988) tratta il Prolog Technology Theorem Prover (PTTP), che unisce i vantaggi della compilazione Prolog con la completezza della tecnica di eliminazione dei modelli (Loveland, 1968). SETHEO (Letz et al., 1992) è un altro dimostratore di teoremi ampiamente utilizzato che si basa sullo stesso approccio; su workstation moderne può eseguire diversi milioni di inferenze al secondo. LEANTAP (Beckert e Posegga, 1995) è un efficiente dimostratore di teoremi implementato in sole 25 righe di Prolog.

Le prime ricerche nel campo della sintesi automatica di programmi furono svolte da Simon (1963), Green (1969a) e Manna e Waldinger (1971). Il sistema di trasformazioni di Burstall e Darlington (1977) utilizzava il ragionamento equazionale per la sintesi di programmi ricorsivi. KIDS (Smith, 1990, 1996), che funziona come un aiutante esperto, è uno dei sistemi moderni più potenti. Manna e Waldinger (1992) forniscono un'introduzione didascalica allo stato corrente dell'arte, ponendo l'accento sul loro personale approccio deduttivo. *Automating Software Design* (Lowry e McCartney, 1991) raccoglie un certo numero di articoli sull'argomento. L'uso della logica nella progettazione hardware è esposto da Kern e Greenstreet (1999); Clarke et al. (1999) tratta il model checking per la verifica hardware.

Computability and Logic (Boolos e Jeffrey, 1989) è un buon fonte di riferimento per quanto riguarda la completezza e l'indecidibilità. Molti vecchi articoli di logica matematica sono raccolti in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). La più importante rivista per la logica matematica pura (contrapposta alla deduzione automatica) è *The Journal of Symbolic Logic*. Libri di testo orientati alla deduzione automatica includono il classico *Symbolic Logic and Mechanical Theorem Proving* (Chang e Lee, 1973) oltre a lavori più recenti di Wos et al. (1992), Bibel (1993) e Kaufmann et al. (2000). L'antologia *Automation of Reasoning* (Siekmann e Wrightson, 1983) include molti dei primi articoli dedicati sulla deduzione automatica. Altre panoramiche storiche sono state scritte da Loveland (1984) e Bundy (1999). La rivista più importante nel campo della dimostrazione di teoremi è il *Journal of Automated Reasoning*; il congresso principale è l'annuale Conference on Automated Deduction (CADE). La ricerca sulla dimostrazione di teoremi è strettamente imparentata all'uso della logica per l'analisi di programmi e linguaggi di programmazione: per questa disciplina il congresso più importante è Logic in Computer Science.

Esercizi

- 9.1** Dimostrate, partendo da principî primitivi, che l'istanziazione universale è corretta e che l'istanziazione esistenziale produce una base di conoscenza inferenzialmente equivalente.
- 9.2** Da *Gradisce(Jerry, Gelato)* sembra ragionevole inferire $\exists x \text{ Gradisce}(x, \text{Gelato})$. Scrivete una regola generale, l'**introduzione dell'esistenziale**, che formalizza questo tipo di inferenza. Fate attenzione nel definire precisamente le condizioni che devono essere soddisfatte dalle variabili e dai termini coinvolti.
- 9.3** Supponiamo che una base di conoscenza contenga una sola formula, $\exists x \text{ AltoCome}(x, \text{Everest})$. Quale delle seguenti formule è un risultato legittimo dell'applicazione dell'istanziazione esistenziale?
- AltoCome(Everest, Everest)*.
 - AltoCome(Kilimanjaro, Everest)*.
 - AltoCome(Kilimanjaro, Everest) \wedge AltoCome(MonteRosa, Everest)* (dopo due applicazioni della regola).
- 9.4** Per ogni coppia di formule atomiche scrivete l'unificatore più generale, se esiste:
- $P(A, B, B), P(x, y, z)$.
 - $Q(y, G(A, B)), Q(G(x, x), y)$.
 - Più Vecchio(Padre(y), y), Più Vecchio(Padre(x), Giovanni)* .
 - Conosce(Padre(y), y), Conosce(x, x)* .
- 9.5** Considerate i reticolî di sussunzione della Figura 9.2.
- Costruite il reticolo per la formula *Dipendente(Madre(Giovanni), Padre(Riccardo))*.
 - Costruite il reticolo per la formula *Dipendente(IBM, y)* (ovvero, “tutti lavorano per IBM”). Ricordatevi di includere ogni tipo di interrogazione che unifica con la formula.
 - Assumiamo come ipotesi che STORE indicizzi ogni formula con ogni nodo del suo reticolo di sussunzione. Spiegate come dovrebbe funzionare FETCH quando qualche formula contiene delle variabili; usate come esempi le formule in (a) e (b) qui sopra, nonché la query *Dipendente(x, Padre(x))*.
- 9.6** Supponiamo di inserire in un database logico un frammento dell'ultimo censimento che includa l'età, la città di residenza, la data di nascita e la madre di ogni persona, usando come chiave primaria di ogni elemento il codice fiscale. In questo modo, l'età di Stefano sarebbe rappresentata come *Età(GBRSFN70S23L682P, 34)*. Presumendo che si adotti la normale conca-

tenazione all'indietro, quale dei seguenti schemi di indicizzazione S1-S5 permette di trovare una facile soluzione per quali delle interrogazioni Q1-Q4?

- ♦ S1: un indice per ogni atomo in ogni posizione.
 - ♦ S2: un indice per ogni primo argomento.
 - ♦ S3: un indice per ogni atomo predicato.
 - ♦ S4: un indice per ogni *combinazione* di predicato e primo argomento.
 - ♦ S5: un indice per ogni *combinazione* di predicato e secondo argomento più un indice per ogni primo argomento (non standard).
- ♦ Q1: *Età(GBRSFN70S23L682P, x)*
 - ♦ Q2: *ResidenteIn(x, Lecce)*
 - ♦ Q3: *Madre(x, y)*
 - ♦ Q4: *Età(x, 34) ∧ ResidenteIn(x, Galatina)*

- 9.7 Si potrebbe supporre che, durante la concatenazione all'indietro, sia possibile evitare il problema del conflitto tra variabili nell'unificazione mediante la standardizzazione separata di tutte le formule nella base di conoscenza una volta per tutte. Dimostrate che, per certe formule, quest'approccio non può funzionare (*suggerimento*: considerate una formula in cui una parte della formula stessa unifica con un'altra).
- 9.8 Spiegate come riscrivere un qualsiasi problema 3-SAT di dimensione arbitraria usando una singola clausola definita del primo ordine e non più di 30 fatti ground.
- 9.9 Scrivete le rappresentazioni logiche delle seguenti formule in modo tale che siano utilizzabili con il Modus Ponens generalizzato:
- a. Cavalli, mucche e maiali sono mammiferi.
 - b. Il figlio di un cavallo è un cavallo.
 - c. Barbablù è un cavallo.
 - d. Barbablù è padre di Carletto.
 - e. Padre e figlio sono relazioni inverse.
 - f. Ogni mammifero ha un padre.
- 9.10 In questo esercizio utilizzeremo le formule che avete scritto nel precedente Esercizio 9.9 per rispondere a una domanda mediante l'algoritmo di concatenazione all'indietro.
- a. Disegnate l'albero di dimostrazione generato da un algoritmo esaustivo di concatenazione all'indietro per l'interrogazione $\exists h \text{ Cavallo}(h)$, in cui il matching delle clausole segue l'ordine dato.
 - b. Che cosa potete notare riguardo a questo dominio?
 - c. Quante soluzioni per h seguono effettivamente dalle vostre formule?
 - d. Potete escogitare un modo di trovarle tutte? (*Suggerimento*: potreste consultare Smith et al., 1986).

- 9.11** Un popolare indovinello per bambini recita: "non ho fratelli né sorelle, lo dico senza piglio/ma il padre di quell'uomo di mio padre è il figlio". Usate le regole del dominio dei rapporti di parentela, che abbiamo fornito nel Capitolo 8, per dimostrare chi è l'uomo misterioso. Potete applicare tutti i metodi di inferenza descritti in questo capitolo. Perché ritenete che quest'indovinello sia difficile?
- 9.12** Tracciate su carta l'esecuzione dell'algoritmo di concatenazione all'indietro della Figura 9.6 applicato alla risoluzione del problema del colonnello criminale. Mostrate la sequenza di valori assunti dalla variabile *obiettivi* visualizzandola per mezzo di un albero.
- 9.13** Il seguente codice Prolog definisce un predicato P :
- ```
P(X,[X | Y]) .
P(X,[Y | Z]) :- P(X,Z) .
```
- Disegnate gli alberi di dimostrazione e scrivete le soluzioni delle interrogazioni  $P(A,[ 1,2,3] )$  e  $P(2,[ 1,A,3] )$ .
  - Quale operazione standard sulle liste rappresenta P?
- 9.14** In quest'esercizio prenderemo in esame l'operazione di ordinamento in Prolog.
- Scrivete le clausole Prolog che definiscono il predicato *ordinata(L)*, che è vero solo se la lista L è ordinata.
  - Scrivete una definizione Prolog per il predicato *perm(L, M)*, che è vero solo se L è una permutazione di M .
  - Definite *ordina(L, M)* (M è una versione ordinata di L) usando *perm* e *ordinata*.
  - Eseguite l'ordinamento su liste sempre più lunghe, finché non vi scocciate. Qual è la complessità temporale del vostro programma?
  - Scrivete in Prolog un algoritmo di ordinamento più veloce, come l'insertion sort o il quicksort.
- 9.15** In questo esercizio prenderemo in esame l'applicazione ricorsiva delle regole di riscrittura usando la programmazione logica. Una regola di riscrittura (o **demodulatore**, nel gergo di OTTER) è un'equazione con una direzione specificata. Ad esempio, la regola di riscrittura  $x + 0 \rightarrow x$  suggerisce di rimpiizzare ogni espressione che corrisponde a  $x + 0$  con l'espressione  $x$ . L'applicazione delle regole di riscrittura è un aspetto fondamentale dei sistemi che ragionano su equazioni: per rappresentarle useremo il predicato *riscrivi(X, Y)*. Ad esempio, la regola che abbiamo citato poc'anzi si scriverebbe *riscrivi(X+0, X)*. Alcuni termini sono *primitivi* e non possono essere ulteriormente semplificati; scriveremo *primitivo(0)* per dire che 0 è un termine primitivo.

- a. Scrivete una definizione del predicato `semplifica(X, Y)`, che è vero quando `Y` è una versione semplificata di `X`, ovvero quando non è possibile applicare ulteriori regole di riscrittura ad alcuna sottoespressione di `Y`.
- b. Scrivete una collezione di regole per la semplificazione di espressioni che includono operatori aritmetici e applicate il vostro algoritmo di semplificazione ad alcune espressioni di prova.
- c. Scrivete una collezione di regole di riscrittura per la differenziazione simbolica e usatela insieme alle vostre regole di semplificazione per differenziare e semplificare espressioni aritmetiche, incluso l'esponente.
- 9.16 In quest'esercizio prenderemo in considerazione l'implementazione di algoritmi di ricerca in Prolog. Supponiamo che `successore(X, Y)` sia vero quando lo stato `Y` è un successore dello stato `X` e che `goal(X)` sia vero quando `X` è uno stato obiettivo. Scrivete la definizione di `risolvi(X, P)`, che significa che `P` è un cammino (lista di stati) che comincia con `X`, termina in uno stato obiettivo e consiste in una sequenza di passi legali definiti da `successore`. Troverete che il modo più facile sarà utilizzare una ricerca in profondità. Sarebbe difficile aggiungere un controllo euristico della ricerca?
- 9.17 Come si potrebbe usare la risoluzione per dimostrare che una formula è valida? E insoddisfacibile?
- 9.18 Dalla formula “tutti i cavalli sono animali” segue che “la testa di un cavallo è la testa di un animale”. Dimostrate che quest’inferenza è valida eseguendo i seguenti passi.
- Traducete la premessa e la conseguenza in logica del primo ordine. Usate tre predicati: `TestaDi(b, x)` (che significa “`b` è la testa di `x`”), `Cavallo(x)` e `Animale(x)`.
  - Negate la conclusione e convertite premessa e conclusione negata in forma normale congiuntiva.
  - Usate la risoluzione per mostrare che la conclusione segue dalla premessa.
- 9.19 Ecco due formule in logica del primo ordine:
- (A):  $\forall x \exists y (x \geq y)$
  - (B):  $\exists y \forall x (x \geq y)$
- Assumete che le variabili possano spaziare su tutto l’intervallo dei numeri naturali  $0, 1, 2, \dots, \infty$  e che il predicato “ $\geq$ ” significhi “è maggiore o uguale di”. Sotto questa interpretazione, traducete (A) e (B) in linguaggio naturale.
  - (A) è vera sotto questa interpretazione?
  - (B) è vera sotto questa interpretazione?
  - (A) implica logicamente (B)?

- e. (B) implica logicamente (A)?
  - f. Usando la risoluzione, cercate di dimostrare che (A) è una conseguenza di (B). Fate lo anche se non pensate che (B) implichi logicamente (A); continuate finché la dimostrazione diventa impossibile e non potete procedere (se questo accade). Mostrate la sostituzione unificante per ogni passo di risoluzione. Se la dimostrazione fallisce spiegate esattamente dove, come e perché.
  - g. Ora cercate di dimostrare che (B) è una conseguenza logica di (A).
- 9.20 La risoluzione può produrre dimostrazioni non costruttive se le interrogazioni contengono variabili, la qual cosa ci ha costretti a introdurre meccanismi speciali per estrarre risposte definite. Spiegate perché questo problema non sorge con basi di conoscenza che contengono solo clausole definite.
- 9.21 In questo capitolo abbiamo affermato che la risoluzione non può essere usata per generare tutte le conseguenze logiche di un insieme di formule. Esiste un qualsiasi algoritmo che può farlo?

## Capitolo 10

# Rappresentazione della conoscenza

*In cui mostriamo come usare la logica del primo ordine per rappresentare gli aspetti più importanti del mondo reale, come azioni, spazio, tempo, eventi mentali e shopping.*

Gli ultimi tre capitoli hanno descritto la tecnologia alla base degli agenti basati sulla conoscenza: la sintassi, la semantica e la teoria delle dimostrazioni per la logica proposizionale e quella del primo ordine, e l'implementazione degli agenti che utilizzano tali logiche. In questo capitolo affronteremo il problema di quale contenuto inserire nella base di conoscenza degli agenti: ovvero, come rappresentare fatti che riguardano il mondo.

Il Paragrafo 10.1 introduce l'idea di un'ontologia generale, che organizza tutti gli elementi del mondo in una gerarchia di categorie. Il Paragrafo 10.2 tratta le categorie base degli oggetti, delle sostanze e delle misure. Il Paragrafo 10.3 discute la rappresentazione delle azioni – un aspetto centrale per la costruzione di agenti basati sulla conoscenza – e presenta il concetto più generale di eventi, o frammenti di spazio-tempo. Il Paragrafo 10.4 esamina la conoscenza riguardante le credenze e il Paragrafo 10.5 unisce tutte le categorie di conoscenza presentando come esempio l'ambiente di un negozio su Internet. I Paragrafi 10.6 e 10.7 trattano sistemi specializzati di ragionamento per la rappresentazione di conoscenza incerta e mutevole.

## 10.1 Ingegneria ontologica

Nei domini “giocattolo”, la scelta della rappresentazione non è molto importante; definire un vocabolario consistente risulta facile. D'altra parte, domini complessi come il commercio su Internet o il controllo di un robot in un ambiente fisico mutevole richiedono rappresentazioni più generali e flessibili. Questo capitolo mostra come creare tali rappresentazioni concentrandosi su concetti generali che si presen-

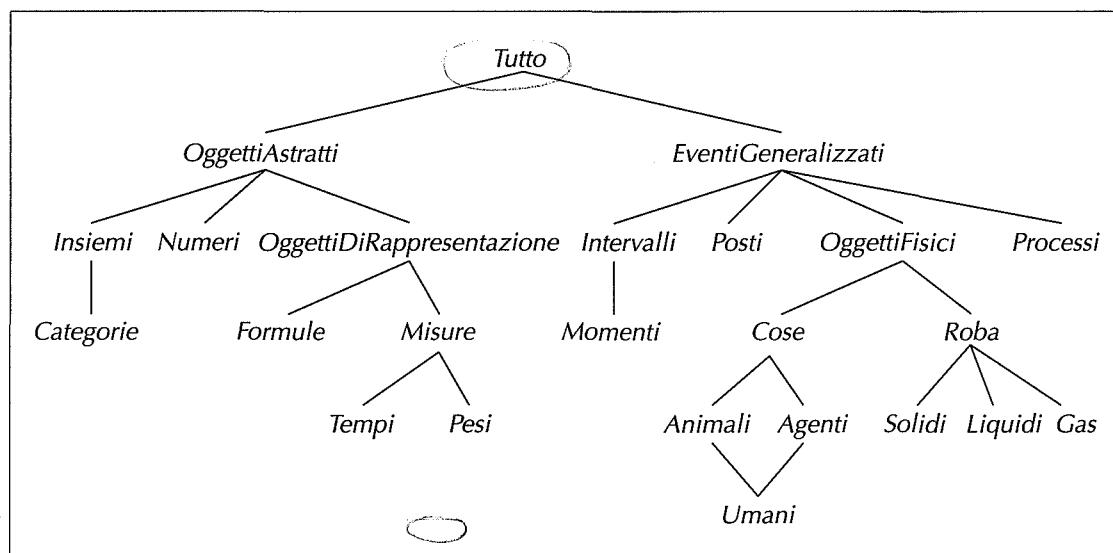
ingegneria ontologica

ontologia superiore

tano in molti domini diversi, come *Azioni*, *Tempo*, *Oggetti Fisici* e *Credenze*. Rappresentare questi concetti astratti prende talvolta il nome di *ingegneria ontologica*: è un'attività imparentata con il processo di *ingegneria della conoscenza* che abbiamo descritto nel Paragrafo 8.4, ma su scala più grande.

La prospettiva di rappresentare *l'intero mondo può intimidire*. Naturalmente il nostro scopo non è scrivere una descrizione completa di tutto l'esistente (sarebbe troppo anche per un libro di 1000 pagine): lasceremo invece dei "segnaposto" per indicare i punti in cui si potrà aggiungere conoscenza appartenente a qualsiasi dominio. Definiremo per esempio che cosa significa essere un oggetto fisico, mentre i dettagli dei diversi tipi di oggetto (robot, televisioni, libri, o qualsiasi altra cosa) potranno essere aggiunti in seguito. L'infrastruttura generale dei concetti prende il nome di *ontologia superiore*, per la convenzione di disegnare i grafi con i concetti più generali in alto e quelli più specifici sotto di essi, come si vede nella Figura 10.1.

Prima di approfondire ulteriormente la nozione di ontologia occorre chiarire un punto importante. Abbiamo stabilito di usare la logica del primo ordine per discutere il contenuto e l'organizzazione della conoscenza, ma alcuni aspetti del mondo reale non sono facili da catturare con la logica dei predicati. La difficoltà principale sta nel fatto che quasi tutte le generalizzazioni prevedono delle eccezioni, o sono vere fino a un certo punto. Ad esempio, la regola "i pomodori sono rossi" in genere è utile, ma esistono pomodori gialli, verdi o arancioni. Eccezioni simili possono essere trovate per quasi tutte le asserzioni generali in questo capitulo.



**Figura 10.1** L'ontologia superiore del mondo, con gli argomenti che tratteremo nel capitolo. Ogni arco indica che il concetto inferiore è una specializzazione di quello superiore.

lo. La capacità di gestire eccezioni e incertezza è estremamente importante, ma del tutto ortogonale alla comprensione di un'ontologia generale. Per questa ragione rimanderemo la discussione delle eccezioni al Paragrafo 10.6 e l'argomento più generale dell'informazione incerta al Capitolo 13, il primo del 2° volume.

Qual è l'utilità di un'ontologia superiore? Considerate ancora l'ontologia per i circuiti del Paragrafo 8.4: si vede subito che ci sono molte ipotesi semplificative. Il tempo, ad esempio, è totalmente mancante. I segnali sono fissi, e non si propagano. La struttura del circuito rimane costante. Se volessimo rendere l'ontologia più generale dovremmo considerare i segnali in momenti precisi, includere la lunghezza delle piste e i ritardi di propagazione. Questo ci permetterebbe di simulare le proprietà temporali del circuito, cosa che i progettisti hardware fanno spesso. Potremmo anche introdurre altre classi interessanti di porte logiche, ad esempio descrivendone la tecnologia (TTL, MOS, CMOS e così via) oltre alla specifica di input/output. Se volessimo discutere l'affidabilità o la diagnostica degli errori, dovremmo includere la possibilità che la struttura del circuito o le proprietà delle porte logiche possano cambiare spontaneamente. Per trattare correttamente le capacità parassite, dovremmo passare da una rappresentazione puramente topologica delle connessioni a una descrizione più realistica delle proprietà geometriche.

Se guardiamo il mondo del wumpus possiamo fare considerazioni simili. Sebbene il tempo sia incluso nella rappresentazione, la sua struttura è molto semplice: non accade nulla se non quando l'agente esegue un'azione, e tutti i cambiamenti hanno durata istantanea. Un'ontologia più generale, e più adatta al mondo reale, dovrebbe prevedere la possibilità che si verifichino cambiamenti simultanei estesi nel tempo. Abbiamo anche usato un predicato *Pozzo* per indicare quali stanze contenevano pozzi: avremmo potuto rappresentarne tipi diversi definendo più individui appartenenti alla classe dei pozzi, ognuno con differenti caratteristiche. In modo analogo, potremmo desiderare di includere altri animali oltre al wumpus. Potrebbe anche non essere possibile dedurre la specie esatta dalle percezioni disponibili, nel qual caso dovremmo costruire una tassonomia biologica del mondo del wumpus per aiutare l'agente a orientarsi con pochi indizi.

A ogni ontologia specializzata è possibile apportare modifiche come queste per aumentarne la generalità. Sorge quindi spontanea la domanda: tutte queste ontologie convergono in un'unica ontologia generale? Dopo secoli di pensiero filosofico e computazionale, la risposta è "può darsi". Noi ne presenteremo una versione che rappresenta la sintesi delle idee che si sono susseguite nei secoli. Le ontologie generali hanno due caratteristiche che le distinguono da semplici collezioni di ontologie dedicate:

- ♦ un'ontologia generale, con l'aggiunta di assiomi specifici, dovrebbe essere applicabile a quasi tutti i domini particolari. Questo significa che, per quanto possibile, nessun problema di rappresentazione dev'essere risolto con soluzioni *ad hoc* o ignorato come se non esistesse;

- ♦ in qualsiasi dominio sufficientemente complicato occorre *unificare* aree diverse di conoscenza, perché il ragionamento e la risoluzione di problemi possono coinvolgere più aree contemporaneamente. Un sistema robotico di riparazione di circuiti, per esempio, dev'essere in grado di ragionare sui circuiti in termini di collegamenti elettrici e architettura fisica, ma anche sul tempo, sia per analizzare i circuiti stessi che per stimare il costo delle riparazioni. Deve quindi essere possibile combinare le formule che descrivono il tempo con quelle che descrivono configurazioni spaziali, e le formule devono applicarsi ugualmente bene a nanosecondi e minuti, angstrom e metri.

Dopo averla presentata, useremo la nostra ontologia generale per descrivere il dominio del commercio su Internet. Questo dominio si presta bene a mettere alla prova l'ontologia e lascia molto spazio al lettore per aggiungere creativamente la rappresentazione di conoscenza aggiuntiva. Considerate che un agente per acquisti su Internet deve conoscere una miriade di titoli e autori per comprare libri su Amazon.com, ogni sorta di cibo per fare la spesa su Peapod.com, e tutto l'immaginabile per fare affari su Ebay.com.<sup>1</sup>

## 10.2 Categorie e oggetti

categorie



L'organizzazione degli oggetti in categorie è una parte fondamentale della rappresentazione della conoscenza. Benché l'interazione con il mondo abbia luogo a livello di singoli oggetti, molta parte del ragionamento si svolge al livello delle categorie. Ad esempio, un cliente potrebbe avere come obiettivo l'acquisto di un pallone da basket, piuttosto che di un particolare pallone come il BB<sub>9</sub>. Le categorie sono anche utili per formulare predizioni sugli oggetti una volta che sono stati classificati. Si può inferire la presenza di un oggetto dagli input percettivi, dedurre dalle caratteristiche percepite la sua appartenenza a una categoria e quindi sfruttare l'informazione nota sulla categoria per formulare predizioni riguardo l'oggetto. Ad esempio se è verde, ha la buccia maculata, è grande e vagamente ovale, si può inferire che l'oggetto è un'anguria e da questo dedurre che non starebbe male in una macedonia.

Per rappresentare le categorie nella logica del primo ordine ci sono due possibilità: si possono usare predicati oppure oggetti. Questo significa che possiamo usare il predicato *PalloneDaBasket(b)*, oppure possiamo reificare la categoria e farla diventare un oggetto, *PalloniDaBasket*. A quel punto potremmo scrivere *Membro(b, PalloniDaBasket)*, abbreviato in  $b \in \text{PalloniDaBasket}$ , per affermare che

<sup>1</sup> Ci scusiamo se, per circostanze al di là del nostro controllo, alcuni di questi negozi online non sono più attivi nel momento in cui state leggendo.

b è un membro della categoria dei palloni da basket. Possiamo anche scrivere Sottoinsieme(PalloniDaBasket, Palloni), abbreviato in PalloniDaBasket ⊂ Palloni, per dire che PalloniDaBasket è una sottocategoria, o sottoinsieme, di Palloni. Una categoria può quindi essere considerata come l'insieme dei suoi membri o come un oggetto più complesso per cui sono definite le relazioni Membro e Sottoinsieme.

Le categorie servono a organizzare e semplificare la base di conoscenza attraverso il meccanismo dell'ereditarietà. Se diciamo che tutte le istanze della categoria Cibo sono commestibili, e affermiamo che Frutta è una sottoclasse di Cibo e Mele è a sua volta una sottoclasse di Frutta, sappiamo che ogni mela è commestibile. Diciamo che le singole mele ereditano la proprietà della commestibilità, in questo caso grazie alla loro appartenenza alla categoria Cibo.

ereditarietà

La relazione di sottoclasse organizza le categorie in una tassonomia, o gerarchia tassonomica. Le tassonomie sono state usate esplicitamente per secoli in varie discipline scientifiche: ad esempio, la biologia sistematica ha come scopo di fornire una tassonomia di tutte le specie viventi ed estinte; la biblioteconomia ne ha sviluppata una di tutte le branche del sapere, rappresentata con il sistema Dewey Decimale; il fisco e gli altri ministeri governativi hanno dato origine a complesse tassonomie delle varie occupazioni e dei prodotti commerciali. Le tassonomie sono anche un aspetto importante del buon senso comune.

tassonomia

La logica del primo ordine rende facile esprimere fatti che riguardano intere categorie, mettendo gli oggetti in relazione con esse o quantificando tutti i membri.

◆ Un oggetto è membro di una categoria. Ad esempio:

$$BB_9 \in PalloniDaBasket$$

◆ Una categoria è una sottoclasse di un'altra categoria. Ad esempio:

$$PalloniDaBasket \subset Palloni$$

◆ Tutti i membri di una categoria hanno determinate caratteristiche.

Ad esempio:

$$x \in PalloniDaBasket \Rightarrow Rotondo(x)$$

◆ I membri di una categoria possono essere riconosciuti dalle loro caratteristiche. Ad esempio:

$$Arancione(x) \wedge Rotondo(x) \wedge Diametro(x)=24\text{cm}.$$

$$\wedge x \in Palloni \Rightarrow x \in PalloniDaBasket$$

◆ Una categoria può avere caratteristiche proprie. Ad esempio:

$$Cani \in SpecieAddomesticate$$

Notate che, dato che Cani è una categoria ed è membro di SpecieAddomesticate, quest'ultima dev'essere una categoria di categorie. Esistono anche categorie di categorie di categorie, ma non sono molto usate.

Benché le relazioni di sottoclasse e di appartenenza siano le più importanti, è anche possibile esprimere relazioni tra categorie che non sono sottoclassi l'una dell'altra. Ad esempio, se ci limitiamo a dire che Maschi e Femmine sono sottoclassi di

disgiunte

scomposizione  
esaustiva  
partizione

*Animali*, non abbiamo espresso il fatto che un maschio non può essere una femmina. Se due categorie non hanno membri in comune si dice che sono disgiunte. Una volta che abbiamo affermato che maschi e femmine sono categorie disgiunte, non è ancora detto che un animale che non è maschio debba essere una femmina: per far questo dobbiamo dire che le due categorie costituiscono una scomposizione esaustiva degli animali. Una scomposizione esaustiva disgiunta prende il nome di partizione. I seguenti esempi illustrano i tre concetti:

*Disgiunte*( $\{Animali, Vegetali\}$ )

*ScomposizioneEsaustiva*( $\{Americani, Canadesi, Messicani\}$ , *NordAmericanis*)

*Partizione*( $\{Maschi, Femmine\}$ , *Animali*) .

Notate che la scomposizione esaustiva dei NordAmericanis non è una partizione, perché alcune persone hanno una doppia cittadinanza. I tre prediciati sono definiti come segue:

*Disgiunte*( $s$ )  $\Leftrightarrow (\forall c_1, c_2 \quad c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow Intersezione(c_1, c_2) = \{\})$

*ScomposizioneEsaustiva*( $s, c$ )  $\Leftrightarrow (\forall i \quad i \in c \Leftrightarrow \exists c_2 \quad c_2 \in s \wedge i \in c_2)$

*Partizione*( $s, c$ )  $\Leftrightarrow Disgiunte(s) \wedge ScomposizioneEsaustiva(s, c)$

Le categorie possono anche essere definite fornendo condizioni necessarie e sufficienti per l'appartenenza. Ad esempio, uno scapolo è un maschio adulto non sposato:

$x \in Scapoli \Leftrightarrow NonSposato(x) \wedge x \in Adulti \wedge x \in Maschi$  .

Come vedremo nel box dedicato ai tipi naturali, una definizione logica precisa delle categorie non è sempre possibile, né sempre necessaria.

## Composizione fisica

L'idea che un oggetto possa essere parte di un altro oggetto è abbastanza intuitiva: il nostro naso fa parte della nostra faccia, la Romania fa parte dell'Europa, e questo capitolo fa parte di un libro. Useremo la relazione generale ParteDi per affermare che una cosa fa parte di un'altra. Gli oggetti possono essere raggruppati in gerarchie ParteDi, che ricordano le gerarchie di sottoinsiemi:

*ParteDi*(*Bucarest, Romania*)

*ParteDi*(*Romania, EuropaOrientale*)

*ParteDi*(*EuropaOrientale, Europa*)

*ParteDi*(*Europa, Terra*) .

La relazione ParteDi è transitiva e riflessiva:

$ParteDi(x, y) \wedge ParteDi(y, z) \Rightarrow ParteDi(x, z)$  .

$ParteDi(x, x)$  .

Possiamo quindi concludere che vale ParteDi(*Bucarest, Terra*) .

Le categorie di oggetti composti sono spesso caratterizzate da relazioni strutturali tra le parti. Ad esempio, un bipede ha due gambe attaccate a un corpo:

$$\text{Bipede}(a) \Rightarrow \exists l_1, l_2, b. \text{Gamba}(l_1) \wedge \text{Gamba}(l_2) \wedge \text{Corpo}(b) \wedge \\ \text{ParteDi}(l_1, a) \wedge \text{ParteDi}(l_2, a) \wedge \text{ParteDi}(b, a) \wedge \\ \text{Attaccata}(l_1, b) \wedge \text{Attaccata}(l_2, b) \wedge \\ l_1 \neq l_2 \wedge [\forall l_3 (\text{Gamba}(l_3) \wedge \text{ParteDi}(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2))].$$

La notazione per esprimere “esattamente due” è un po’ arzigogolata; siamo obbligati a dire che esistono due gambe, che non sono lo stesso oggetto e che se qualcuno fa riferimento a una terza gamba, essa deve corrispondere a una delle prime due. Nel Paragrafo 10.6 presenteremo il formalismo della logica descrittiva, che rende molto più facile esprimere vincoli come questo.

Possiamo definire una relazione *PartizioneDiParti* con un significato analogo a quello di *Partizione* per le categorie (v. Esercizio 10.6). Un oggetto è composto dalle parti che costituiscono la sua *PartizioneDiParti* e può anche derivare da esse alcune caratteristiche: ad esempio, la massa di un oggetto composto è la somma delle masse delle sue parti. Notate che non è affatto così con le categorie, che non hanno alcuna massa, anche se i loro elementi potrebbero averla.

Può anche essere utile definire oggetti composti con parti definite ma nessuna particolare struttura. Potremmo ad esempio voler dire che “le mele in questo sacchetto pesano due chili”. Potremmo essere tentati di assegnare questo peso all’insieme delle mele nel sacchetto, ma questo sarebbe un errore, perché un insieme è un concetto matematico astratto che può contenere elementi ma non può avere alcun peso! Occorre un nuovo concetto, che chiameremo **mucchio** (*bunch*). Per esempio, se le mele sono *Mela*<sub>1</sub>, *Mela*<sub>2</sub> e *Mela*<sub>3</sub>, allora

$$\text{MucchioDi}(\{\text{Mela}_1, \text{Mela}_2, \text{Mela}_3\})$$

denota l’oggetto composto che ha le tre mele come parti (e non come elementi). A questo punto sarà possibile usare il mucchio come un oggetto normale, ancorché privo di struttura. Notate che  $\text{MucchioDi}\{x\} = x$ . Inoltre,  $\text{MucchioDi}(\text{Mele})$  è l’oggetto composto da tutte le mele esistenti, e non dev’essere confuso con *Mele*, la categoria o insieme delle mele.

Possiamo definire *MucchioDi* partendo dalla relazione *ParteDi*. È ovvio che ogni elemento di *s* fa parte di *MucchioDi(s)*:

$$\forall x \ x \in s \Rightarrow \text{ParteDi}(x, \text{MucchioDi}(s)).$$

Inoltre, *MucchioDi(s)* è il più piccolo oggetto che soddisfa questa condizione. In altre parole, *MucchioDi(s)* dev’essere parte di ogni oggetto che ha come parti tutti gli elementi di *s*:

$$\forall y \ [\forall x \ x \in s \Rightarrow \text{ParteDi}(x, y)] \Rightarrow \text{ParteDi}(\text{MucchioDi}(s), y).$$

Questi assiomi sono un esempio della tecnica generale chiamata **minimizzazione logica**, che permette di definire un oggetto come il più piccolo che soddisfa certe condizioni.

oggetti composti

mucchio

minimizzazione logica

misure

funzione di unità

## Misure

Sia per la scienza che per il buon senso, gli oggetti hanno un'altezza, una massa, un costo e così via. I valori che assegniamo a queste caratteristiche prendono il nome di misure. Le misure quantitative ordinarie sono abbastanza facili da rappresentare. Immaginiamo che l'universo includa "oggetti misura" astratti, come la lunghezza di questo segmento: [ ]. Possiamo chiamare questa lunghezza 1,5 pollici o 3,81 centimetri. Così, nel nostro linguaggio la stessa lunghezza può avere nomi differenti. Nella logica questo può essere fatto combinando una funzione di unità con un numero (l'Esercizio 10.8 prende in esame uno schema alternativo). Se il nome del segmento è  $L_1$ , possiamo scrivere

$$\text{Lunghezza}(L_1) = \text{Pollici}(1,5) = \text{Centimetri}(3,81).$$

La conversione tra unità di misura si ottiene esprimendo equazioni di uguaglianza tra multipli di unità diverse:

$$\text{Centimetri}(2,54 \times d) = \text{Pollici}(d).$$

Assiomi simili possono essere scritti per libbre e chilogrammi, secondi e giorni, dollari e centesimi. Le misure possono essere usate per descrivere oggetti nel modo seguente:

$$\text{Diametro}(\text{PalloneDaBasket}_{12}) = \text{Pollici}(9,5)$$

$$\text{Prezzo}(\text{PalloneDaBasket}_{12}) = \$ (19)$$

$$d \in \text{Giorni} \Rightarrow \text{Durata}(d) = \text{Ore}(24)$$

Notate che \$(1) non è un biglietto da un dollaro! Si possono possedere due o più banconote uguali, ma c'è un solo oggetto di nome \$(1). Notate anche che, sebbene Pollici(0) e Centimetri(0) si riferiscano alla stessa lunghezza nulla, non sono affatto identici ad altre misure zero, come Secondi(0).

Semplici misure quantitative sono facili da rappresentare; la cosa si fa più difficile quando non esiste una scala di valori precisa. Gli esercizi hanno un grado di difficoltà, i dessert un grado di dolcezza e le poesie di bellezza, tuttavia è impossibile assegnare a queste caratteristiche un valore numerico. Un fanatico della contabilità potrebbe pensare che proprietà come queste non siano utili per il ragionamento logico o, ancora peggio, cercare di imporre una scala numerica per la bellezza. Questo sarebbe un grave errore, perché non è necessario: l'aspetto più importante delle misure infatti non è il loro particolare valore numerico, ma il fatto che possono essere ordinate.

Benché le misure non siano numeri, possiamo ancora confrontarle usando un simbolo di ordinamento come  $>$ . Ad esempio, si potrebbe affermare che gli esercizi di Norvig sono più difficili di quelli di Russell, e che uno studente ottiene meno punti quando deve svolgere esercizi difficili:

$$e_1 \in \text{Esercizi} \wedge e_2 \in \text{Esercizi} \wedge \text{ScrittoDa}(\text{Norvig}, e_1) \wedge \text{ScrittoDa}(\text{Russell}, e_2) \Rightarrow \\ \text{Difficoltà}(e_1) > \text{Difficoltà}(e_2).$$

$$e_1 \in \text{Esercizi} \wedge e_2 \in \text{Esercizi} \wedge \text{Difficoltà}(e_1) > \text{Difficoltà}(e_2) \Rightarrow \\ \text{PunteggioAtteso}(e_1) < \text{PunteggioAtteso}(e_2).$$

### TIPI NATURALI

Alcune categorie hanno una definizione precisa; un oggetto è un triangolo se e solo se è un poligono con tre lati. D'altra parte, la maggior parte delle categorie del mondo reale non si possono definire esattamente: le chiameremo **tipi naturali**. I pomodori, ad esempio, di solito hanno un colore rosso spento; sono più o meno sferici con una depressione in alto in corrispondenza del picciolo; sono grandi dai cinque ai dieci centimetri; hanno una buccia sottile ma resistente e contengono polpa, succo e semi. Questo non significa che non esistano delle variazioni: alcuni pomodori sono arancioni, quelli acerbi sono verdi, ce ne sono di più grandi o più piccoli della media, e poi ci sono i "ciliegini" che sono tutti molto più piccoli. Invece di fornire una definizione precisa dei pomodori, possiamo elencare una serie di caratteristiche che possono aiutarci a identificare oggetti che sono chiaramente pomodori tipici, ma che potrebbero risultare inadeguate nel caso di altri oggetti (potrebbe esistere un pomodoro coperto di peluria, come una pesca?).

Tutto questo per un agente logico rappresenta un problema. L'agente non può essere sicuro che l'oggetto che ha percepito è un pomodoro, e anche se lo fosse, non potrebbe sapere con certezza quali proprietà dei pomodori tipici abbia questa particolare istanza. Questo problema rappresenta l'inevitabile conseguenza di un ambiente parzialmente osservabile.

Un approccio utile consiste nel separare ciò che è vero per tutte le istanze di una categoria da ciò che vale solo per le istanze tipiche. Così, oltre alla categoria *Pomodori*, avremo anche la categoria *Tipici(Pomodori)*. La funzione *Tipici* fa corrispondere una categoria alla sua sottoclasse che contiene solo istanze tipiche:

$$\text{Tipici}(c) \subseteq c$$

La maggior parte della conoscenza relativa ai tipi naturali si applicherà alle loro istanze tipiche:

$$x \in \text{Tipici}(\text{Pomodori}) \Rightarrow \text{Rosso}(x) \wedge \text{Sferico}(x)$$

In questo modo è possibile scrivere informazioni utili sulle categorie senza vincolarsi a definizioni esatte.

La difficoltà di fornire definizioni precise della maggior parte delle categorie naturali è stata analizzata approfonditamente da Wittgenstein (1953) nel suo libro *Investigazioni Filosofiche*, in cui sfruttò l'esempio dei giochi per mostrare che i membri di una categoria condividono "somiglianze di famiglia" e non caratteristiche necessarie e sufficienti.

La stessa utilità del concetto di definizione precisa fu contestata da Quine (1953), che fece notare che persino la definizione di "scapolo" come maschio adulto non sposato è sospetta. Consideriamo ad esempio una frase come "il Papa è scapolo": sebbene non sia falsa in senso stretto, in questo caso l'uso del termine è certamente infelice, perché porta a inferenze non volute da parte dell'ascoltatore. Questa tensione potrebbe forse essere risolta distinguendo tra le definizioni logiche, adatte per la rappresentazione della conoscenza interna, e i criteri più raffinati richiesti dall'uso appropriato del linguaggio. Quest'ultimo potrebbe essere ottenuto "filtrando" le affermazioni ricavate dalle definizioni logiche. Potrebbe anche darsi che i fallimenti nell'uso del linguaggio possano servire da feedback per la modifica delle definizioni interne, in modo tale che il filtraggio diventi superfluo.

Questo è sufficiente per permettere a uno studente di decidere quali esercizi svolgere, anche se da nessuna parte è stata usata una scala numerica per la difficoltà (naturalmente, sarà necessario scoprire chi è l'autore di ogni esercizio). Questo tipo di relazioni monotone tra misure forma la base della fisica qualitativa, un sottoinsieme dell'IA che studia il ragionamento su sistemi fisici senza ricorrere a equazioni dettagliate e simulazioni numeriche. Accenneremo alla fisica qualitativa nelle note storiche alla fine del capitolo.

## Oggetti e sostanze

Si può pensare che il mondo reale consista di oggetti primitivi (particelle) e oggetti composti costruiti per aggregazione. Mantenendo il ragionamento al livello di oggetti grandi come mele e automobili, possiamo gestire la complessità inherente alla trattazione individuale di un grande numero di oggetti primitivi. Tuttavia, una porzione significativa di realtà sembra sfuggire a qualsiasi evidente individuazione, intendendo con questo termine la divisione in oggetti distinti. Daremo a questa porzione il nome generico di roba (stuff). Ad esempio, supponiamo di avere di fronte a noi un po' di burro e un formichiere. Si può dire che il formichiere è uno, ma non c'è un numero evidente di "oggetti-burro", dato che ogni parte di un oggetto-burro è anch'essa un oggetto-burro, almeno finché non arriviamo a particelle davvero infinitesimali. Questa è la differenza principale tra cose e roba. Se tagliamo a metà un formichiere, non ne otteniamo due (sfortunatamente).

Il linguaggio naturale distingue chiaramente cose e roba: infatti diciamo "un formichiere", ma non possiamo dire "un burro". I linguisti distinguono tra sostantivi contabili, come i formichieri, i buchi e i teoremi, e sostantivi collettivi come burro, acqua ed energia. Molte ontologie, in competizione tra loro, sostengono di essere in grado di gestire questa distinzione. Noi ne descriveremo solo una; le altre sono citate nelle note storiche.

Per rappresentare la roba nel modo adeguato, cominciamo dalle cose ovvie. La nostra ontologia dovrà prevedere come oggetti, quantomeno, i "pezzi" di roba con i quali possiamo interagire. Potremmo quindi riconoscere il pezzo di burro come lo stesso pezzo che abbiamo lasciato sul tavolo la notte scorsa; potremmo prenderlo, pesarlo, venderlo, e così via. In questa accezione, il burro è un oggetto proprio come il formichiere. Chiamiamolo *Burro*<sub>3</sub>. Definiamo anche la categoria *Burro*: informalmente i suoi elementi saranno tutte quelle cose di cui si può dire "ehi, è burro", tra cui *Burro*<sub>3</sub>. Lasciando da parte le particelle davvero piccole, che per ora trascureremo, ogni parte di un oggetto-burro è anch'essa un oggetto-burro:

$$\forall x \in \text{Burro} \wedge \text{ParteDi}(y, x) \Rightarrow y \in \text{Burro} .$$

Ora possiamo dire che il burro si scioglie a una temperatura intorno ai 30 gradi centigradi:

$$x \in \text{Burro} \Rightarrow \text{PuntoFusione}(x, \text{Centigradi}(30)) .$$

Potremmo continuare e dire che il burro è giallo, meno denso dell'acqua, morbido a temperatura ambiente, molto grasso, e così via. D'altra parte, il burro non ha una dimensione, una forma o un peso particolari. Possiamo definire categorie specializzate di burro come *BurroSalato*, che sarà anch'esso un tipo di roba. Attenzione però che *ChiloDiBurro*, che include come membri tutti gli oggetti-burro che pesano un chilo, non è una sostanza! Se tagliamo a metà un chilo di burro non otteniamo, ahinoi, due chili di burro.

Tutto questo è dovuto al fatto che alcune proprietà degli oggetti sono intrinseche: non appartengono all'oggetto in sé, ma alla sostanza di cui è fatto. Quando tagliate una sostanza a metà, i due pezzi mantengono tutte le proprietà intrinseche come la densità, il punto di ebollizione, il sapore, il colore, il proprietario etc. D'altra parte, le proprietà estrinseche funzionano nel modo opposto: peso, lunghezza, forma, funzione eccetera non rimangono inalterate dopo una suddivisione.

Una classe di oggetti che include nella sua definizione solo proprietà intrinseche è quindi una sostanza, o sostantivo collettivo; se invece ha anche una sola proprietà estrinseca è un sostantivo contabile. La categoria *Roba* è quella più generale per le sostanze, dato che non specifica alcuna proprietà intrinseca. La categoria *Cosa* è quella più generale per gli oggetti discreti e non specifica alcuna proprietà estrinseca. Tutti gli oggetti fisici appartengono a entrambe, per cui le due categorie sono coestensive, ovvero si riferiscono alle stesse entità.

intrinseche

estrinseche

## 10.3 Azioni, situazioni ed eventi

Il ragionamento sui risultati delle azioni è un aspetto centrale del funzionamento di un agente basato sulla conoscenza. Nel Capitolo 7 abbiamo fornito esempi di formule proposizionali che descrivono l'effetto delle azioni sul mondo del wumpus: ad esempio, l'Equazione (7.3) a pag. 292 specifica come cambia la posizione dell'agente in seguito a un movimento in avanti. Una delle limitazioni della logica proposizionale è la necessità di scrivere una copia della descrizione di un'azione per ogni istante in cui potrebbe essere eseguita. In questo paragrafo descriveremo un metodo di rappresentazione che risolve il problema utilizzando la logica del primo ordine.



## L'ontologia del calcolo delle situazioni

Un modo ovvio di evitare di scrivere copie multiple di assiomi è di quantificare semplicemente sul tempo, scrivendo formule come " $\forall t$ , questo è il risultato in  $t + 1$  dell'esecuzione dell'azione al tempo  $t$ ". Invece di gestire esplicitamente istanti come  $t + 1$ , in questo paragrafo ci concentreremo sulle situazioni, che denotano gli stati risultanti dall'esecuzione di qualche azione. Questo approccio viene chiamato calcolo delle situazioni e si appoggia all'ontologia descritta qui sotto.

- ◆ Come nel Capitolo 8, le azioni sono termini logici come *Avanti* e *Gira(Destra)*. Per adesso ci limiteremo al caso in cui l'ambiente contiene un solo agente: se ce ne fosse più d'uno, sarebbe sufficiente aggiungere un argomento per specificare quale agente sta eseguendo l'azione.
- ◆ Le situazioni sono termini logici che consistono in una situazione iniziale (di solito indicata con  $S_0$ ) e in tutte le situazioni generate dall'applicazione a essa di un'azione. La funzione *Risultato(a, s)* (talvolta chiamata *Do*, dall'inglese *Do*) rappresenta la situazione risultante dall'esecuzione dell'azione  $a$  nella situazione  $s$ . La Figura 10.2 rappresenta questo concetto.
- ◆ I fluenti sono funzioni e predici che variano da una situazione all'altra, come la posizione dell'agente e la condizione di stato in vita del wumpus. Il dizionario dice che un fluente è qualcosa che scorre, come un liquido: in questo caso si "fluisce" tra le situazioni. Per convenzione la situazione è sempre l'ultimo argomento di un fluente. Per esempio,  $\neg \text{Portando}(G_1, S_0)$  afferma che l'agente non sta portando l'oro  $G_1$  nella situazione iniziale  $S_0$ .  $\text{Eta}(Wumpus, S_0)$  si riferisce all'età del wumpus in  $S_0$ .
- ◆ Sono permessi anche predici e funzioni atemporali o eterni. Gli esempi includono il predicato *Oro(G<sub>1</sub>)* e la funzione *GambaSinistra(Wumpus)*.

Oltre che sulle azioni singole, è utile ragionare sulle sequenze di azioni. Possiamo definire i risultati di una sequenza partendo dai risultati delle azioni singole. Prima di tutto diciamo che eseguire una sequenza vuota non cambia la situazione:

$$\text{Risultato}(\emptyset, s) = s$$

Eseguire una sequenza non vuota è la stessa cosa di eseguire la prima azione e poi, nella situazione risultante, tutte le altre:

$$\text{Risultato}([a \text{ seq}], s) = \text{Risultato}(\text{seq}, \text{Risultato}(a, s)).$$

Un agente basato sul calcolo delle situazioni dovrebbe essere capace di dedurre il risultato di una determinata sequenza di azioni; questo compito è chiamato proiezione. Con un adeguato algoritmo di inferenza costruttivo, dovrebbe anche essere in grado di trovare la sequenza che raggiunge un effetto desiderato; quest'attività si chiama pianificazione. Useremo come esempio una versione modificata

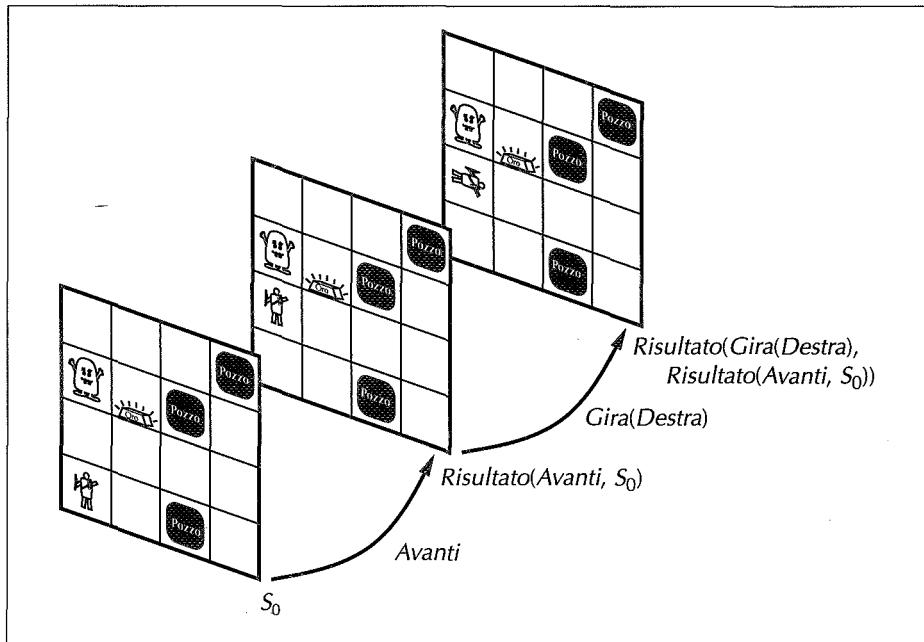
*Situazioni*  
*Calcolo*  
calcolo delle situazioni

situazioni

fluenti

proiezione

planificazione



**Figura 10.2**  
Nel calcolo delle situazioni, ogni situazione (tranne  $S_0$ ) è il risultato di un'azione.

del mondo del wumpus in cui non dobbiamo preoccuparci dell'orientamento dell'agente e in cui possiamo spostarci da una stanza a qualsiasi stanza adiacente. Supponiamo che l'agente si trovi in  $[1, 1]$  e che l'oro sia nella posizione  $[1, 2]$ : lo scopo è portare l'oro in  $[1, 1]$ . I predicati fluenti sono Posizione( $o, x, s$ ) e Portando( $o, s$ ). La base di conoscenza iniziale potrebbe quindi contenere la seguente descrizione:

$$\text{Posizione}(\text{Agente}, [1, 1], S_0) \wedge \text{Posizione}(G_1, [1, 2], S_0).$$

Questo non è abbastanza, comunque, perché non dice quello che non è vero in  $S_0$  (v. pag. 450 per una discussione di quest'aspetto). La descrizione completa è la seguente:

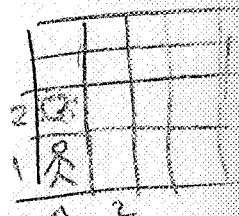
$$\begin{aligned} \text{Posizione}(o, x, S_0) &\Leftrightarrow [(o = \text{Agente} \wedge x = [1, 1]) \vee (o = G_1 \wedge x = [1, 2])] . \quad \text{⑦} \\ \neg \text{Portando}(o, S_0) . \quad \text{⑧} \end{aligned}$$

Dobbiamo anche dire che  $G_1$  è l'oro e che  $[1, 1]$  e  $[1, 2]$  sono adiacenti:

$$\text{Oro}(G_1) \wedge \text{Adiacente}([1, 1], [1, 2]) \wedge \text{Adiacente}([1, 2], [1, 1]) .$$

Gradiremmo dimostrare che l'agente può raggiungere il suo obiettivo andando in  $[1, 2]$ , afferrando l'oro e tornando in  $[1, 1]$ , ovvero

$$\begin{aligned} \text{Posizione}(G_1, [1, 1], \\ \text{Risultato}([Vai([1, 1], [1, 2]), \text{Affera}(G_1), Vai([1, 2], [1, 1])], S_0)) . \end{aligned}$$



Ancora più interessante è la possibilità di costruire un piano per ottenere l'oro, cosa che si può fare rispondendo alla query “quale sequenza di azioni ha come risultato la presenza dell’oro in [1,1]?”

$$\exists \text{seq } \text{Posizione}(G_1, [1, 1], \text{Risultato}(\text{seq}, S_0)).$$

Vediamo cosa dev’essere presente nella base di conoscenza per poter rispondere a interrogazioni come questa.

## Descrivere le azioni nel calcolo delle situazioni

assioma di possibilità  
assioma di effetto

Nella versione più semplice del calcolo delle situazioni ogni azione è descritta da due assiomi: un assioma di possibilità che dice quando è possibile eseguire l’azione e un assioma di effetto che specifica quello che accade quando un’azione possibile è eseguita. Useremo  $\text{Poss}(a, s)$  per indicare che nella situazione  $s$  è possibile eseguire l’azione  $a$ . Gli assiomi hanno la seguente forma:

ASSIOMA DI POSSIBILITÀ:  $\text{Precondizioni} \Rightarrow \text{Poss}(a, s)$ .

ASSIOMA DI EFFETTO:  $\text{Poss}(a, s) \Rightarrow \text{Cambiamenti risultanti dall'esecuzione dell'azione}$ .

Presenteremo questi assiomi per il mondo del wumpus modificato. Per rendere le formule più brevi ometteremo i quantificatori universali che si applicano a un’intera formula. La variabile  $s$  farà riferimento alle situazioni,  $a$  alle azioni,  $o$  agli oggetti (inclusi gli agenti),  $g$  all’oro,  $x$  e  $y$  alle posizioni.

Gli assiomi di possibilità affermano che un agente può spostarsi tra stanze adiacenti, afferrare un pezzo d’oro che si trova nella posizione corrente e lasciar cadere l’oro che sta portando:

$$\begin{array}{ll} \text{Posizione}(\text{Agente}, x, s) \wedge \text{Adiacente}(x, y) & \Rightarrow \text{Poss}(\text{Vai}(x, y), s) . \\ \text{Oro}(g) \wedge \text{Posizione}(\text{Agente}, x, s) \wedge \text{Posizione}(g, x, s) & \Rightarrow \text{Poss}(\text{Afferra}(g), s) . \\ \text{Portando}(g, s) & \Rightarrow \text{Poss}(\text{Lascia}(g), s) . \end{array}$$

Gli assiomi di effetto affermano che, se un’azione è possibile, certe proprietà (fluenti) saranno verificate nella situazione che risulta dall’esecuzione di tale azione. Andare da  $x$  a  $y$  ha come risultato quello di trovarsi in  $y$ , afferrare l’oro ha come risultato il fatto di portarlo, mentre lasciarlo cadere ha come risultato il fatto che non lo si porta più:

$$\begin{array}{l} \text{Poss}(\text{Vai}(x, y), s) \Rightarrow \text{Posizione}(\text{Agente}, y, \text{Risultato}(\text{Vai}(x, y), s)) . \\ \text{Poss}(\text{Afferra}(g), s) \Rightarrow \text{Portando}(g, \text{Risultato}(\text{Afferra}(g), s)) . \\ \text{Poss}(\text{Lascia}(g), s) \Rightarrow \neg \text{Portando}(g, \text{Risultato}(\text{Lascia}(g), s)) . \end{array}$$

Avendo espresso questi assiomi, possiamo dimostrare che il nostro piccolo piano ci permette di raggiungere l’obiettivo? Sfortunatamente no! All’inizio tutto va per il meglio;  $\text{Vai}([1, 1], [1, 2])$  è effettivamente possibile in  $S_0$  e l’assioma di effetto per  $\text{Vai}$  ci permette di concludere che l’agente raggiunge la posizione [1,2]:

$$\text{Posizione}(\text{Agente}, [1, 2], \text{Risultato}(\text{Vai}([1, 1], [1, 2]), S_0)) .$$

Ora consideriamo l'azione *Afferra*( $G_1$ ). Dobbiamo dimostrare che nella nuova situazione questo è possibile, ovvero che

$$\text{Posizione}(G_1, [1, 2], \text{Risultato}(\text{Vai}([1, 1], [1, 2]), S_0)) .$$

Purtroppo non c'è nulla nella base di conoscenza che giustifichi tale conclusione. Intuitivamente sappiamo che l'azione *Vai* non dovrebbe avere alcun effetto sulla posizione dell'oro, che dovrebbe quindi trovarsi ancora nella posizione in cui era in  $S_0$ , e cioè  $[1, 2]$ . Il problema è che gli assiomi di effetto dicono quello che cambia, non quello che rimane uguale.

La rappresentazione di tutte le cose che non cambiano prende il nome di problema del frame.<sup>2</sup> A questo problema occorre trovare una soluzione efficiente perché, nel mondo reale, quasi nulla cambia da una situazione alla successiva: le azioni modificano solo una minima frazione di tutti i fluenti.

Un approccio è scrivere assiomi di frame che affermino *esplicitamente* ciò che non cambia. Il movimento dell'agente ad esempio non modifica la posizione di alcun oggetto, a meno che l'agente non lo stia trasportando:

$$\text{Posizione}(o, x, s) \wedge (o \neq \text{Agente}) \wedge \neg \text{Portando}(o, s) \Rightarrow$$

$$\text{Posizione}(o, x, \text{Risultato}(\text{Vai}(y, z), s)) .$$

Se ci sono  $F$  predicati fluenti e  $A$  azioni, ci serviranno  $O(AF)$  assiomi di frame. D'altra parte, se ogni azione ha al più  $E$  effetti, dove  $E$  tipicamente è molto più piccolo di  $F$ , allora dovremmo essere in grado di rappresentare quello che accade con una base di conoscenza di dimensioni molto inferiori a  $O(AE)$ . Questo è il **problema di rappresentazione del frame**. Il **problema inferenziale del frame**, a esso strettamente imparentato, consiste nel proiettare i risultati di una sequenza di azioni di  $t$  passi in un tempo  $O(Et)$ , invece di  $O(Ft)$  o  $O(AEt)$ . Rimane ancora un altro problema aperto: quello di assicurarsi che siano state specificate *tutte* le condizioni necessarie per il successo di un'azione. Ad esempio, *Vai* fallisce se l'agente muore a metà strada. Questo prende il nome di **problema della qualificazione**, per cui non esiste una soluzione completa.



problema del frame

assiomi di frame

problema di rappresentazione del frame

problema inferenziale del frame

problema della qualificazione

<sup>2</sup> Il nome "problema del frame" deriva dal "frame di riferimento" nella fisica: il fondale (o cornice) immobile rispetto a cui si misura il movimento. Anche in un film o in un cartone animato, tipicamente, i cambiamenti da un frame (fotogramma) all'altro sono molto ridotti.

assiomi di stato  
successore

## Risolvere il problema di rappresentazione del frame

La soluzione del problema di rappresentazione del frame richiede un piccolo cambiamento di prospettiva nella scrittura degli assiomi. Invece di scrivere gli effetti di ogni azione, dobbiamo considerare come evolve nel tempo ogni predicato fluente.<sup>3</sup> Gli assiomi prendono così il nome di **assiomi di stato successore**, e hanno la forma seguente:

ASSIOMA DI STATO SUCCESSORE:

*l'azione è possibile*  $\Rightarrow$

(il fluente è vero nello stato risultante  $\Leftrightarrow$  l'effetto dell'azione l'ha reso vero  
 $\vee$  era già vero e l'azione non l'ha modificato).

Dopo la qualificazione che non stiamo considerando azioni impossibili, notate che questa definizione usa  $\Leftrightarrow$ , non  $\Rightarrow$ . Questo significa che l'assioma afferma che il fluente sarà vero se e solo se è verificata la parte destra. Per dirla in un altro modo, stiamo specificando il valore di verità di ogni fluente nello stato successivo come funzione dell'azione intrapresa e del suo valore nello stato corrente. Questo vuol dire che lo stato successivo è completamente specificato da quello corrente e quindi non occorre aggiungere alcun assioma di frame.

L'assioma di stato successore per la posizione dell'agente afferma che l'agente si trova in  $y$  dopo aver eseguito un'azione se l'azione è possibile e consiste nel muoversi in  $y$  oppure se l'agente si trovava già in  $y$  e l'azione non consiste nell'andarsene da qualche altra parte:

$Poss(a, s) \Rightarrow$

$(Posizione(Agente, y, Risultato(a, s)) \Leftrightarrow a = Vai(x, y)$   
 $\vee (Posizione(Agente, y, s) \wedge a \neq Vai(y, z)))$ .

L'assioma per *Portando* afferma che l'agente sta portando  $g$  dopo l'esecuzione di un'azione se essa consisteva nell'afferrare  $g$  (e questo è possibile) oppure se l'agente stava già portando  $g$  e l'azione non consisteva nel lasciarlo cadere:

$Poss(a, s) \Rightarrow$

$(Portando(g, Risultato(a, s)) \Leftrightarrow a = Afferra(g)$   
 $\vee (Portando(g, s) \wedge a \neq Lascia(g)))$ .

Gli assiomi di stato successore risolvono il problema di rappresentazione del frame perché la dimensione totale degli assiomi è  $O(AE)$  letterali: ognuno degli  $E$  effetti di ogni azione  $A$  è menzionato esattamente una volta. I letterali sono suddivisi su  $F$  assiomi differenti, ragion per cui la dimensione media di un assioma è  $AE/F$ .

<sup>3</sup> Essenzialmente, questo è l'approccio che abbiamo adottato quando abbiamo costruito l'agente basato su circuiti nel Capitolo 7. In effetti, assiomi come l'Equazione (7.4) o l'Equazione (7.5) possono essere considerati assiomi di stato successore.

Il lettore attento avrà già notato che questi assiomi gestiscono il fluente *Posizione* per l'agente, ma non per l'oro; quindi non possiamo ancora dimostrare che il nostro piano in tre passi raggiunge l'obiettivo di portare l'oro nella posizione [1, 1]. Dobbiamo esprimere che un **effetto implicito** del movimento dell'agente da  $x$  a  $y$  è il fatto che con lui si muoverà anche l'oro che sta portando (così come le formiche che camminano sull'oro, i batteri che vivono sulle formiche etc.). La gestione degli effetti impliciti prende il nome di **problema della ramificazione**. Lo discuteremo più avanti: in questo dominio, per risolverlo basta scrivere un assioma di stato successore più generale per *Posizione*. Il nuovo assioma, che rimpiazza la versione precedente, afferma che un oggetto  $o$  si trova in  $y$  se l'agente è andato in  $y$  e  $o$  è l'agente stesso o qualcosa che l'agente stava portando; oppure se  $o$  si trovava già in  $y$  e l'agente non è andato da nessun'altra parte,  $o$  essendo l'agente o qualcosa da esso trasportata.

effetto implicito

problema della  
ramificazione

$$\text{Poss}(a, s) \Rightarrow$$

$$\begin{aligned} \text{Posizione}(o, y, \text{Risultato}(a, s)) \Leftrightarrow & (a = \text{Vai}(x, y) \wedge (o = \text{Agente} \vee \text{Portando}(o, s))) \\ & \vee (\text{Posizione}(o, y, s) \wedge \neg(\exists z \ y \neq z \wedge a = \text{Vai}(y, z) \wedge \\ & (o = \text{Agente} \vee \text{Portando}(o, s))). \end{aligned}$$

A questo punto sorge un ulteriore problema tecnico: un processo di inferenza che usa questi assiomi dev'essere capace di dimostrare le non-identità. Il tipo più semplice di non-identità è quello tra costanti: ad esempio,  $\text{Agente} \neq G_1$ . La semantica generale della logica del primo ordine permette a costanti distinte di far riferimento allo stesso oggetto, per cui la base di conoscenza deve contenere un assioma che lo impedisca. L'**assioma dei nomi unici** afferma la disuguaglianza di ogni coppia di costanti nella base di conoscenza. Quando questo fatto non è scritto esplicitamente nella base di conoscenza, ma dato comunque per scontato dal dimostratore di teoremi, viene chiamata **ipotesi dei nomi unici**. È anche necessario asserire la disuguaglianza tra i termini delle azioni:  $\text{Vai}([1, 1], [1, 2])$  è un'azione diversa da  $\text{Vai}([1, 2], [1, 1])$  o  $\text{Afferra}(G_1)$ . Per far questo per prima cosa occorre dire che ogni tipo di azione è distinto: nessuna azione *Vai* è un'azione *Afferra*. Per ogni coppia di nomi di azione  $A$  e  $B$ , avremo

assioma dei nomi unici

$$A(x_1, \dots, x_m) \neq B(y_1, \dots, y_n).$$

Fatto questo, occorre dire che due termini di azione con lo stesso nome di azione si riferiscono alla stessa azione solo se coinvolgono gli stessi oggetti:

$$A(x_1, \dots, x_m) = A(y_1, \dots, y_m) \Leftrightarrow x_1 = y_1 \wedge \dots \wedge x_m = y_m.$$

Tutti questi sono collettivamente chiamati **assiomi delle azioni distinte**. La combinazione di descrizione dello stato iniziale, assiomi di stato successore, assioma dei nomi unici e assiomi delle azioni distinte è sufficiente a dimostrare che il piano proposto consente di raggiungere l'obiettivo.

assiomi delle azioni  
distinte

calcolo dei fluenti

## Risolvere il problema inferenziale del frame

Gli assiomi di stato successore risolvono il problema di rappresentazione del frame, ma non quello inferenziale. Dato un piano a  $t$  passi  $p$  tale che  $S_t = \text{Risultato}(p, S_0)$ , per decidere quali fluenti sono veri in  $S_t$  occorre considerare ognuno degli  $F$  assiomi di frame in ognuno dei  $t$  istanti temporali. Poiché gli assiomi hanno una dimensione media  $AE/F$ , questo significa un lavoro inferenziale  $O(AEt)$ . La parte più grande di tale lavoro consisterebbe nel copiare fluenti immutati da una situazione alla successiva.

Ci sono due modi di risolvere il problema inferenziale del frame. Per prima cosa potremmo scartare del tutto il calcolo delle situazioni e inventare un nuovo formalismo per scrivere gli assiomi: questo è stato fatto, ad esempio, con il **calcolo dei fluenti**. Un'altra possibilità è alterare il meccanismo di inferenza in modo che gestisca gli assiomi di frame in modo più efficiente. Che questo sia possibile è anche suggerito dalla complessità temporale dell'approccio più ingenuo, che è  $O(AEt)$ : perché mai dovrebbe dipendere dal numero di azioni  $A$ , quando conosciamo precisamente la singola azione eseguita a ogni passo? Per vedere come possiamo migliorare il procedimento consideriamo per prima cosa il formato degli assiomi di frame:

$$\begin{aligned} \text{Poss}(a, s) \Rightarrow \\ F_i(\text{Risultato}(a, s)) &\Leftrightarrow (a = A_1 \vee a = A_2 \dots) \\ &\quad \vee F_i(s) \wedge (a \neq A_3) \wedge (a \neq A_4) \dots \end{aligned}$$

Come si vede, ogni assioma menziona diverse azioni che possono rendere vero il fluente e molte che possono renderlo falso. Possiamo formalizzare tutto ciò introducendo il predicato  $\text{PosEffect}(a, F_i)$ , che significa che l'azione  $a$  fa diventare vero  $F_i$ , e  $\text{NegEffect}(a, F_i)$ , che significa che  $a$  fa diventare  $F_i$  falso. A questo punto possiamo riscrivere il precedente schema di assioma come segue:

$$\begin{aligned} \text{Poss}(a, s) \Rightarrow \\ F_i(\text{Risultato}(a, s)) &\Leftrightarrow \text{PosEffect}(a, F_i) \vee [F_i(s) \wedge \neg \text{NegEffect}(a, F_i)] \\ \text{PosEffect}(A_1, F_i) \\ \text{PosEffect}(A_2, F_i) \\ \text{NegEffect}(A_3, F_i) \\ \text{NegEffect}(A_4, F_i) . \end{aligned}$$

La possibilità di fare tutto ciò in modo automatico dipende dal formato esatto degli assiomi di frame. Affinché una procedura di inferenza che usa assiomi come questo risulti efficiente è necessario fare tre cose.

1. Indicizzare i predici  $\text{PosEffect}$  e  $\text{NegEffect}$  con il loro primo argomento, in modo tale che una volta fornita l'azione eseguita all'istante  $t$  sia possibile determinarne gli effetti in un tempo  $O(1)$ .

2. Indicizzare gli assiomi in modo tale che, una volta appurato che  $F_i$  è un effetto di un'azione, risulti possibile trovare gli assiomi per  $F_i$  in un tempo  $O(1)$ . A questo punto sarà possibile trascurare completamente gli assiomi per i fluenti che non fanno parte degli effetti dell'azione.
3. Rappresentare ogni situazione come la situazione precedente più un delta. Così, se da un passo all'altro non cambia nulla, non sarà necessario svolgere alcun lavoro. Nell'approccio precedente, al contrario, sarebbe stato richiesto un tempo  $O(F)$  per generare un'asserzione per ogni fluente  $F_i(Risultato(a, s))$  dalle precedenti  $F_i(s)$  asserzioni.

In questo modo a ogni passo si guarda l'azione corrente, si recuperano i suoi effetti e si aggiorna l'insieme dei fluenti veri. A ogni iterazione avremo una media di  $E$  aggiornamenti, per una complessità totale  $O(Et)$ . Questa rappresenta una soluzione al problema inferenziale del frame.

## Tempo e calcolo degli eventi

Il calcolo delle situazioni funziona bene quando esiste un solo agente che esegue azioni istantanee e discrete: quando le azioni hanno una durata e possono sovrapporsi, diventa un po' complicato. Di conseguenza, per trattare questi casi useremo un formalismo alternativo noto come **calcolo degli eventi**, che è basato su punti nel tempo piuttosto che situazioni (i termini "evento" e "azione" possono essere usati in modo intercambiabile: informalmente, "evento" connota una classe di azioni più ampie, tra cui quelle che non sono eseguite da un agente esplicito. Queste ultime sono più facili da gestire con il calcolo degli eventi che con quello delle situazioni).

Nel calcolo degli eventi i fluenti sono verificati in alcuni punti nel tempo piuttosto che in determinate situazioni, e il calcolo è progettato per permettere di ragionare su interi intervalli temporali. L'assioma del calcolo degli eventi dice che un fluente è vero in un punto del tempo se è stato iniziato da un evento in qualche punto del passato e non è stato terminato da alcun altro evento. Le relazioni *Inizia* e *Termina* hanno un ruolo simile alla relazione *Risultato* nel calcolo delle situazioni; *Inizia*( $e, f, t$ ) significa che l'occorrenza dell'evento  $e$  al tempo  $t$  ha fatto sì che il fluente  $f$  diventasse vero, mentre *Termina*( $w, f, t$ ) significa che  $f$  cessa di essere vero. Usiamo *Accade*( $e, t$ ) per indicare che l'evento  $e$  accade al tempo  $t$  e *Interrotto*( $f, t, t_2$ ) per indicare che  $f$  è terminato da qualche evento in un istante compreso tra  $t$  e  $t_2$ . Formalmente l'assioma è il seguente:

ASSIOMA DEL CALCOLO DEGLI EVENTI:

$$\begin{aligned} T(f, t_2) &\Leftrightarrow \exists e, t \ Accade(e, t) \wedge Inizia(e, f, t) \wedge (t < t_2) \\ &\quad \wedge \neg Interrotto(f, t, t_2) \\ Interrotto(f, t, t_2) &\Leftrightarrow \exists e, t_1 \ Accade(e, t_1) \wedge Termina(e, f, t_1) \\ &\quad \wedge (t < t_1) \wedge (t_1 < t_2). \end{aligned}$$

calcolo degli eventi

Tutto questo ci fornisce una funzionalità analoga a quella del calcolo delle situazioni con la capacità aggiuntiva di parlare di punti precisi nel tempo e intervalli, cosicché possiamo scrivere *Accade(Spegni(Luce<sub>1</sub>), 1:00)* per dire che la luce è stata spenta esattamente all'una di notte.

Sono state proposte molte estensioni al calcolo degli eventi per gestire effetti indiretti, eventi duraturi, eventi concorrenti, eventi che cambiano con continuità, effetti non deterministici, vincoli causali e altre complicazioni. Ricon sidereremo tra poco alcuni di questi aspetti. È corretto dire che, al momento, per la maggior parte di essi non esistono ancora soluzioni completamente soddisfacenti, ma non sono neppure stati incontrati ostacoli insormontabili.

## Eventi generalizzati

Fin qui abbiamo esaminato due concetti principali: azioni e oggetti. Ora è il momento di vedere come possono convivere in un'ontologia globale in cui sia azioni che oggetti sono considerati aspetti dell'universo fisico. Noi pensiamo che ogni particolare universo abbia una dimensione sia spaziale che temporale. Il mondo del wumpus ha una componente spaziale costituita da una griglia bidimensionale e un tempo discreto; il nostro mondo ha tre dimensioni spaziali e una temporale,<sup>4</sup> tutte continue. Un **evento generalizzato** è composto da alcuni aspetti di un “pezzo di spazio-tempo”: un pezzo, cioè, dell'universo multidimensionale spaziotemporale. Quest'astrazione ci consente di generalizzare la maggior parte dei concetti che abbiamo visto sin qui, tra cui le azioni, le posizioni, gli istanti, i fluenti e gli oggetti fisici. La Figura 10.3 fornisce un'idea generale. D'ora in poi per riferirci agli eventi generalizzati useremo semplicemente il termine “evento”.

La Seconda Guerra Mondiale, ad esempio, è un evento che si è verificato in vari punti dello spazio-tempo, come si vede dalla forma irregolare della macchia grigia. Possiamo suddividerla in **sottoeventi**:<sup>5</sup>

*SottoEvento(BattagliaDInghilterra, SecondaGuerraMondiale)* .

In modo analogo, la Seconda Guerra Mondiale è un sottoevento del ventesimo secolo:

*SottoEvento(SecondaGuerraMondiale, VentesimoSecolo)* .

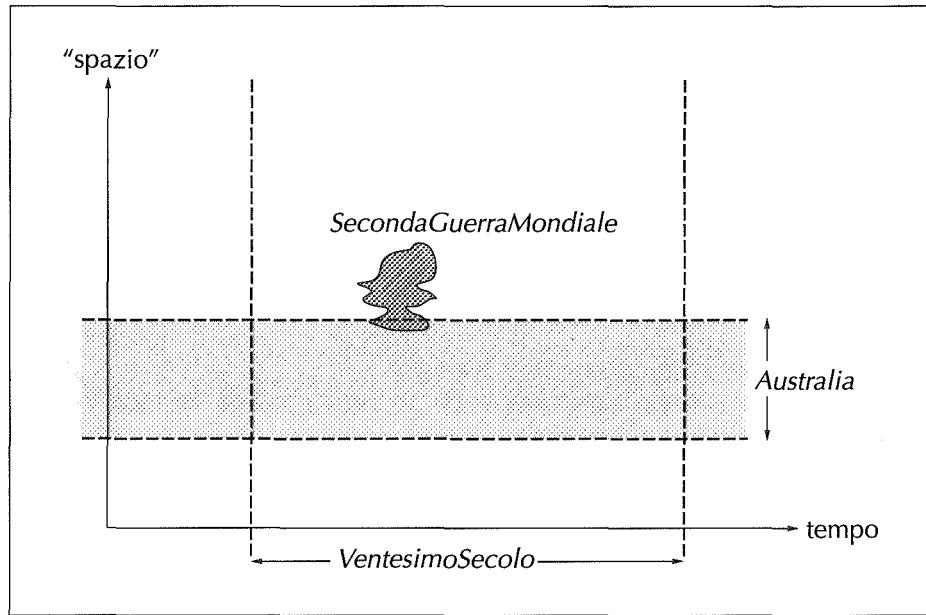
---

<sup>4</sup> Alcuni fisici che studiano la teoria delle stringhe sostengono la necessità di usare 10 o più dimensioni, mentre altri sono favorevoli all'uso di un mondo discreto. Comunque, ai fini del ragionamento comune, un mondo continuo con uno spazio-tempo quadridimensionale è una rappresentazione adeguata.

<sup>5</sup> Notate che *SottoEvento* è un caso speciale della relazione *ParteDi* ed è anch'esso transitivo e riflessivo.

evento generalizzato

sottoeventi



**Figura 10.3** Eventi generalizzati. Un universo ha dimensioni spaziali e temporali; in questa figura rappresentiamo una sola dimensione spaziale. Tutti gli eventi sono *ParteDi* dell'universo. Un evento, come la *SecondaGuerraMondiale*, accade in una porzione di spazio-tempo i cui confini sono in qualche modo arbitrari e mutevoli. Un *Intervallo*, come *VentesimoSecolo*, ha un'estensione temporale limitata e fissa e un'estensione spaziale massima, mentre un *Luogo*, come l'*Australia*, ha un'estensione spaziale più o meno fissa e un'estensione temporale massima.

Il XX secolo è un *intervallo* di tempo. Gli intervalli sono pezzi di spazio-tempo che includono tutto lo spazio compreso tra due istanti temporali. La funzione *Periodo(e)* denota l'intervallo più piccolo che racchiude completamente l'evento *e*. *Durata(i)* è la lunghezza del periodo di tempo occupato da un intervallo, per cui possiamo scrivere *Durata(Periodo(SecondaGuerraMondiale)) > Anni(5)*.

L'Australia è un *luogo*; un pezzo di universo con dei limiti spaziali fissi. I confini possono variare nel tempo, per cause geologiche o politiche. Usiamo il predicato *In* per indicare la relazione di sottoevento che si verifica quando la proiezione spaziale di un evento è *ParteDi* quella di un altro:

*In(Sydney, Australia)*.

La funzione *Locazione(e)* denota il luogo più piccolo che racchiude completamente l'evento *e*. Come ogni altro oggetto, anche gli eventi possono essere raggruppati in categorie: per esempio, *SecondaGuerraMondiale* appartiene alla categoria *Guerre*. Per dire che si è verificata una guerra civile in Inghilterra negli anni '40 del 1600, scriveremo

$\exists w \ w \in \text{GuerreCivili} \wedge \text{SottoEvento}(w, 1640-50) \wedge \text{In}(\text{Locazione}(w), \text{Inghilterra})$ .

Il concetto di categoria di eventi risponde a una domanda che abbiamo evitato quando abbiamo descritto gli effetti delle azioni nel Paragrafo 10.3: a cosa esattamente fanno riferimento termini logici come  $Vai([1, 1], [1, 2])$ ? Sono eventi? La risposta, che potrà forse sorprendervi, è *no*. Per vederlo possiamo considerare un piano con due azioni “identiche”, come

$$[Vai([1, 1], [1, 2]), Vai([1, 2], [1, 1]), Vai([1, 1], [1, 2])] .$$

In questo piano,  $Vai([1, 1], [1, 2])$  non può essere il nome di un evento, perché *ce ne sono due diversi* che accadono in momenti differenti. In effetti,  $Vai([1, 1], [1, 2])$  è il nome di una *categoria* di eventi: e precisamente, di tutti quelli in cui l’agente si muove da  $[1, 1]$  a  $[1, 2]$ . Il nostro piano in tre passi ci dice quindi che avranno luogo tre istanze di categorie di eventi.

Notate che questa è la prima volta che vediamo nomi di categorie costituiti da termini complessi anziché semplici simboli di costante. Questo non rappresenta una fonte di nuove difficoltà; anzi, la struttura degli argomenti può essere usata a nostro vantaggio. L’eliminazione di un argomento può creare una categoria più generale:

$$Vai(x, y) \subseteq VaiA(y) \quad Vai(x, y) \subseteq VaiDa(x) .$$

Analogamente, è possibile aggiungere argomenti per creare categorie più specifiche: ad esempio, possiamo aggiungere un argomento “agente” per descrivere le azioni di altri agenti. In questo modo per dire che Shankar ieri è volato da New York a Nuova Delhi scriveremo:

$$\exists e \ e \in Vola(Shankar, NewYork, NuovaDelhi) \wedge SottoEvento(e, Ieri) .$$

Questa forma è così comune che utilizzeremo un’abbreviazione appositamente creata;  $E(c, i)$  significherà che un elemento della categoria di eventi  $c$  è un sotto-evento dell’evento o dell’intervallo  $i$ :

$$E(c, i) \Leftrightarrow \exists e \ e \in c \wedge SottoEvento(e, i) .$$

Scriveremo quindi:

$$E(Vola(Shankar, NewYork, NuovaDelhi), Ieri) .$$

## Processi

eventi discreti

Gli eventi visti fin qui sono chiamati **eventi discreti** e sono dotati di una struttura definita. Il viaggio di Shankar ha un inizio, una parte centrale e una fine. Se venisse interrotto a metà, l’evento sarebbe diverso: non più un volo da New York a Nuova Delhi, ma da New York a qualche luogo in Europa. D’altra parte, la categoria di eventi denominata *Volando(Shankar)* ha una qualità diversa. Se prendiamo un piccolo intervallo del volo di Shankar, diciamo il terzo segmento di 20 minuti

(mentre sta aspettando ansiosamente un secondo sacchetto di arachidi), quell'evento è ancora membro della categoria *Volando(Shankar)*. Questo è vero per ogni sotto-intervallo.

Categorie di eventi con questa proprietà sono chiamate **processi** o **eventi liquidi**. Ogni sotto-intervallo di un processo è anche membro della stessa categoria del processo. Possiamo usare la stessa notazione degli eventi discreti per dire, ad esempio, che Shankar stava volando in qualche momento di ieri:

$$E(\text{Volando}(\text{Shankar}), \text{Ieri}) .$$

Spesso vogliamo dire che un processo si è svolto *attraverso tutto* un intervallo piuttosto che in qualche sotto-intervallo compreso in esso. Per far questo usiamo il predicato *T*:

$$T(\text{Lavorando}(\text{Stuart}), \text{OggiInPausaPranzo}) .$$

*T(c, i)* significa che un evento di tipo *c* è avvenuto esattamente nell'intervallo *i*, intendendo con questo che il suo inizio e la sua fine coincidono con quelli dell'intervallo stesso.

La distinzione tra eventi liquidi e non liquidi è esattamente analoga a quella tra le sostanze, o *roba*, e gli oggetti individuali. In effetti, alcuni hanno chiamato gli eventi liquidi **sostanze temporali**, riservando alle cose come il burro il nome di **sostanze spaziali**.

Oltre a descrivere i processi con cambiamento continuo, gli eventi liquidi possono anche descrivere processi in cui nulla cambia mai. Spesso questi vengono chiamati **stati**. “Shankar rimane a New York”, ad esempio, è una categoria di stati che indichiamo con *In(Shankar, New York)*. Per dire che è rimasto a New York tutto il giorno, scriveremo

$$T(\text{In}(\text{Shankar}, \text{New York}), \text{Oggi}) .$$

Possiamo costruire stati ed eventi più complessi combinando quelli primitivi: è l'approccio adottato dal **calcolo dei fluenti**. Questo calcolo reifica combinazioni di fluenti, e non solo fluenti singoli. Un modo di rappresentare l'evento in cui due cose accadono nello stesso momento è la funzione *Entrambi(e<sub>1</sub>, e<sub>2</sub>)*. Nel calcolo dei fluenti si può usare per comodità la notazione infissa *e<sub>1</sub> o e<sub>2</sub>*. Per dire che qualcuno cammina e mastica la gomma contemporaneamente, quindi, possiamo scrivere

$$\exists p, i \quad (p \in \text{Persone}) \wedge T(\text{Cammina}(p) \circ \text{MasticaGomma}(p), i) .$$

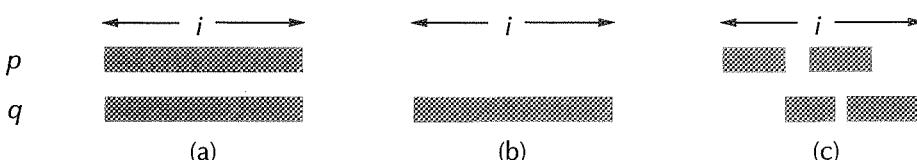
La funzione “*o*” è commutativa e associativa, come ogni congiunzione logica. Possiamo anche definire operatori analoghi alla disgiunzione e alla negazione, ma dobbiamo stare attenti: infatti ci sono due modi diversi, ma entrambi ragionevoli, di interpretare la disgiunzione. Quando diciamo “l'agente stava camminando o masticando gomma per gli ultimi due minuti” potremmo intendere che faceva l'una o l'altra azione per l'intero intervallo, oppure che alternava l'esecuzione delle due. Per indicare queste due possibilità useremo rispettivamente *UnaDi* e *UnaOAltra*. I diagrammi nella Figura 10.4 esemplificano gli eventi complessi.

processi  
eventi liquidi

sostanze temporali  
sostanze spaziali

stati

calcolo dei fluenti



**Figura 10.4** Una rappresentazione grafica di eventi complessi. (a)  $T(Entrambi(p, q), i)$ , indicato anche con  $T(p \circ q, i)$ . (b)  $T(\lnot p \wedge q, i)$ . (c)  $T(\lnot p \vee q, i)$ .

## Intervalli

Per ogni agente che esegue azioni il tempo è molto importante; per questa ragione sono stati profusi molti sforzi nella rappresentazione degli intervalli temporali. Ne considereremo due tipi: i momenti e gli intervalli estesi. La distinzione sta nel fatto che i momenti hanno una durata pari a zero:

*Partizione*( $\{Momenti, IntervalliEstesi\}$ ,  $Intervalli$ )  
 $i \in Momenti \Leftrightarrow Durata(i) = Secondi(0)$  .

Per avere tempi assoluti è necessario inventare una scala temporale e associare i punti su di essa ai momenti. La scala è arbitraria; la misureremo in secondi e stabiliremo che l'istante iniziale, di tempo 0, è la mezzanotte (GMT) dell'1 gennaio 1900. Le funzioni *Inizio* e *Fine* indicano rispettivamente il primo e l'ultimo momento di un intervallo, e la funzione *Tempo* restituisce il riferimento sulla scala temporale assoluta di un dato momento. La funzione *Durata* rappresenta la differenza tra i tempi di fine e di inizio.

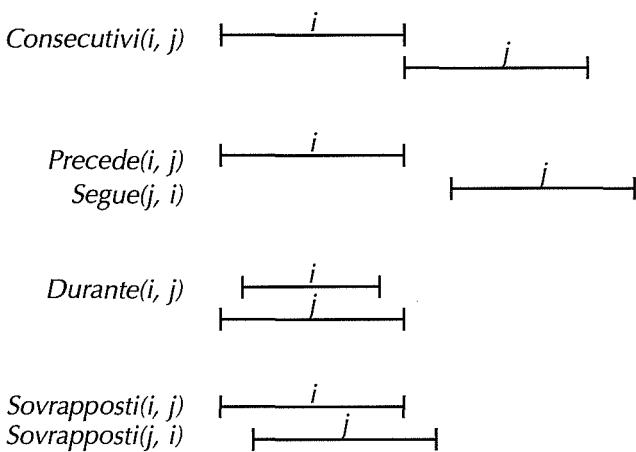
$\text{Intervallo}(i) \Rightarrow \text{Durata}(i) = (\text{Tempo}(\text{Fine}(i)) - \text{Tempo}(\text{Inizio}(i)))$  .  
 $\text{Tempo}(\text{Inizio}(\text{AD}1900)) = \text{Secondi}(0)$  .  
 $\text{Tempo}(\text{Inizio}(\text{AD}2001)) = \text{Secondi}(3187324800)$  .  
 $\text{Tempo}(\text{Fine}(\text{AD}2001)) = \text{Secondi}(3218860800)$  .  
 $\text{Durata}(\text{AD}2001) = \text{Secondi}(31536000)$  .

Per aumentare la leggibilità di questi valori possiamo introdurre una funzione *Data*, che prende sei argomenti (ore, minuti, secondi, giorno, mese e anno) e restituisce un punto sulla scala temporale:

$Tempo(\\text{Inizio}(AD2001)) = Data(0, 0, 0, 1, gen, 2001)$   
 $Data(0, 20, 21, 24, 1, 1995) = Secondi(3000000000)$

Due intervalli sono *Consecutivi* se la fine del primo coincide con l'inizio del secondo. È possibile definire predicati come *Precede*, *Segue*, *Durante* e *Sovraposti* basandosi unicamente sulla definizione di *Consecutivi*, ma è più intuitivo definirli facendo riferimento ai punti sulla scala temporale: la Figura 10.5 fornisce una rappresentazione grafica.

**Figura 10.5**  
Predicati sugli  
intervalli  
temporali.



$$\begin{array}{ll}
 \text{Consecutivi}(\textit{i}, \textit{j}) & \Leftrightarrow \text{Tempo}(\text{Fine}(\textit{i})) = \text{Tempo}(\text{Inizio}(\textit{j})) . \\
 \text{Precede}(\textit{i}, \textit{j}) & \Leftrightarrow \text{Tempo}(\text{Fine}(\textit{i})) < \text{Tempo}(\text{Inizio}(\textit{j})) . \\
 \text{Segue}(\textit{j}, \textit{i}) & \Leftrightarrow \text{Precede}(\textit{i}, \textit{j}) . \\
 \text{Durante}(\textit{i}, \textit{j}) & \Leftrightarrow \text{Tempo}(\text{Inizio}(\textit{j})) \leq \text{Tempo}(\text{Inizio}(\textit{i})) \\
 & \wedge \text{Tempo}(\text{Fine}(\textit{i})) \leq \text{Tempo}(\text{Fine}(\textit{j})) . \\
 \text{Sovraposti}(\textit{i}, \textit{j}) & \Leftrightarrow \exists k \text{ Durante}(k, \textit{i}) \wedge \text{Durante}(k, \textit{j}) .
 \end{array}$$

Ad esempio, per dire che il regno di Elisabetta II ha seguito quello di Giorgio VI, e quello di Elvis si è sovrapposto con gli anni '50, possiamo scrivere quanto segue:

$$\begin{aligned}
 &\text{Segue}(\text{RegnoDi}(\text{ElisabettaII}), \text{RegnoDi}(\text{GiorgioVI})) . \\
 &\text{Sovraposti}(\text{Anni50}, \text{RegnoDi}(\text{Elvis})) . \\
 &\text{Inizio}(\text{Anni50}) = \text{Inizio}(\text{AD1950}) . \\
 &\text{Fine}(\text{Anni50}) = \text{Fine}(\text{AD1959}) .
 \end{aligned}$$

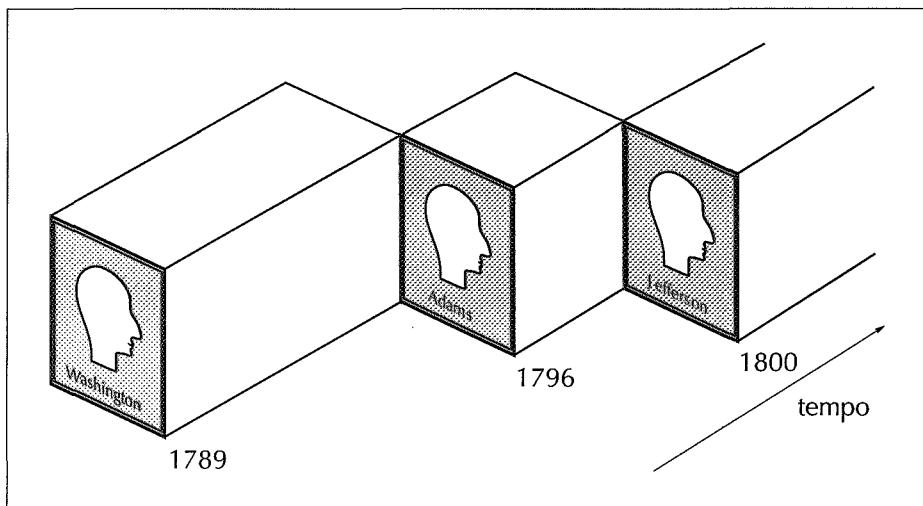
## Fluenti e oggetti

Abbiamo già menzionato il fatto che gli oggetti fisici possono essere considerati eventi generalizzati, nel senso che un oggetto fisico è anch'esso un pezzo di spazio-tempo. Gli Stati Uniti, per esempio, possono essere considerati un evento cominciato nel 1776 come unione di 13 stati e ancora in atto oggi come unione di 50.

Possiamo descrivere le proprietà in evoluzione di *StatiUniti* per mezzo dei fluenti di stato. Per esempio, per dire che nel 1999 la popolazione era di 271 milioni scriveremo:

$$E(\text{Popolazione}(\text{StatiUniti}, 271000000), \text{AD1999}) .$$

**Figura 10.6**  
 Una visione schematica dell'oggetto *Presidente(USA)* per i primi 15 anni della sua esistenza.



Un'altra proprietà degli Stati Uniti è che, salvo imprevisti, il presidente cambia ogni quattro od ogni otto anni. Si potrebbe pensare che il termine logico *Presidente(StatiUniti)* possa indicare oggetti diversi in tempi differenti: sfortunatamente non è così, dato che in un modello un termine denota esattamente un oggetto. Il termine *Presidente(StatiUniti, t)* può riferirsi a oggetti diversi a seconda del valore di  $t$ , ma la nostra ontologia tiene separati indici temporali e fluenti. L'unica possibilità è che *Presidente(StatiUniti)* denoti effettivamente un singolo oggetto, ma che questo consista di persone diverse in tempi diversi: sarà George Washington dal 1789 al 1796, John Adams dal 1796 al 1800 e così via, come si vede nella Figura 10.6. Per dire che George Washington è stato presidente per tutto il 1790, possiamo scrivere

$$T(\text{Presidente}(\text{StatiUniti})) = \text{George Washington, AD1790}.$$

Comunque, occorre stare attenti. Nella formula qui sopra “=” dev’essere un simbolo di funzione e non l’operatore logico standard. L’interpretazione *non* è che *George Washington* e *Presidente(StatiUniti)* sono logicamente identici nel 1790; l’identità logica del resto non è qualcosa che può cambiare con il tempo. L’identità logica che esiste effettivamente è quella tra i sottoeventi dei due oggetti identificati dal periodo 1790.

Non dovete neppure confondere l’oggetto fisico *George Washington* con una collezione di atomi. *George Washington* non è logicamente identico ad *alcuna* collezione di atomi, dato che l’insieme di atomi che lo costituisce varia considerevolmente nel tempo. Del resto Washington ha un lasso di vita piuttosto breve rispetto a quello degli atomi, che continuano a esistere per un tempo molto lungo. Quello che si può dire è che per un certo periodo le loro esistenze si intersecano, e durante quel periodo la “fetta” temporale di ogni atomo è *ParteDi* *George Washington*. Finito il periodo, ognuno va per la sua strada.

## 10.4 Eventi e oggetti mentali

Gli agenti che abbiamo costruito fin qui hanno credenze, e ne possono dedurre di nuove. Tuttavia, nessuno di essi possiede conoscenza *che riguarda* le credenze o lo stesso processo di deduzione. Nei domini ad agente singolo, la conoscenza riguardante la propria stessa conoscenza e i propri processi di ragionamento può essere utile per controllare l'inferenza. Per esempio, se uno sa di non sapere nulla sulla geografia rumena, non dovrà compiere enormi sforzi computazionali per cercare di calcolare il cammino più breve da Arad a Bucarest. L'agente potrebbe anche ragionare sulla propria conoscenza in modo da costruire un piano per modificarla: ad esempio, comprando una mappa della Romania. Nei domini multiagente, diventa importante ragionare sugli stati mentali degli altri agenti. Ad esempio, un vigile rumeno probabilmente conoscerà la strada migliore per Bucarest, così l'agente potrebbe decidere di chiedergli aiuto.

In pratica, quello che ci serve è un modello degli oggetti mentali nella testa delle persone (o nella base di conoscenza degli agenti) e dei processi mentali che manipolano tali oggetti. Il modello dev'essere fedele, ma non è necessario che sia dettagliato. Non occorre prevedere quanti millisecondi serviranno a un agente per dedurre un fatto, né quali neuroni si attiveranno in un animale sottoposto a un particolare stimolo visivo. Ci basterà concludere che il vigile rumeno saprà spiegarci come arrivare a Bucarest, se conosce la strada e si è convinto che ci siamo persi.

### Una teoria formale delle credenze

Cominceremo a trattare le relazioni tra agenti e "oggetti mentali" come *Crede*, *Conosce* e *Desidera*. Relazioni di questo tipo sono chiamate **attitudini proposizionali**, dato che descrivono l'attitudine di un agente nei confronti, appunto, di una proposizione. Supponiamo che Lois creda qualcosa: *Crede(Lois, x)*. Che cos'è *x*? Chiaramente non può essere una formula logica. Infatti se *Vola(Superman)* è una formula logica non possiamo scrivere *Crede(Lois, Vola(Superman))*, perché come argomenti dei predicati possiamo avere solo termini e non formule. Ma se *Vola* è una funzione, allora *Vola(Superman)* è un possibile candidato a essere un oggetto mentale, e *Crede* può essere una relazione tra un agente e un fluente proposizionale. Trasformare una proposizione in un oggetto si chiama **reificazione**.<sup>6</sup>

Questo sembra fornirci ciò che desideriamo: la capacità, da parte di un agente, di ragionare sulle credenze degli agenti. Sfortunatamente, quest'approccio presenta un problema: se Clark e Superman sono lo stesso oggetto, cioè *Clark = Superman*, al-

attitudini proposizionali

reificazione

<sup>6</sup> Il termine "reificazione" deriva dal latino *res*, che significa "cosa". John McCarthy ha proposto il termine *thingification*, letteralmente "cosificazione", ma l'idea non ha mai preso piede.

lora il volo di Clark e quello di Superman sono la stessa categoria di eventi, cioè  $Vola(Clark) = Vola(Superman)$ . Dobbiamo quindi concludere che se Lois crede che Superman possa volare, deve anche credere che Clark possa volare, *anche se non crede che Clark sia Superman*:

$$(Superman = Clark) \models \\ (Crede(Lois, Vola(Superman)) \Leftrightarrow Crede(Lois, Vola(Clark))) .$$

In un certo senso questo è anche giusto: Lois crede effettivamente che una certa persona, che talvolta viene chiamata Clark, possa volare. Ma in un altro senso non è giusto affatto: infatti se chiedessimo a Lois “Clark può volare?” risponderebbe certamente di no. Gli oggetti e gli eventi reificati funzionano bene se adottiamo il primo significato di *Crede*, ma per il secondo dovremmo reificare le *descrizioni* di tali oggetti ed eventi, in modo che *Clark* e *Superman* possano essere descrizioni differenti (anche se si riferiscono allo stesso oggetto).

Tecnicamente, la possibilità di sostituire liberamente un termine con un termine uguale si chiama **trasparenza referenziale**. Nella logica del primo ordine, ogni relazione è referenzialmente trasparente. Ci piacerebbe definire *Crede* (e le altre attitudini proposizionali) come relazioni il cui secondo argomento è referenzialmente **opaco**, e non può quindi essere sostituito da un termine uguale senza cambiare il significato della formula.

Ci sono due modi di far questo. Il primo è usare una forma diversa di logica chiamata **logica modale**, in cui attitudini proposizionali come *Crede* e *Conosce* diventano **operatori modali** referenzialmente opachi. Vi accenneremo nelle note storiche. Il secondo approccio, che svilupperemo qui, è ottenere un’effettiva opacità all’interno di un linguaggio referenzialmente trasparente utilizzando una **teoria sintattica** degli oggetti mentali. Questo significa che gli oggetti mentali saranno rappresentati da **stringhe**. Il risultato è un rozzo modello della base di conoscenza di un agente fatto di stringhe che rappresentano le formule in cui l’agente crede. Una stringa non è altro che un termine complesso che denota una lista di simboli, così l’evento *Vola(Clark)* può essere rappresentato dalla lista di caratteri  $[V, o, l, a, (, C, l, a, r, k,)]$ , che abbrevieremo con la stringa racchiusa tra doppi apici “*Vola(Clark)*”. La teoria sintattica include un **assioma della stringa unica** che afferma che due stringhe sono identiche se e solo se sono composte da caratteri identici. In questo modo, anche se *Clark* = *Superman*, abbiamo ancora “*Clark*” ≠ “*Superman*”.

Ora tutto quello che dobbiamo fare è fornire una sintassi, una semantica e una teoria della dimostrazione per il linguaggio di rappresentazione delle stringhe, come abbiamo fatto nel Capitolo 7. La differenza sta nel fatto che dovremo definire tutto ciò in logica del primo ordine. Cominceremo con il definire *Den* come la funzione che mette in corrispondenza una stringa con l’oggetto da essa denotato, e *Nome* come la funzione che mette in corrispondenza un oggetto con la stringa che rappresenta il nome della costante che lo denota. Ad esempio, la denotazione di

trasparenza  
referenziale

opaco

logica modale  
operatori modali

teoria sintattica

stringhe

assioma della stringa  
unica

“Clark” e di “Superman” è in entrambi i casi l’oggetto a cui fa riferimento il simbolo di costante *UomoDAcciaio*, e il nome di tale oggetto nella base di conoscenza potrebbe essere “*Superman*”, “*Clark*” o anche un’altra costante come “*X<sub>11</sub>*”:

$$\begin{aligned} \text{Den}(\text{"Clark"}) &= \text{UomoDAcciaio} \wedge \text{Den}(\text{"Superman"}) = \text{UomoDAcciaio}. \\ \text{Nome}(\text{UomoDAcciaio}) &= \text{"X}_{11}\text{".} \end{aligned}$$

Il passo successivo è definire regole di inferenza per gli agenti logici. Ad esempio, potremmo dire che un agente logico può applicare il Modus Ponens: se crede *p* e crede *p*  $\Rightarrow$  *q*, allora crederà anche *q*. Un primo tentativo di scrivere quest’assioma potrebbe essere

$$\text{AgenteLogico}(a) \wedge \text{Crede}(a, p) \wedge \text{Crede}(a, "p \Rightarrow q") \Rightarrow \text{Crede}(a, q).$$

Questo però non è corretto, perché la stringa “*p*  $\Rightarrow$  *q*” contiene le lettere ‘*p*’ e ‘*q*’ ma non ha nulla a che vedere con le stringhe che costituiscono i valori delle variabili *p* e *q*. La formulazione corretta è

$$\begin{aligned} \text{AgenteLogico}(a) \wedge \text{Crede}(a, p) \wedge \text{Crede}(a, \text{Concat}(p, "\Rightarrow", q)) \\ \Rightarrow \text{Crede}(a, q). \end{aligned}$$

dove *Concat* è una funzione che concatena due stringhe. D’ora in poi abbrevieremo *Concat*(*p*, “ $\Rightarrow$ ”, *q*) come “*p*  $\Rightarrow$  *q*”. Questo significa che l’occorrenza di una variabile *x* all’interno di una stringa dovrà sempre essere sostituita dal corrispondente valore di *x*: i programmatori Lisp riconosceranno il meccanismo dell’operatore comma/apostrofo inverso, mentre ai programmatori Perl ricorderà l’interpolazione delle \$-variabili.

Una volta che avremo aggiunto altre regole di inferenza oltre al Modus Ponens, saremo in grado di rispondere a domande come “dato un agente logico che conosce queste premesse, è possibile che tragga questa conclusione?”. Oltre alle normali regole di inferenza, dovremo aggiungerne altre specifiche per la credenza. Ad esempio, la regola seguente afferma che se un agente logico crede in qualcosa, allora crede anche di crederci.

$$\text{AgenteLogico}(a) \wedge \text{Crede}(a, p) \Rightarrow \text{Crede}(a, \text{"Crede"}(\text{Nome}(a), p)).$$

Ora, in base ai nostri assiomi, un agente può dedurre infallibilmente tutte le conseguenze delle sue credenze. Questa prende il nome di **onniscienza logica**. Sono stati fatti molti tentativi di definire agenti razionali limitati, in grado di eseguire un numero limitato di inferenze in un dato lasso di tempo. Nessuna soluzione è completamente soddisfacente, ma tali formulazioni permettono di restringere sensibilmente l’estensione delle predizioni sugli agenti limitati.

onniscienza logica

## Conoscenza e credenza

La relazione tra credenza e conoscenza è stata studiata approfonditamente in filosofia. Generalmente si dice che la conoscenza è una credenza la cui verità è giustificabile: questo significa che se una persona crede qualcosa per una ragione inoppugnabile, e quella cosa è effettivamente vera, allora la conosce. La “ragione inoppugnabilmente buona” è necessaria per impedire che qualcuno affermi “so che tirando questa moneta verrà testa” e sia nel giusto la metà delle volte.

Sia  $Conosce(a, p)$  la formula che indica che un agente  $a$  sa che la proposizione  $p$  è vera. È anche possibile definire altri tipi di conoscenza; ad esempio, ecco la definizione di “sapere se”:

$$ConosceSe(a, p) \Leftrightarrow Conosce(a, p) \vee Conosce(a, \neg p).$$

Riprendendo il nostro esempio, Lois “sa se” Clark può volare se sa che può farlo oppure se sa che non può. Il concetto di “conoscere una cosa” è più complicato. Potremmo essere tentati di dire che un agente conosce il numero di telefono di Bob se esiste un  $x$  tale che l’agente sa che  $Numero\,Telefono(Bob) = x$ . Ma questo non è sufficiente, perché l’agente potrebbe sapere che Alice e Bob hanno lo stesso numero (quindi  $Numero\,Telefono(Bob) = Numero\,Telefono(Alice)$ ), ma se non conosce il numero di Alice non gli sarà molto utile. Una definizione migliore di “conoscere una cosa” specifica che un agente deve essere a conoscenza di un  $x$  che è una stringa di cifre ed è anche il numero di Bob:

$$\begin{aligned} ConosceCosa(a, "Numero\,Telefono(\underline{b})) &\Leftrightarrow \\ \exists x \ Conosce(a, "\underline{x} = Numero\,Telefono(\underline{b})) \wedge x \in StringheDiCifre. \end{aligned}$$

Naturalmente, nel caso di domande diverse cambieranno anche i criteri per decidere se una risposta è accettabile. Per la domanda “qual è la capitale dello stato di New York”, una risposta accettabile è un nome come “Albany”, non qualcosa come “la città in cui ha sede la camera legislativa”. Per gestire tutto questo  $ConosceCosa$  sarà una relazione ternaria che prende come argomenti un agente, una stringa che rappresenta un termine e una categoria a cui deve appartenere la risposta. Avremo quindi formule come le seguenti:

$$\begin{aligned} ConosceCosa(Agente, "Capitale(New\,York)", NomiDiCitt\ddot{a}) \\ ConosceCosa(Agente, "Numero\,Telefono(Bob)", StringheDiCifre). \end{aligned}$$

## Conoscenza, tempo e azione

Nella maggior parte delle situazioni reali, un agente dovrà gestire credenze (proprie o di altri agenti) che cambiano nel tempo. L’agente dovrà anche formulare piani che coinvolgono dei cambiamenti alla sua stessa conoscenza, come comprare una mappa per scoprire la strada per Bucarest. Come con gli altri predicati, possiamo reificare  $Crede$  e parlare delle credenze nel tempo. Per esempio, per dire che Lois crede oggi che Superman possa volare, scriviamo

$$T(Crede(Lois, "Vola(Superman)"), Oggi).$$

Se l'oggetto della credenza è una proposizione che può cambiare nel tempo, anch'essa può essere descritta usando l'operatore  $T$  all'interno della stringa. Lois potrebbe credere oggi che Superman poteva volare ieri:

$$T(Crede(Lois, "T(Vola(Superman), Ieri)", Oggi)).$$

Avendo la possibilità di descrivere le credenze nel tempo, possiamo usare il calcolo degli eventi per formulare piani che coinvolgono credenze. Le azioni possono avere **precondizioni di conoscenza** ed **effetti sulla conoscenza**. Ad esempio, l'azione di comporre il numero di telefono di una persona ha la precondizione di conoscerlo, e l'azione di cercare il numero sulla guida ha come effetto la sua conoscenza. Possiamo descrivere quest'ultima azione usando il calcolo degli eventi:

$$\begin{aligned} &Inizia(Cerca(a, "Numero\,Telefono(b)"), \\ &\quad ConosceCosa(a, "Numero\,Telefono(b)", StringheDiCifre), t). \end{aligned}$$

I piani che coinvolgono la raccolta e l'uso d'informazione sono spesso rappresentati usando la notazione abbreviata delle **variabili runtime**, molto simile alla convenzione che abbiamo descritto poco fa riguardo alla sostituzione delle variabili all'interno delle stringhe. Così, il piano consistente nel cercare il numero di Bob e chiamarlo potrà essere scritto

$$[Cerca(Agente, "Numero\,Telefono(Bob)", \underline{n}), Componi(\underline{n})].$$

Qui,  $\underline{n}$  è una variabile runtime il cui valore sarà legato dall'azione *Cerca* e usato dall'azione *Componi*. Piani di questo tipo si verificano frequentemente in domini parzialmente osservabili: ne vedremo alcuni esempi nel prossimo paragrafo e nel Capitolo 12.

precondizioni di conoscenza  
effetti sulla conoscenza

variabili runtime

## 10.5 Il mondo dello shopping su Internet

In questo paragrafo codificheremo conoscenza relativa allo shopping su Internet e costruiremo un agente di ricerca che possa aiutare un acquirente a trovare sulla rete offerte di prodotti. L'acquirente deve fornire una descrizione del prodotto, e l'agente per lo shopping ha il compito di produrre una lista di pagine web che ne offrono la vendita. In alcuni casi la descrizione del prodotto da parte dell'utente sarà precisa, come in *macchina fotografica digitale Coolpix 995*, e il compito si ridurrà alla ricerca del negozio (o negozi) che offrono il prezzo migliore. In altri casi la descrizione sarà specificata solo parzialmente, come in *macchina fotografica digitale di costo inferiore a 300 dollari*, e l'agente dovrà confrontare prodotti differenti.

L'ambiente dell'agente per lo shopping è costituito dall'intero World Wide Web: non un ambiente giocattolo simulato, ma lo stesso ambiente complesso e in costante evoluzione utilizzato ogni giorno da milioni di persone. Le percezioni dell'agente sono pagine web, ma laddove un utente umano le vedrebbe presentate in forma grafica, l'agente le percepisce come stringhe di caratteri comprendenti parole

## Generico Negozio Online

Scegli nella nostra sfiziosa linea di prodotti:

- Computer
- Fotocamere
- Libri
- Video
- Musica

```
<h1>Generico Negozio Online</h1>
<i>Scegli</i> nella nostra sfiziosa linea di prodotti:

 Computer
 Fotocamere
 Libri
 Video
 Musica

```

**Figura 10.7** La pagina web di un generico negozio online nella forma percepita dall’utente umano di un browser (in alto), e la corrispondente stringa HTML così com’è percepita dal browser stesso o da un agente per lo shopping (in basso). In HTML, i caratteri compresi tra `<e>` sono direttive di *markup* che specificano l’aspetto grafico della pagina. Ad esempio, la stringa `<i>Scegli</i>` significa di passare al corsivo, visualizzare la parola *Scegli* e terminare l’uso del corsivo. Un identificatore di pagina come `http://gen-store.com/books` è chiamato **Uniform Resource Locator** o **URL**. Il markup `<a href="url">&ancora</a>` crea un collegamento ipertestuale diretto a *url* contraddistinto dal testo *&ancora*.

comuni mescolate a etichette del linguaggio HTML. La Figura 10.7 mostra una pagina web e la corrispondente stringa di caratteri HTML; l’agente dovrà essere in grado di estrarre informazioni utili da questo tipo di percezioni.

Interpretare la percezione di una pagina web è palesemente più facile di interpretare, poniamo, le percezioni ricevute mentre si guida un taxi al Cairo. Ciononostante anche in questo caso ci sono delle complicazioni. La pagina web della Figura 10.7 è molto semplice in confronto a quelle dei siti reali, che possono includere cookie, codice Java, Javascript e Flash, protocolli di esclusione dei robot, HTML scorretto, file sonori, filmati e testo integrato nelle immagini grafiche. Un agente capace di gestire *tutti* questi aspetti di Internet è quasi complesso come un robot che si muove nel mondo reale: noi ci concentreremo su un agente semplice, che ignora gran parte di queste complicazioni.

Il primo compito dell’agente è trovare offerte di prodotti rilevanti (vedremo più avanti come scegliere la migliore tra queste). Sia *query* la descrizione del pro-

dotto inserita dall'utente (ad es. "portatile"); allora la pagina è un'offerta rilevante per *query* se è rilevante ed è effettivamente un'offerta. Terremo anche traccia dell'URL della pagina:

$$\text{OffertaRilevante}(\textit{pagina}, \textit{url}, \textit{query}) \Leftrightarrow \text{Rilevante}(\textit{pagina}, \textit{url}, \textit{query}) \\ \wedge \text{Offerta}(\textit{pagina}) .$$

Una pagina con la recensione dei modelli più recenti di portatile sarebbe rilevante, ma se non offre la possibilità di acquistare, non è un'offerta. Per ora, limitiamoci a dire che una pagina è un'offerta se contiene le parole "acquista" o "prezzo" all'interno di un collegamento o di un form HTML. In altre parole, se la pagina contiene una stringa nel formato "<a . . . acquista . . . </a>" allora è un'offerta; al posto di "acquista" potrebbe essere scritto "prezzo" e al posto di "a" potremmo avere "form". Possiamo scrivere gli assiomi seguenti.

$$\begin{aligned} \text{Offerta}(\textit{pagina}) &\Leftrightarrow (\text{InTag}("a", \textit{str}, \textit{pagina}) \vee \text{InTag}("form", \textit{str}, \textit{pagina})) \\ &\quad \wedge (\text{In}("acquista", \textit{str}) \vee \text{In}("prezzo", \textit{str})) . \\ \text{InTag}(\textit{tag}, \textit{str}, \textit{pagina}) &\Leftrightarrow \text{In}(" < " + \textit{tag} + \textit{str} + "< /" + \textit{tag}, \textit{pagina}) . \\ \text{In}(\textit{sub}, \textit{str}) &\Leftrightarrow \exists i \text{ str } [i : i + \text{Lunghezza}(\textit{sub})] = \textit{sub} . \end{aligned}$$

Ora dobbiamo trovare le pagine rilevanti. La strategia è cominciare dalla home page di un negozio online e considerare tutte le pagine che possono essere raggiunte seguendo collegamenti rilevanti.<sup>7</sup> L'agente conoscerà un certo numero di siti, ad esempio:

$$\begin{aligned} \text{Amazon} &\in \text{NegoziOnline} \wedge \text{Homepage}(\text{Amazon}, \text{"amazon.com"}) . \\ \text{Ebay} &\in \text{NegoziOnline} \wedge \text{Homepage}(\text{Ebay}, \text{"ebay.com"}) . \\ \text{GenStore} &\in \text{NegoziOnline} \wedge \text{Homepage}(\text{GenStore}, \text{"gen-store.com"}) . \end{aligned}$$

Questi siti classificano i prodotti in categorie, fornendo collegamenti a quelle più importanti direttamente dalla home page. Le sottocategorie possono essere raggiunte seguendo una catena di collegamenti rilevanti, che a un certo punto giungeranno fino alle offerte. In altre parole, una pagina è rilevante per la query se può essere raggiunta da una catena di collegamenti di categoria rilevanti partendo dalla home page di un negozio online e se da lì si può raggiungere un'offerta di prodotto:

$$\begin{aligned} \text{Rilevante}(\textit{pagina}, \textit{url}, \textit{query}) &\Leftrightarrow \\ \cdot \quad \exists \text{ negozio, home } &\text{negozio} \in \text{NegoziOnline} \wedge \text{Homepage}(\text{negozio}, \text{home}) \\ \wedge \exists \text{ url}_2 \text{ CatenaRilevante}(&\text{home}, \textit{url}_2, \textit{query}) \wedge \text{Link}(\textit{url}_2, \textit{url}) \\ \wedge \textit{pagina} &= \text{GetPagina}(\textit{url}) . \end{aligned}$$

---

<sup>7</sup> Una strategia alternativa consiste nell'uso di un motore di ricerca; la tecnologia alla base dei motori di ricerca su Internet sarà trattata nel Paragrafo 23.2, nel 2° volume.

Il predicato  $\text{Link}(\text{partenza}, \text{arrivo})$  significa che c'è un collegamento ipertestuale dall'URL  $\text{partenza}$  all'URL  $\text{arrivo}$  (v. Esercizio 10.13). Per definire quello che conta in una *CatenaRilevante* non possiamo seguire qualsiasi collegamento, ma solo quelli il cui testo associato indica che il collegamento è rilevante per il prodotto della query. Per far questo useremo il  $\text{LinkText}(\text{partenza}, \text{arrivo}, \text{testo})$  per indicare che il link tra  $\text{partenza}$  e  $\text{arrivo}$  è contrassegnato dalla stringa  $\text{testo}$ . Una catena di collegamenti tra due URL, che chiameremo *inizio* e *fine*, è rilevante per una descrizione  $d$  se il testo associato a ogni collegamento è un nome di categoria rilevante per  $d$ . L'esistenza della catena stessa è determinata da una definizione ricorsiva, con una catena vuota ( $\text{inizio} = \text{fine}$ ) come caso base:

$$\begin{aligned} \text{CatenaRilevante}(\text{inizio}, \text{fine}, \text{query}) &\Leftrightarrow (\text{inizio} = \text{fine}) \\ &\vee (\exists u, \text{testo } \text{LinkText}(\text{inizio}, u, \text{testo}) \wedge \text{NomeCategoriaRilevante}(\text{query}, \text{testo}) \\ &\quad \wedge \text{CatenaRilevante}(u, \text{fine}, \text{query})). \end{aligned}$$

Ora occorre definire cosa significa dire che  $\text{testo}$  è un *NomeCategoriaRilevante* per  $\text{query}$ . Prima di tutto dobbiamo mettere in relazione le stringhe alle categorie di cui rappresentano i nomi. Per far questo useremo il predicato  $\text{Nome}(s, c)$ , che afferma che la stringa  $s$  è un nome della categoria  $c$ : potremmo ad esempio asserire  $\text{Nome}(\text{"laptop"}, \text{ComputerPortatili})$ . Altri esempi del predicato *Nome* compaiono nella Figura 10.8(b). Fatto questo, occorre definire la rilevanza. Supponiamo che la  $\text{query}$  sia "portatili". Allora  $\text{NomeCategoriaRilevante}(\text{query}, \text{testo})$  è vero quando vale una delle seguenti condizioni:

- ♦ il  $\text{testo}$  e il nome della  $\text{query}$  rappresentano lo stesso nome di categoria: ad esempio, "computer laptop" e "portatili";

$\text{Libri} \subset \text{Prodotti}$

$\text{IncisioniMusicali} \subset \text{Prodotti}$

$\text{CDMusicali} \subset \text{IncisioniMusicali}$

$\text{Audiocassette} \subset \text{IncisioniMusicali}$

$\text{Elettronica} \subset \text{Prodotti}$

$\text{FotocamereDigitali} \subset \text{Elettronica}$

$\text{Stereo} \subset \text{Elettronica}$

$\text{Computer} \subset \text{Elettronica}$

$\text{ComputerPortatili} \subset \text{Computer}$

$\text{ComputerDesktop} \subset \text{Computer}$

...

(a)

$\text{Nome}(\text{"libri"}, \text{Libri})$

$\text{Nome}(\text{"musica"}, \text{IncisioniMusicali})$

$\text{Nome}(\text{"CD"}, \text{CDMusicali})$

$\text{Nome}(\text{"nastri"}, \text{Audiocassette})$

$\text{Nome}(\text{"elettronica"}, \text{Elettronica})$

$\text{Nome}(\text{"fotocamere"}, \text{FotocamereDigitali})$

$\text{Nome}(\text{"stereo"}, \text{Stereo})$

$\text{Nome}(\text{"computer"}, \text{Computer})$

$\text{Nome}(\text{"laptop"}, \text{ComputerPortatili})$

$\text{Nome}(\text{"desktop"}, \text{DesktopComputers})$

...

(b)

Figura 10.8 (a) Una tassonomia di categorie di prodotti. (b) Le parole che si riferiscono a tali categorie.

- ◆ il *testo* rappresenta il nome di una supercategoria come “computer”;
- ◆ il *testo* rappresenta il nome di una sottocategoria come “portatili ultraleggeri”.

La definizione logica di *NomeCategoriaRilevante* è la seguente:

$$\begin{aligned} \text{NomeCategoriaRilevante}(\text{query}, \text{testo}) &\Leftrightarrow \\ \exists c_1, c_2 \quad \text{Nome}(\text{query}, c_1) \wedge \text{Nome}(\text{testo}, c_2) \wedge (c_1 \subseteq c_2 \vee c_2 \subseteq c_1). \end{aligned} \quad (10.1)$$

In caso contrario il testo associato al collegamento è irrilevante perché rappresenta il nome di una categoria esterna, come “mainframe” o “giardinaggio”.

Per riuscire a seguire i collegamenti rilevanti, quindi, è essenziale conoscere una complessa gerarchia di categorie di prodotti. Il livello superiore di questa gerarchia potrebbe somigliare alla Figura 10.8(a). Non si possono elencare *tutte* le possibili categorie dello shopping perché un acquirente potrebbe sempre escogitare un nuovo desiderio, e le aziende non si faranno pregare nell'escogitare nuovi prodotti per soddisfarlo (scalda-rotule elettrici?). Nonostante questo, un'ontologia che comprenda circa un migliaio di categorie potrà soddisfare le necessità della maggior parte degli acquirenti.

Oltre alla gerarchia di prodotti in sé, ci occorrerà anche un ricco vocabolario di nomi di categorie. La vita sarebbe molto più semplice se ci fosse una corrispondenza biunivoca tra le categorie e le stringhe di caratteri che le denominano. Abbiamo già accennato al problema della **sinonimia**, che si verifica quando la stessa categoria ha più nomi, come “computer laptop” e “portatili”. C’è anche il problema dell'**ambiguità**, quando lo stesso nome si riferisce a due o più categorie. Per esempio, se aggiungiamo la formula

$$\text{Nome}(\text{"CD"}, \text{CertificatiDiDeposito})$$

alla base di conoscenza della Figura 10.8(b), “CD” risulterà essere il nome di due diverse categorie.

La sinonimia e l’ambiguità possono essere causa di un significativo incremento del numero di cammini seguiti dall’agente, e talvolta può rendere difficile determinare se una data pagina è effettivamente rilevante. Un problema molto più grave è rappresentato dall’enorme varietà delle descrizioni che un utente può inserire o dei nomi di categoria utilizzati da un negozio. Ad esempio, un collegamento potrebbe essere contrassegnato dal testo “portatile” mentre la base di conoscenza conosce solo “portatili”; o l’utente potrebbe chiedere “un computer che possa stare sul ripiano ribaltabile della classe economica di un Boeing 737”. È impossibile elencare in anticipo tutti i modi in cui si può chiamare una categoria, ragione per cui l’agente dovrà essere in grado di svolgere ragionamenti aggiuntivi per determinare quando vale la relazione *Nome*. Nel caso pessimo questo richiederà un sistema completo di comprensione del linguaggio naturale, un argomento che rimandiamo al Capitolo 22 (nel 2° vol.). Nella pratica, per risolvere la maggior parte dei problemi sono sufficienti poche semplici regole, come quella di permettere a “portatile” di corrispondere a una categoria di nome “portatili”. L’Esercizio 10.15 vi chiederà di sviluppare un insieme di regole simili dopo aver svolto una ricerca sui negozi online.

collegamento  
procedurale

wrapper

A questo punto, date le definizioni logiche dei precedenti paragrafi e date adeguate basi di conoscenza di categorie di prodotti e di convenzioni sui nomi, siamo pronti ad applicare un algoritmo di inferenza per ottenere un insieme di offerte rilevanti per la nostra query? Non proprio! Manca ancora la funzione *GetPagina(url)*, che fa riferimento alla pagina HTML corrispondente a un dato URL. L'agente non conserva certo i contenuti di ogni URL nella sua base di conoscenza; né possiede regole esplicite per dedurli. Al posto della deduzione, possiamo organizzare un meccanismo che esegua la corretta procedura HTTP ogni volta che un sotto-obiettivo comprende la funzione *GetPagina*. In questo modo al motore inferenziale sembrerà sempre che la base di conoscenza contenga l'intero Web. Questo è un esempio di una tecnica generale chiamata **collegamento procedurale**, che permette di gestire particolari predicati e funzioni con metodi specializzati.

## Confrontare le offerte

Supponiamo che i processi di ragionamento descritti fin qui abbiano prodotto un insieme di pagine che rappresentano altrettante offerte per la nostra query “portatili”. Per confrontarle, l'agente dovrà estrarre informazione rilevante: prezzo, velocità, dimensioni del disco, peso e così via. Con delle vere pagine web questo può essere un compito arduo, per tutte le ragioni che abbiamo già menzionato. Per trattare questo problema si possono usare programmi chiamati **wrapper**, capaci di estrarre informazioni da una pagina. La tecnologia relativa sarà discussa nel Paragrafo 23.3, nel 2° volume: per ora possiamo semplicemente presumere che i wrapper esistano e che siano in grado, una volta ricevuta una pagina e una base di conoscenza, di aggiungere asserzioni alla base stessa. Tipicamente, a una pagina si applica un'intera gerarchia di wrapper: il più generale estrarrà date e prezzi, uno più specifico potrà estrarre attributi per prodotti informatici, e se necessario si potrà anche ricorrere a un wrapper altamente specializzato che conosce il formato di un particolare negozio online. Data una pagina del sito gen-store.com con il testo

YVM ThinkBook 970. Il nostro prezzo: \$1449.00

Seguito da varie specifiche tecniche, ci piacerebbe avere un wrapper per estrarre informazione come questa:

$$\begin{aligned}
 \exists lc, offerta \quad & lc \in Portatili \wedge offerta \in Offerte \wedge \\
 & DimensioniSchermo(lc, Pollici(14)) \wedge TipoSchermo(lc, LCD) \wedge \\
 & Memoria(lc, Megabyte(512)) \wedge CPU(lc, GHz(2.4)) \wedge \\
 & ProdottoOfferto(offerta, lc) \wedge Negozio(offerta, GenStore) \wedge \\
 & URL(offerta, "genstore.com/comps/34356.html") \wedge \\
 & Prezzo(offerta, $(1449)) \wedge Data(offerta, Oggi).
 \end{aligned}$$

Questo esempio illustra varie questioni che sorgono quando si intraprende seriamente l'attività del knowledge engineering applicato alle transazioni commerciali. Notate ad esempio che il prezzo è un attributo dell'*offerta*, e non del prodotto stes-

so. Questo punto è importante perché l'offerta di un dato negozio può cambiare da un giorno all'altro anche per lo stesso portatile; per alcune categorie – come case o dipinti – lo stesso singolo oggetto può essere addirittura offerto simultaneamente da diversi intermediari a prezzi differenti. Ci sono ancora altre complicazioni che non abbiamo considerato, come la possibilità che il prezzo dipenda dal metodo di pagamento o dal diritto dell'acquirente di ottenere certi sconti. Tutto sommato, c'è ancora un sacco di lavoro interessante da compiere.

L'ultimo passo è confrontare le offerte tra loro. Considerate ad esempio queste tre:

*A* : 2.4 GHz CPU, 512MB RAM, 80 GB disk, DVD, CDRW, \$1695 .

*B* : 2.0 GHz CPU, 1GB RAM, 120 GB disk, DVD, CDRW, \$1800 .

*C* : 2.2 GHz CPU, 512MB RAM, 80 GB disk, DVD, CDRW, \$1800 .

L'offerta *C* è dominata da *A*: infatti *A* è più veloce ed economico, mentre tutti gli altri parametri sono identici. In generale, *X* domina *Y* se gli è superiore in almeno un attributo e non gli è inferiore in alcun altro. Tra *A* e *B*, comunque, nessuno domina l'altro. Per decidere tra i due dobbiamo sapere quanto l'acquirente valuta la velocità del processore e il prezzo rispetto alla memoria e allo spazio su disco. L'argomento generale della preferenza tra attributi multipli è trattato nel Paragrafo 16.4 (nel 2° vol.); per adesso il nostro agente si limiterà a restituire una lista di tutte le offerte non dominate che soddisfano la descrizione dell'acquirente. In questo esempio, *A* e *B* non sono dominate. Notate che questo risultato si basa sull'assunto che tutti preferiscono prezzi bassi, processori veloci e dischi capienti. Alcuni attributi, come la dimensione dello schermo di un portatile, dipendono dalle preferenze personali dell'utente (portabilità vs. comodità): in questo caso l'agente non potrà far altro che chiedere a lui.

L'agente per lo shopping che abbiamo descritto è molto semplice; è possibile introdurre molti raffinamenti. Nonostante questo, se gli viene fornita la giusta conoscenza specifica del dominio, è abbastanza potente da essere effettivamente utile. Grazie alla sua costruzione dichiarativa, è molto facile estenderlo per gestire applicazioni più complesse. Lo scopo principale di questo paragrafo è mostrare che un agente come questo necessita di una certa quantità di rappresentazione della conoscenza, in questo caso costituita dalla gerarchia di prodotti, e che una volta ricevuta conoscenza in tale forma non è troppo difficile sviluppare tutto il resto.

## 10.6 Sistemi di ragionamento per categorie

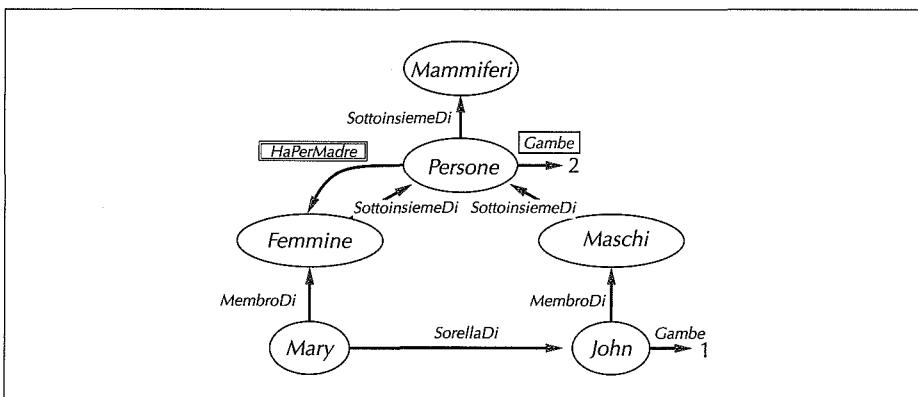
Come abbiamo visto, le categorie sono gli elementi principali per la costruzione di schemi di rappresentazione su grande scala. Questo paragrafo descrive i sistemi esplicitamente progettati per organizzare e ragionare sulle categorie. Ci sono due famiglie di sistemi, strettamente imparentate: le reti semantiche permettono di visualizzare graficamente una base di conoscenza e forniscono algoritmi efficienti per inferire le proprietà di un oggetto sulla base della sua appartenenza a una categoria; le logiche descrittive rappresentano un linguaggio formale per costruire e combinare definizioni di categorie e forniscono algoritmi efficienti per determinare le relazioni di sottoinsieme e superinsieme tra categorie.

### Reti semantiche

grafi esistenziali

Nel 1909, Charles Peirce propose una notazione grafica basata su nodi e archi che chiamò grafi esistenziali e definì “la logica del futuro”. Cominciò così un lungo dibattito tra i paladini della “logica” e quelli delle “reti semantiche”. Sfortunatamente, le diatribe nascosero il fatto che le reti, almeno quelle la cui semantica è ben definita, sono una forma di logica. Per certi tipi di formule la notazione offerta dalle reti semantiche è spesso più comoda, ma se non consideriamo le questioni di “interfacciamento” con gli esseri umani, i concetti sottostanti – oggetti, relazioni, quantificazioni etc. – sono identici.

Ci sono molte varianti di reti semantiche, ma tutte sono capaci di rappresentare oggetti singoli, categorie di oggetti e relazioni. Una notazione grafica tipica rappresenta i nomi degli oggetti e delle categorie racchiusi in ovali o rettangoli, collegati con archi etichettati. Ad esempio, la Figura 10.9 ha un collegamento *MembroDi* tra *Mary* e *Femmine*, che corrisponde all’asserzione logica *Mary ∈ Femmine*;



**Figura 10.9** Una rete semantica con quattro oggetti (*John*, *Mary*, 1, 2) e quattro categorie. Le relazioni sono rappresentate da archi etichettati.

analogalemente, il collegamento *SorellaDi* tra *Mary* e *John* corrisponde all'asserzione *SorellaDi(Mary, John)*. Possiamo mettere in relazione categorie mediante collegamenti *SottoinsiemeDi*, e così via. Disegnare bolle e frecce è così divertente che ci si può anche lasciar prendere la mano. Per esempio, sappiamo che tutte le persone hanno come madre una femmina; è possibile quindi tracciare un collegamento *HaPerMadre* da *Persone* a *Femmine*? La risposta è no, perché *HaPerMadre* è una relazione tra una persona e sua madre, mentre le categorie non hanno madri.<sup>8</sup> Per questa ragione, nella Figura 10.9, abbiamo usato come notazione speciale un collegamento la cui etichetta è racchiusa da un doppio rettangolo. Quel collegamento asserisce che

$$\forall x \ x \in \text{Persone} \Rightarrow [\forall y \underset{\text{membro di}}{\text{HaPerMadre}}(x, y) \Rightarrow y \in \text{Femmine}] .$$

Potremmo anche voler asserire che le persone hanno due gambe, ovvero:

$$\forall x \ x \in \text{Persone} \Rightarrow \text{Gambe}(x, 2) .$$

Anche in questo caso dobbiamo stare attenti a non dire che un categoria ha le gambe; il collegamento nella Figura 10.9 ha un'etichetta racchiusa in un rettangolo singolo ed è usato per asserire le proprietà comuni a ogni membro di una categoria.

La notazione delle reti semantiche rende molto semplice ragionare su un'ereditarietà del tipo che abbiamo introdotto nel Paragrafo 10.2. In virtù del fatto di essere una persona, ad esempio, Mary eredita la proprietà di avere due gambe. Così, per determinare il numero di gambe di Mary, l'algoritmo di ereditarietà seguirà il collegamento *MembroDi* da *Mary* alla categoria a cui ella appartiene e da lì seguirà la catena di collegamenti *SottoinsiemeDi* finché non troverà una categoria che ha un collegamento etichettato dalla scritta *Gambe* racchiusa in un rettangolo: in questo caso, *Persone*. La semplicità e l'efficienza di questo meccanismo di inferenza rispetto alla dimostrazione logica di teoremi è stata una delle attrattive principali delle reti semantiche.

L'ereditarietà diventa complicata quando un oggetto può appartenere a più di una categoria, o quando una categoria può essere un sottoinsieme di più di un'altra categoria; in questo caso si parla di ereditarietà multipla. Ora l'algoritmo potrebbe trovare, in risposta a una query, due o più valori in conflitto. Per questa ragione alcuni linguaggi di programmazione orientata agli oggetti (OOP), come Java, proibiscono l'uso dell'ereditarietà multipla nella gerarchia delle classi. Nelle reti semantiche, comunque, essa è generalmente consentita: ne ripareremo nel Paragrafo 10.7.

<sup>8</sup> Molti dei primi sistemi non riuscivano a distinguere tra le proprietà dei membri di una categoria e quelle della categoria stessa. Questo può portare a delle gravi inconsistenze, come ha fatto notare Drew McDermott (1976) nel suo articolo "L'Intelligenza Artificiale incontra la Naturale Stupidità". Un altro problema comune era l'uso di collegamenti *ÈUn* per indicare sia la relazione di sottoinsieme che quella di appartenenza a una categoria, come nel linguaggio naturale: "un gatto è un mammifero" e "Cirillo è un gatto". L'Esercizio 10.25 esplora ulteriormente quest'argomento.

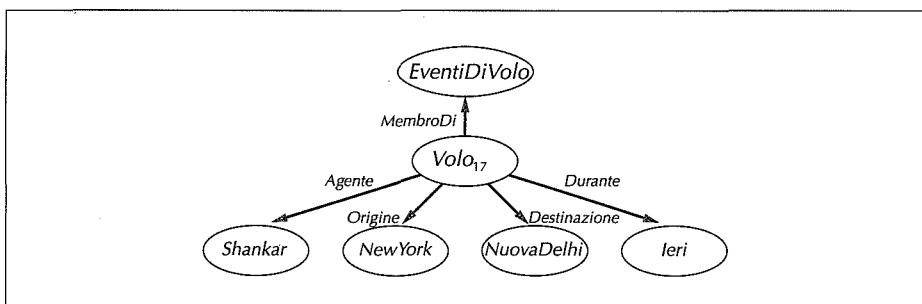
collegamenti inversi

Un'altra forma diffusa di inferenza è l'uso dei collegamenti inversi. Ad esempio, *HaPerSorella* è l'inverso di *SorellaDi*, il che significa che

$$\forall p, s \text{ } HaPerSorella(p, s) \Leftrightarrow SorellaDi(s, p).$$

Questa formula può essere espressa in una rete semantica se i collegamenti sono reificati, cioè trasformati in oggetti a sé stanti. Potremmo quindi avere un oggetto *SorellaDi* con un collegamento Inverso che punta verso *HaPerSorella*. Data un'interrogazione che chiede chi è la *SorellaDi* John, l'algoritmo di inferenza può scoprire che *HaPerSorella* è l'inverso di *SorellaDi* e rispondere alla query seguendo il collegamento *HaPerSorella* da *John* a *Mary*. Senza l'informazione inversa sarebbe stato necessario controllare ogni persona di sesso femminile per verificare l'eventuale presenza di un collegamento *SorellaDi* verso John. Questo è dovuto al fatto che le reti semantiche forniscono un'indicizzazione diretta solo per gli oggetti, le categorie e i collegamenti uscenti da essi; in termini di logica del primo ordine, è come avere una base di conoscenza in cui è indicizzato solo il primo argomento di ogni predicato.

Il lettore avrà già notato un'ovvia limitazione delle reti semantiche rispetto alla logica del primo ordine, e cioè il fatto che i collegamenti tra le bolle possono rappresentare solo relazioni binarie. La formula *Vola(Shankar, NewYork, NuovaDelhi, Ieri)*, ad esempio, in una rete semantica non può essere espressa direttamente. Questo non toglie che sia possibile ottenere l'effetto di un'asserzione n-aria; per far questo si deve reificare la proposizione stessa come un evento (v. Paragrafo 10.3) appartenente a una categoria di eventi appropriata. La Figura 10.10 mostra la struttura di una rete semantica per questo particolare evento. Notate che la restrizione delle relazioni binarie ha come conseguenza la creazione di una ricca ontologia di concetti reificati; in effetti, gran parte dell'ontologia che abbiamo sviluppato in questo capitolo ha avuto origine nel campo dei sistemi di reti semantiche.



**Figura 10.10** Un frammento di rete semantica che mostra la rappresentazione dell'asserzione logica *Vola(Shankar, NewYork, NuovaDelhi, Ieri)*.

La reificazione delle proposizioni rende possibile rappresentare ogni formula atomica, ground e priva di funzioni della logica del primo ordine con la notazione delle reti. Alcuni tipi di formule universalmente quantificate possono essere espresse usando collegamenti inversi e con frecce etichettate con rettangoli singoli e doppi applicati alle categorie; questo tuttavia ci lascia ben lontani dalla potenza espressiva della logica del primo ordine completa. Tra le altre cose non possiamo esprimere la negazione, la disgiunzione, funzioni annidate e la quantificazione esistenziale. È possibile estendere la notazione per renderla del tutto equivalente alla logica del primo ordine, come hanno fatto gli stessi grafi esistenziali di Peirce o le reti semantiche partizionate di Hendrix(1975), questo però elimina uno dei vantaggi principali delle reti semantiche: la semplicità e la trasparenza dei processi di inferenza. I progettisti possono costruire una grande rete e continuare ad avere una buona idea di quali interrogazioni risulteranno efficienti, perché (a) è facile visualizzare i passi della procedura di inferenza e (b) in alcuni casi il linguaggio di query è così semplice che non si possono proprio esprimere interrogazioni complesse. Nei casi in cui la potenza espressiva è troppo limitata, per colmare le lacune molte reti semantiche sfruttano il cosiddetto meccanismo del collegamento procedurale (*procedural attachment*): questa tecnica fa sì che una query (e talvolta anche un'asserzione) che riguarda una certa relazione abbia come risultato la chiamata di una procedura speciale appositamente progettata, senza utilizzare l'algoritmo generale di inferenza.

Uno degli aspetti più importanti delle reti semantiche è la capacità di rappresentare valori di default per le categorie. Esaminando attentamente la Figura 10.9, potete notare che John ha una sola gamba, nonostante il fatto che sia una persona e che tutte le persone abbiano due gambe. In una KB strettamente logica, questa sarebbe una contraddizione; in una rete semantica, tuttavia, l'asserzione che tutte le persone hanno due gambe ha solo un valore di default. Questo significa che si presume sempre che le persone abbiano due gambe a meno che ciò non sia contraddetto da informazione più specifica. La semantica dei valori di default è supportata in modo naturale dall'algoritmo di ereditarietà, perché quest'ultimo segue i link verso l'alto partendo dall'oggetto (John, in questo caso) e fermandosi non appena trova un valore. Si dice che il valore di default è sovrascritto (*overridden*) dal valore più specifico. Notate che potremmo anche sovrascrivere il numero di gambe di default creando una categoria di *PersoneConUnaGamba*, un sottoinsieme di *Persone* di cui John sarebbe membro.

Per mantenere la stretta correttezza logica della rete, possiamo dire che l'asserzione Gambe delle Persone include un'eccezione nel caso di John:

$$\forall x . \ x \in \text{Persone} \wedge x \neq \text{John} \Rightarrow \text{Gambe}(x, 2) .$$

Per una rete di struttura prefissata questa semantica è accettabile, ma nel caso ci siano molte eccezioni le espressioni logiche saranno molto meno concise della notazione grafica. Nel caso che la rete debba essere aggiornata con nuove asservizioni,

valori di default

sovrascrittura

poi, quest'approccio fallirà completamente: dovremmo dire che anche tutte le persone (presentemente sconosciute) con una sola gamba costituiscono un'eccezione alla regola. Il Paragrafo 10.7 approfondisce questo problema e il ragionamento di default in generale.

## Logiche descrittive

La sintassi della logica del primo ordine è progettata per rendere facile parlare degli oggetti. Le logiche descrittive sono notazioni progettate per facilitare la descrizione della definizione e delle proprietà delle categorie. I sistemi basati sulla logica descrittiva si sono evoluti dalle reti semantiche per rispondere alla necessità di formalizzare il significato delle reti, mantenendo l'enfasi sulla struttura tassonomica come principio organizzativo.

Il principale compito inferenziale delle logiche descrittive è la sussunzione, che consiste nel determinare se una categoria è il sottoinsieme di un'altra confrontando le loro definizioni, e la classificazione, ovvero la verifica che un oggetto appartiene a una categoria. Alcuni sistemi includono anche la verifica di consistenza di una definizione di categoria: una definizione è consistente se i criteri di appartenenza sono logicamente soddisfacibili.

Il linguaggio CLASSIC (Borgida et al., 1989) è una tipica logica descrittiva. La sintassi delle descrizioni CLASSIC è mostrata nella Figura 10.11.<sup>9</sup> Per esempio, per dire che gli scapoli sono maschi adulti non sposati scriveremo

*Scapolo = And (NonSposato, Adulto, Maschio).*

In logica del primo ordine, la formula equivalente sarebbe

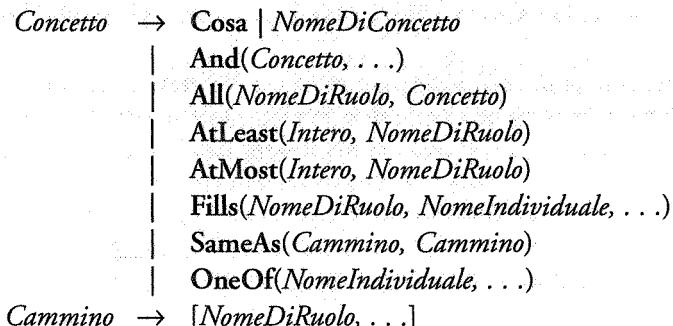
*Scapolo (x)  $\Leftrightarrow$  NonSposato(x)  $\wedge$  Adulto(x)  $\wedge$  Maschio(x).*

Notate che la logica descrittiva permette di applicare direttamente operazioni logiche ai predicati: non è più necessario scrivere le formule e unirle con i connettivi. Ogni descrizione in CLASSIC può essere scritta in logica del primo ordine, ma in alcuni casi CLASSIC è più semplice. Ad esempio, per descrivere l'insieme degli uomini con almeno tre figli tutti disoccupati e sposati con dottori e con al più due figlie tutte professoresse di fisica o matematica, potremmo scrivere

*And(Uomo, AtLeast(3, Figlio), AtMost(2, Figlia),  
All(Figlio, And(Disoccupato, Sposato, All(Coniuge, Dottore))),  
All(Figlia, And(Professore, Fills(Dipartimento, Fisica, Matematica)))) .*

Lasciamo ai lettori l'esercizio di tradurre tutto ciò in logica del primo ordine.

<sup>9</sup> Notate che il linguaggio *non* permette di affermare semplicemente che un concetto, o una categoria, è il sottoinsieme di un altro. Questa è una scelta voluta: la sussunzione tra categorie dev'essere derivabile da aspetti delle loro stesse descrizioni. Se questo non è il caso, è segno che le descrizioni sono in qualche modo carenti.



**Figura 10.11** La sintassi delle descrizioni in un sottoinsieme del linguaggio CLASSIC.

L'aspetto più importante delle logiche descrittive è forse la loro enfasi sulla trattabilità dell'inferenza. Un'istanza di problema è risolta descrivendola e poi chiedendo al sistema se è sussunta da una tra molte possibili categorie di soluzioni. Nei sistemi standard basati sulla logica del primo ordine, predire il tempo richiesto per il calcolo di una soluzione è quasi sempre impossibile. Quando un problema richiede settimane di tempo di calcolo, è spesso compito dell'utente organizzare la rappresentazione in modo da "girare intorno" agli insiemi di formule che sembrano problematiche. Uno degli scopi principali della logica descrittiva, al contrario, è assicurare che la verifica di sussunzione possa essere effettuata in un tempo che cresce con un polinomio della dimensione delle descrizioni.<sup>10</sup>

In via di principio questo sembra fantastico, finché non si realizza che ci possono essere due sole possibili conseguenze: i problemi difficili non potranno proprio essere espressi, o le loro descrizioni dovranno essere esponenzialmente grandi! I risultati, comunque, permettono di riconoscere i costrutti che causano problemi e aiutano quindi l'utente a capire come si comportano le diverse rappresentazioni.

Le logiche descrittive normalmente non sono dotate di negazione e disgiunzione, che obbligano i sistemi del primo ordine a eseguire un'analisi potenzialmente esponenziale per assicurare la completezza. Questa è anche la ragione, del resto, per cui sono state escluse dal Prolog. CLASSIC prevede una forma limitata di disgiunzione nei costrutti Fills e OneOf, che permettono di esprimere disgiunzioni su elementi esplicitamente enumerati ma non sulle descrizioni. Se le descrizioni fossero disgiuntive, definizioni nidificate potrebbero facilmente portare a un numero esponenziale di cammini alternativi per cui una categoria può sussumerne un'altra.

---

<sup>10</sup> CLASSIC offre un controllo della sussunzione efficiente in pratica, ma il tempo di esecuzione nel caso pessimo è esponenziale.

## 10.7 Ragionare con informazione di default

Nel paragrafo precedente abbiamo incontrato un semplice esempio di asserzione con uno stato di default: le persone hanno due gambe. Questo default può essere sovrascritto da informazione più specifica, come il fatto che Long John Silver ha una gamba sola. Come abbiamo visto, il meccanismo dell'ereditarietà nelle reti semantiche implementa la sovrascrittura dei valori di default in modo semplice e naturale. In questo paragrafo studieremo i default in modo più generale, cercando di capire la loro semantica e non solo di fornire meccanismi procedurali.

### Mondi aperti e mondi chiusi

Supponiamo di guardare la bacheca del dipartimento di Informatica e notare un avviso che dice “I codici dei corsi attivati sono: CS 101, CS 102, CS 106, EE 101”. Secondo voi quanti corsi sono stati attivati? Se avete risposto “quattro”, sareste d'accordo con un tipico sistema di basi di dati. Infatti, in un database relazionale che contiene l'equivalente delle quattro asserzioni

$$\text{Corso}(CS, 101), \text{Corso}(CS, 102), \text{Corso}(CS, 106), \text{Corso}(EE, 101), \quad (10.2)$$

L'interrogazione SQL `count * from Corso` restituisce 4. D'altra parte, un sistema di logica del primo ordine non risponderebbe quattro, ma “un valore compresso tra uno e infinito”. Il fatto è che le asserzioni *Corso* non precludono la possibilità che siano offerti altri corsi non menzionati, né specificano che i quattro corsi riportati sono distinti tra loro.

Questo esempio mostra che le basi di dati e le convenzioni umane differiscono dalla logica del primo ordine in almeno due punti. Prima di tutto, i database (e le persone) presumono sempre che l'informazione fornita sia completa, cosicché le formule atomiche ground di cui non si afferma la verità sono considerate false. Questa prende il nome di ipotesi del mondo chiuso, o CWA dall'acronimo inglese. In secondo luogo, noi diamo sempre per scontato che nomi distinti si riferiscono a oggetti distinti. Questa è l'ipotesi dei nomi unici, o UNA, già introdotta nel Paragrafo 10.3 nel contesto dei nomi di azione.

La logica del primo ordine non fa proprie queste convenzioni, e dev'essere quindi più esplicita. Per dire che sono offerti solo quattro corsi distinti, dovremo scrivere:

$$\begin{aligned} \text{Corso}(d, n) \Leftrightarrow & [d, n] = [CS, 101] \vee [d, n] = [CS, 102] \\ & \vee [d, n] = [CS, 106] \vee [d, n] = [EE, 101] . \end{aligned} \quad (10.3)$$

L'Equazione 10.3 si chiama **completamento**<sup>11</sup> di 10.2. In generale, il completamento conterrà una definizione (cioè una formula del tipo *se e solo se*) per ogni predicato, e ogni definizione conterrà un disgiunto per ogni clausola definita che ha quel predicato nella testa.<sup>12</sup> In generale, il completamento è costruito come segue:

1. si raccolgono tutte le clausole con lo stesso nome di predicato ( $P$ ) e la stessa aritria ( $n$ );
2. si traduce ogni clausola in **Forma Normale di Clark** rimpiazzando

$$P(t_1, \dots, t_n) \leftarrow \text{Corpo},$$

completamento

Forma Normale di Clark

dove i  $t_i$  sono termini, con

$$P(v_1, \dots, v_n) \leftarrow \exists w_1 \dots w_m [v_1, \dots, v_n] = [t_1, \dots, t_n] \wedge \text{Corpo},$$

dove le  $v_i$  sono nuove variabili inventate appositamente e le  $w_i$  quelle che appaiono nella clausola originale. In ogni clausola va usato lo stesso insieme di  $v_i$ . Questo ci dà un insieme di clausole

$$P(v_1, \dots, v_n) \leftarrow B_1$$

 $\vdots$ 

$$P(v_1, \dots, v_n) \leftarrow B_k$$

3. ora le clausole sono combinate tutte insieme in una grande clausola disgiuntiva:

$$P(v_1, \dots, v_n) \leftarrow B_1 \vee \dots \vee B_k$$

4. infine, per formare il completamento, si rimpiazza il  $\leftarrow$  con un'equivalenza:

$$P(v_1, \dots, v_n) \Leftrightarrow B_1 \vee \dots \vee B_k.$$

La Figura 10.12 mostra un esempio di completamento di Clark per una base di conoscenza che comprende sia fatti ground che regole. Per aggiungere l'ipotesi dei nomi unici sarà sufficiente costruire il completamento di Clark per la relazione di uguaglianza, in cui i soli fatti noti saranno  $CS = CS$ ,  $101 = 101$  e così via. Questo è lasciato come esercizio.

<sup>11</sup> Chiamato anche "completamento di Clark" in onore dell'inventore, Keith Clark.

<sup>12</sup> Notate che questa è la stessa forma degli assiomi di stato successore che abbiamo visto nel Paragrafo 10.3.

**Clausole di Horn**

$$\begin{aligned} & \text{Corso}(CS, 101) \\ & \text{Corso}(CS, 102) \\ & \text{Corso}(CS, 106) \\ & \text{Corso}(EE, 101) \\ & \text{Corso}(EE, i) \leftarrow \text{Intero}(i) \\ & \quad \wedge 101 \leq i \wedge i \leq 130 \\ & \text{Corso}(CS, m + 100) \Leftarrow \\ & \quad \text{Corso}(CS, m) \wedge 100 \leq m \\ & \quad \wedge m < 200 \end{aligned}$$
**Completamento di Clark**

$$\begin{aligned} & \text{Corso}(d, n) \Leftrightarrow [d, n] = [CS, 101] \\ & \quad \vee [d, n] = [CS, 102] \\ & \quad \vee [d, n] = [CS, 106] \\ & \quad \vee [d, n] = [EE, 101] \\ & \quad \vee \exists i [d, n] = [EE, i] \wedge \text{Intero}(i) \\ & \quad \wedge 101 \leq i \wedge i \leq 130 \\ & \quad \vee \exists m [d, n] = [CS, m + 100] \\ & \quad \quad \wedge \text{Corso}(CS, m) \wedge 100 \leq m \\ & \quad \quad \wedge m < 200 \end{aligned}$$

**Figura 10.12** Il completamento di Clark di un insieme di clausole di Horn. Il programma di Horn originale (a sinistra) elenca esplicitamente quattro corsi e asserisce che c'è un corso di matematica (codice *EE*) per ogni intero che va da 101 a 130, e che per ogni corso di informatica (codice *CS*) nella serie 100 (studenti non laureati) c'è un corrispondente corso nella serie 200 (dottorandi e studenti di master). Il completamento di Clark (a destra) afferma che non esistono altri corsi. Con il completamento e l'ipotesi dei nomi unici (e l'ovvia definizione del predicato *Intero*) arriviamo alla conclusione desiderata, e cioè che ci sono esattamente 36 corsi: 30 di matematica e 6 di informatica.

L'ipotesi del mondo chiuso ci permette di trovare il **modello minimo** di una relazione, cioè quello con il numero minimo di elementi. Nell'Equazione (10.2) il modello minimo di *Corso* ha quattro elementi; con un numero inferiore avremo una contraddizione. Nelle basi di conoscenza di Horn, il modello minimo è sempre **unico**. Notate che, con l'ipotesi dei nomi unici, questo si applica anche alla relazione di uguaglianza: ogni termine è uguale solo a se stesso. Paradossalmente questo significa che i modelli minimi sono anche massimi, nel senso che hanno il maggior numero di oggetti possibile.

È possibile prendere un programma di Horn, generare il completamento di Clark e passare il risultato a un dimostratore di teoremi per svolgere le inferenze; spesso però è più efficiente utilizzare un meccanismo di inferenza specializzato come il Prolog, che include nativamente le ipotesi del mondo chiuso e dei nomi unici.

Quando si fa propria l'ipotesi del mondo chiuso occorre fare attenzione al tipo di ragionamento svolto. Ad esempio, in un database usato per un censimento sarebbe opportuno adottare la CWA quando si ragiona sulla popolazione corrente delle città, ma sarebbe scorretto concludere che non nascerà mai alcun bambino solo perché il database non contiene date di nascita nel futuro. La CWA rende il database **completo**, nel senso che ogni interrogazione atomica avrà una risposta positiva o negativa; qualora fossimo effettivamente ignoranti dei fatti (come le nascite future) non potremmo adottare l'ipotesi del mondo chiuso. Un sistema di rappresentazione della conoscenza più sofisticato potrebbe permettere all'utente di formulare regole per specificare quando applicare la CWA.

## La negazione come fallimento e la semantica del modello stabile

Come abbiamo visto nei Capitoli 7 e 9, le basi di conoscenza in forma di Horn possiedono diverse caratteristiche computazionali desiderabili. In molte applicazioni, tuttavia, il requisito che ogni letterale nel corpo di una clausola sia positivo è piuttosto scomodo: vorremmo poter dire "puoi uscire se non piove" senza ricorrere a prediciati come *NonPiove*. In questo paragrafo esamineremo la possibilità di aggiungere una forma di negazione esplicita alle clausole di Horn per mezzo della negazione come fallimento. L'idea è che un letterale negativo, *not P*, può essere "dimostrato" vero se la dimostrazione di *P* fallisce. Questa è una forma di ragionamento di default molto vicina all'ipotesi del mondo chiuso: presumiamo che qualcosa sia falso se non possiamo dimostrarne che è vero. Useremo "*not*" per distinguere la negazione come fallimento dall'operatore logico " $\neg$ ".

Prolog ammette l'uso dell'operatore *not* nel corpo di una clausola. Ad esempio, considerate il seguente programma:

```
IDEdrive ← Drive ∧ not SCSIdrive .
SCSIdrive ← Drive ∧ not IDEdrive .
SCSIcontroller ← SCSIdrive .
Drive .
```

(10.4)

La prima regola afferma che se abbiamo un disco rigido nel computer e il disco non è SCSI, allora dev'essere IDE. La seconda dice che se non è IDE allora dev'essere per forza SCSI. La terza specifica che se c'è un disco SCSI dev'esserci per forza anche il suo controller, e la quarta asserisce il fatto che abbiamo effettivamente un disco. Questo programma ha *due* modelli minimi:

$$\begin{aligned} M_1 &= \{Drive, IDEdrive\} , \\ M_2 &= \{Drive, SCSIdrive, SCSIcontroller\} . \end{aligned}$$

I modelli minimi non catturano la semantica intesa dei programmi che applicano la negazione come fallimento. Considerate il programma

$$P \leftarrow \text{not } Q .$$
(10.5)

Ci sono due modelli minimi,  $\{P\}$  e  $\{Q\}$ . Dal punto di vista della logica del primo ordine questo ha senso, perché  $P \Leftarrow \neg Q$  è equivalente a  $P \vee Q$ . Ma in Prolog c'è qualche problema: infatti *Q* non compare mai a sinistra della freccia, per cui come può essere una conseguenza?

Un'idea alternativa è costituita dai modelli stabili: un modello stabile è un modello minimo in cui ogni atomo ha una giustificazione, ovvero una regola di cui l'atomo è la testa e ogni letterale del corpo è soddisfatto. Tecnicamente si dice che *M* è un modello stabile di un programma *H* se *M* è il modello minimo unico del ridotto di *H* rispetto a *M*. Il ridotto di un programma *H* si ottiene cancellando per prima cosa da *H* ogni regola che ha nel corpo un letterale *not A*, dove *A* è nel

negazione come  
fallimento

modelli stabili  
giustificazione

ridotto

modello, e quindi cancellando ogni letterale negativo nelle regole rimanenti. Dato che il ridotto di  $H$  è una lista di clausole di Horn, deve avere un modello minimo unico.

Il ridotto di  $P \leftarrow \text{not } Q$  rispetto a  $\{P\}$  è  $P$ , che ha come modello minimo  $\{P\}$ . Ne consegue che  $\{P\}$  è un modello stabile. Il ridotto rispetto a  $\{Q\}$  è il programma vuoto, che ha come modello minimo  $\{\}$ . Quindi  $\{Q\}$  non è un modello stabile, perché  $Q$  non ha alcuna giustificazione nell'Equazione (10.5). Un altro esempio è il ridotto di 10.4 rispetto a  $M_1$ :

```
IDEdrive ← Drive .
SCSIcontroller ← SCSIdrive .
Drive .
```

programmazione  
con insiemi di risposta

insiemi di risposta

Questo ha il modello minimo  $M_1$ , che è quindi un modello stabile. La **programmazione con insiemi di risposta** (*answer set programming*) è un tipo di programmazione logica che adotta la negazione come fallimento e funziona traducendo un programma logico in forma ground e poi cercando modelli stabili (denominati **insiemi di risposta**) mediante tecniche di model checking proposizionale. La programmazione con insiemi di risposta deriva quindi sia dal Prolog che dai dimostratori di soddisfacibilità proposizionali più veloci, come WALKSAT; in effetti è stata applicata con successo agli stessi problemi di pianificazione affrontati da questi ultimi. Il vantaggio della pianificazione basata su insiemi di risposta è il suo grado di flessibilità: gli stessi operatori e i vincoli possono essere espressi come programmi logici e non sono limitati al formato ristretto di un particolare formalismo. Gli svantaggi sono quelli tipici delle tecniche proposizionali: se il numero degli oggetti nell'universo diventa molto alto, si può verificare un rallentamento esponenziale.

## Circoscrizione e logica di default

Abbiamo visto due esempi in cui processi di ragionamento apparentemente naturali violavano la proprietà della **monotonicità** della logica che abbiamo dimostrato nel Capitolo 7.<sup>13</sup> Nel primo esempio, una proprietà ereditata da tutti i membri di una categoria in una rete semantica poteva essere sovrascritta da informazione più specifica relativa a una sottocategoria. Nel secondo esempio, letterali negati derivati dall'ipotesi del mondo chiuso potevano essere sovrascritti dall'aggiunta di letterali positivi.

Una semplice introspezione ci suggerisce che queste eccezioni alla monotonicità sono estremamente frequenti nel ragionamento comune: sembra che le persone tendano a “saltare alle conclusioni”. Guardando un macchina parcheggiata, ad

---

<sup>13</sup> Ricordiamo che la monotonicità richiede che tutte le formule implicate rimangano tali dopo l'aggiunta di nuove formule alla KB. In altre parole, se  $KB \models \alpha$  allora  $KB \wedge \beta \models \alpha$ .

esempio, tenderemo a credere che abbia quattro ruote anche se ne vediamo solo tre (e se siete pronti a dubitare dell'esistenza della quarta ruota, considerate anche la possibilità che le tre visibili siano delle semplici sagome di cartone). La teoria della probabilità può aiutarci a concludere che la quarta ruota esiste quasi certamente, ma non è questo il punto: per la maggior parte delle persone, la possibilità che la macchina non abbia quattro ruote *non si pone a meno che non si presenti nuova informazione in suo supporto*. Sembra quindi che la conclusione sul numero delle ruote sia raggiunta *per default*, a meno che non ci siano ragioni per dubitarne. Qualora dovessero presentarsi altre informazioni (se dovessimo ad esempio vedere che il proprietario sta trasportando una ruota, e la macchina è stata sollevata con il cric) la conclusione potrebbe essere ritrattata. Si dice che questo tipo di ragionamento è **non monotono**, perché l'insieme delle credenze non cresce monotonicamente nel tempo man mano che giungono nuove informazioni. Per catturare questo tipo di comportamento sono state inventate **logiche non monotone**, che prevedono definizioni diverse di verità e implicazione. Esamineremo due logiche che sono state studiate estensivamente: la circoscrizione e la logica di default.

La **circoscrizione** può essere vista come una versione più precisa e potente dell'ipotesi del mondo chiuso. L'idea è specificare particolari predicati che si presu-mono "più falsi possibile": in altre parole, saranno falsi per tutti gli oggetti tranne quelli per cui è noto che sono veri. Supponiamo ad esempio di voler esprimere la regola di default secondo cui gli uccelli volano. Dovremo introdurre un predicato, diciamo *Abnormal<sub>1</sub>(x)*, e scrivere

$$\text{Uccello}(x) \wedge \neg \text{Abnormal}_1(x) \Rightarrow \text{Vola}(x).$$

Se diciamo che *Abnormal<sub>1</sub>* dev'essere **circoscritto**, un ragionatore circoscrittivo potrà dare per scontato  $\neg \text{Abnormal}_1(x)$  a meno che la verità di *Abnormal<sub>1</sub>(x)* sia nota. Questo permette di concludere *Vola(Titti)* dalla premessa *Uccello(Titti)*, ma la conclusione non varrà più qualora si appurasse che *Abnormal<sub>1</sub>(Titti)*.

La circoscrizione può essere considerata un esempio di logica con **preferenza di modelli**. In tali logiche una formula è implicata (per default) se è vera in tutti i modelli *preferiti* della KB, mentre nella logica classica il requisito è che sia vera in *tutti* i modelli possibili. Nella circoscrizione un modello è preferito a un altro se ha meno oggetti anormali.<sup>14</sup> Vediamo come funziona quest'idea nel contesto dell'ereditarietà multipla nelle reti semantiche. L'esempio classico per illustrare i problemi

non monotono

logiche non monotone

circoscrizione

preferenza di modelli

<sup>14</sup> Per l'ipotesi del mondo chiuso, un modello è preferito a un altro se ha meno atomi veri: in altre parole, i modelli preferiti sono quelli **minimi**. C'è un'affinità naturale tra la CWA e le KB composte da clausole definite, perché il punto fisso raggiunto dalla concatenazione in avanti su tali KB è il modello minimo unico (v. pag. 282).

di ereditarietà multipla è chiamato il “rombo di Nixon”. Richard Nixon era infatti sia un Quacchero (e quindi per default un pacifista) che un Repubblicano (per default, non pacifista). Possiamo scriverlo come segue:

$$\text{Repubblicano}(\text{Nixon}) \wedge \text{Quacchero}(\text{Nixon}) .$$

$$\text{Repubblicano}(x) \wedge \neg \text{Abnormal}_2(x) \Rightarrow \neg \text{Pacifista}(x) .$$

$$\text{Quacchero}(x) \wedge \neg \text{Abnormal}_3(x) \Rightarrow \text{Pacifista}(x) .$$

Se circoscriviamo  $\text{Abnormal}_2$  e  $\text{Abnormal}_3$ , ci sono due modelli preferiti: nel primo valgono  $\text{Abnormal}_2(\text{Nixon})$  e  $\text{Pacifista}(\text{Nixon})$ , nell’altro  $\text{Abnormal}_3(\text{Nixon})$  e  $\neg \text{Pacifista}(\text{Nixon})$ . Il ragionatore circoscrittivo, quindi, rimarrà correttamente agnostico riguardo alla natura pacifista di Nixon. Se, oltre a tutto ciò, vogliamo assicurare che le credenze religiose hanno la precedenza su quelle politiche, possiamo usare un formalismo chiamato **circoscrizione con priorità** che permette di dare la preferenza ai modelli in cui  $\text{Abnormal}_3$  è minimizzato.

**La logica di default** è un formalismo in cui si possono scrivere **regole di default** per generare conclusioni contingenti non monotone. Una regola di default ha quest’aspetto:

$$\text{Uccello}(x) : \text{Vola}(x) / \text{Vola}(x) .$$

Questa regola significa che se  $\text{Uccello}(x)$  è vero, e  $\text{Vola}(x)$  è consistente con la base di conoscenza, allora si può concludere  $\text{Vola}(x)$  per default. In generale, una regola di default ha la forma

$$P : J_1, \dots, J_n / C$$

dove  $P$  è il prerequisito,  $C$  la conclusione e  $J_i$  le giustificazioni: se una qualsiasi di esse può essere dimostrata falsa, non si può trarre la conclusione. Tutte le variabili che compaiono in  $J_i$  o  $C$  devono anche comparire in  $P$ . L’esempio del rombo di Nixon può essere rappresentato in logica di default con un fatto e due regole di default:

$$\text{Repubblicano}(\text{Nixon}) \wedge \text{Quacchero}(\text{Nixon}) .$$

$$\text{Repubblicano}(x) : \neg \text{Pacifista}(x) / \neg \text{Pacifista}(x) .$$

$$\text{Quacchero}(x) : \text{Pacifista}(x) / \text{Pacifista}(x) .$$

Per interpretare il significato delle regole di default definiamo il concetto di **estensione** di una teoria di default come l’insieme massimale di conseguenze di tale teoria. In altre parole, un’estensione  $S$  consiste nei fatti noti originali e nell’insieme di conclusioni delle regole di default tale che non sia possibile trarre nuove conclusioni da  $S$  e che le giustificazioni di ogni conclusione di default in  $S$  siano consistenti con  $S$  stesso. Come nel caso dei modelli preferiti nella circoscrizione, abbiamo due possibili estensioni per il rombo di Nixon: in una è pacifista, nell’altra non lo è. Esistono schemi con priorità nei quali alcune regole di default possono avere la precedenza su altre, permettendo così di risolvere qualche ambiguità.

circoscrizione con priorità

logica di default  
regole di default

estensione

Dal 1980, anno in cui sono state proposte per la prima volta le logiche non monotone, sono stati fatti molti progressi nella comprensione delle loro proprietà matematiche: a partire dalla fine degli anni '90, sistemi basati sulla programmazione logica e utilizzabili in pratica hanno cominciato a mostrare molto potenziale come strumenti per la rappresentazione della conoscenza. Tuttavia, rimangono ancora molte questioni irrisolte. Ad esempio, se la formula “le macchine hanno quattro ruote” è falsa, che cosa significa averla nella propria base di conoscenza? Se non possiamo decidere se una regola, presa singolarmente, appartiene o no alla base di conoscenza, sorgono dei seri problemi di non-modularità. Infine, com’è possibile usare credenze di default per prendere decisioni? Questo è probabilmente il problema più spinoso del ragionamento di default. Le decisioni spesso richiedono di valutare dei compromessi, e quindi diventa necessario confrontare la *forza* delle credenze nei risultati di azioni diverse. Nei casi in cui viene preso ripetutamente lo stesso tipo di decisioni, è possibile interpretare le regole di default come asserzioni di “soglia di probabilità”: ad esempio, la regola di default “i miei freni sono sempre a posto” significherà in realtà “la probabilità che i miei freni siano a posto, in assenza di altre informazioni, è sufficientemente alta che la decisione ottima sarà sempre quella di cominciare a guidare senza controllarli”. Quando il contesto della decisione muta – ad esempio, quando si sta guidando un autocarro carico lungo una ripida strada di montagna – la regola di default diventa improvvisamente inappropriata, anche se non ci sono nuove informazioni che facciano pensare che i freni sono rotti. Queste considerazioni hanno portato alcuni ricercatori a considerare la possibilità di integrare il ragionamento di default nella teoria della probabilità.

## 10.8 Sistemi di mantenimento della verità

Nel paragrafo precedente abbiamo argomentato che molte delle inferenze prodotte da un sistema di rappresentazione della conoscenza avranno solo uno stato di default, anziché essere assolutamente certe. Inevitabilmente alcuni di questi fatti inferiti si riveleranno errati e, alla luce di ulteriori informazioni, dovranno essere ritrattati. Questo processo prende il nome di **revisione delle credenze**.<sup>15</sup> Supponiamo che un base di conoscenza *KB* contenga una formula *P* errata – forse una

revisione delle  
credenze

<sup>15</sup> La revisione delle credenze viene spesso contrapposta all’aggiornamento delle credenze, che si verifica quando una base di conoscenza viene modificata per riflettere un cambiamento nel mondo piuttosto che la disponibilità di ulteriori informazioni riguardanti un mondo fisso. L’aggiornamento delle credenze combina la revisione delle credenze con il ragionamento sul tempo e il cambiamento; è imparentato anche al processo di **filtraggio** che descriveremo nel Capitolo 15, nel 2° volume.

conclusione di default ottenuta da un processo di concatenazione in avanti, o forse semplicemente un'asserzione scorretta – e di voler eseguire l'azione  $\text{TELL}(KB, \neg P)$ . Per non creare una contraddizione, si deve prima eseguire  $\text{RETRACT}(KB, P)$ . Fin qui tutto sembra facile. I problemi sorgono quando formule *aggiuntive* sono state inferite da  $P$  e inserite nella KB: ad esempio, l'implicazione  $P \Rightarrow Q$  potrebbe aver causato l'aggiunta di  $Q$ . La soluzione intuitiva di rimuovere anche tutte le formule inferite da  $P$  non è praticabile, perché tali formule potrebbero avere altre giustificazioni oltre a  $P$ . Riprendendo l'esempio di prima, se nella KB fossero presenti anche  $R$  e  $R \Rightarrow Q$ , allora  $Q$  non dovrebbe affatto essere rimossa. I sistemi di mantenimento della verità, o TMS, sono progettati per gestire proprio questo tipo di complicazioni.

Un approccio molto semplice al mantenimento della verità consiste nel tener traccia dell'ordine con cui le formule sono inserite nella base di conoscenza numerandole da  $P_1$  a  $P_n$ . Al momento della chiamata  $\text{RETRACT}(KB, P_i)$  il sistema ritorna allo stato immediatamente precedente all'inserimento di  $P_i$ , rimuovendo quindi sia  $P_i$  che tutte le formule da essa inferite. In seguito, le formule da  $P_{i+1}$  a  $P_n$  possono essere aggiunte nuovamente. Questa soluzione è semplice, e garantisce la consistenza della base di conoscenza, ma la ritrattazione di  $P_i$  richiede che siano ritrattate (e poi asserte nuovamente)  $n - i$  formule; quel che è peggio è che dovranno essere annullate e rieseguite anche tutte le inferenze tratte da esse. Per i sistemi con un grande numero di fatti, come le basi di dati commerciali, tutto ciò non è praticabile.

Un approccio più efficiente è rappresentato dai sistemi di mantenimento della verità basati sulla giustificazione, o JTMS. In un JTMS, ogni formula nella base di conoscenza è annotata con una **giustificazione** che consiste nell'insieme di formule da cui è stata inferita. Ad esempio, se la base di conoscenza contiene già  $P \Rightarrow Q$ , allora  $\text{TELL}(P)$  causerà anche l'aggiunta di  $Q$  con la giustificazione  $\{P, P \Rightarrow Q\}$ . In generale, una formula può avere un numero qualsiasi di giustificazioni. Esse servono a rendere efficienti le ritrattazioni: alla chiamata  $\text{RETRACT}(P)$ , il JTMS cancellerà esattamente le formule in cui  $P$  è membro di ogni giustificazione. Così una formula  $Q$  con la singola giustificazione  $\{P, P \Rightarrow Q\}$  sarà rimossa, con la giustificazione aggiuntiva  $\{P, P \vee R \Rightarrow Q\}$  sarà ancora rimossa, ma questo non accadrà nel caso la giustificazione sia  $\{R, P \vee R \Rightarrow Q\}$ . In questo modo il tempo richiesto per la ritrattazione di  $P$  dipende solo dal numero di formule derivate da essa e non da tutte le altre formule eventualmente aggiunte alla base di conoscenza dopo l'inserimento di  $P$ .

Il JTMS presume sempre che le formule considerate una volta lo saranno probabilmente ancora, così invece di cancellarla fisicamente dalla base di conoscenza non appena ha perso tutte le giustificazioni, una formula viene semplicemente marcata come se fosse assente. Se un'asserzione successiva ripristina una delle sue giustificazioni, si può marcare di nuovo la formula come presente. In questo modo il JTMS conserva tutte le catene di inferenza utilizzate e non deve derivare daccapo le formule una volta che sono nuovamente giustificate.

Oltre a gestire la ritrattazione di informazioni scorrette, i TMS possono essere usati per velocizzare l'analisi di situazioni ipotetiche multiple. Supponiamo che il Comitato Olimpico Rumeno stia scegliendo i luoghi per gli eventi di nuoto, atletica ed equitazione ai Giochi del 2048, che si terranno in Romania. Supponiamo che la prima ipotesi sia *Sito(Nuoto, Pitesti)*, *Sito(Atletica, Bucarest)* e *Sito(Equitazione, Arad)*. Per calcolare tutte le conseguenze logiche e quindi l'opportunità di questa scelta occorrerà svolgere una gran mole di ragionamento. Se poi vogliamo considerare la scelta *Sito(Atletica, Sibiu)*, il TMS ci permetterà di non ricalcolare tutto daccapo: basterà ritrattare *Sito(Atletica, Bucarest)* e asserire al suo posto *Sito(Atletica, Sibiu)*; il TMS si prenderà cura di tutte le revisioni necessarie. Le catene di inferenza generate quando la scelta era caduta su Bucarest potranno essere riusate nel caso di Sibiu, a patto che le conclusioni siano le stesse.

Un sistema di mantenimento di verità basato su assunzioni, o ATMS, è progettato per rendere particolarmente efficiente questo tipo di "cambio di contesto" tra mondi ipotetici. In un JTMS, la memorizzazione delle giustificazioni permette di muoversi rapidamente da uno stato a un altro con qualche ritrattazione e assezione, ma in ogni istante è rappresentato un solo stato. Un ATMS rappresenta contemporaneamente *tutti* gli stati mai considerati. Laddove un JTMS si limita a etichettare ogni formula come *in* o *out* dalla KB, un ATMS tiene traccia, per ogni formula, di quali assunzioni la renderebbero vera. Una formula vale solo nei casi in cui tutte le assunzioni di uno degli insiemi di assunzioni sono verificate.

I sistemi di mantenimento della verità forniscono anche un meccanismo per generare **spiegazioni**. Tecnicamente, una spiegazione di una formula *P* è un insieme di formule *E* tali che *E* implica *P*. Se le formule in *E* sono già note come vere, allora *E* fornisce semplicemente la base per dimostrare che *P* deve valere. Le spiegazioni tuttavia possono anche includere **assunzioni**: formule che non sono note come vere, ma basterebbero a dimostrare *P* se lo fossero. Si potrebbe non avere abbastanza informazioni per dimostrare che la propria macchina non partirà, ma una spiegazione ragionevole potrebbe includere l'assunzione che la batteria sia scarica. Questo, insieme alla conoscenza del funzionamento delle macchine spiegherebbe il comportamento osservato. Nella maggior parte dei casi preferiremo le spiegazioni minime, intendendo con questo che non deve esistere un sottoinsieme proprio della spiegazione che sia anch'esso una spiegazione. Un ATMS può generare spiegazioni per il problema "la macchina non parte" formulando assunzioni (come "serbatoio pieno" o "batteria scarica") in qualsiasi ordine desideriamo, anche se alcune di esse sono contraddittorie. Alla fine potremo leggere l'etichetta della formula "la macchina non parte" e leggere gli insiemi di assunzioni che la giustificano.

Gli algoritmi che implementano i sistemi di mantenimento della verità sono un po' complessi, e non li tratteremo qui. La complessità computazionale del problema è almeno pari a quello dell'inferenza proposizionale, ovvero NP-difficile. Di conseguenza, non ci si può aspettare che il mantenimento della verità rappresenti una panacea: usato accortamente, comunque, un TMS può fornire un incremento significativo nella capacità di un sistema logico di gestire ambienti e ipotesi complesse.

ATMS

spiegazioni

assunzioni

## 10.9 Riepilogo

Di tutti i capitoli incontrati sin qui, questo è stato il più articolato. Avendo esplorato i particolari della rappresentazione di molti tipi di conoscenza, speriamo di aver dato ai lettori un'idea di come sono costruite le basi di conoscenza reali. I punti più importanti sono i seguenti.

- ❖ La rappresentazione della conoscenza su larga scala richiede un'ontologia generale per organizzare e collegare i diversi domini della conoscenza.
- ❖ Un'ontologia generale deve coprire una grande varietà di conoscenze diverse e dovrebbe essere capace, in linea di principio, di gestire qualsiasi dominio.
- ❖ Abbiamo presentato un'**ontologia superiore** basata sulle categorie e sul calcolo degli eventi. Abbiamo trattato gli oggetti strutturati, lo spazio e il tempo, il cambiamento, i processi, le sostanze e le credenze.
- ❖ Le azioni, gli eventi e il tempo possono essere rappresentati con il calcolo delle situazioni o ricorrendo a rappresentazioni più espansive, come il calcolo degli eventi o quello dei fluenti. Tali rappresentazioni permettono a un agente di costruire piani per mezzo dell'inferenza logica.
- ❖ Gli stati mentali degli agenti possono essere rappresentati per mezzo di stringhe che denotano credenze.
- ❖ Abbiamo presentato un'analisi dettagliata del dominio dello shopping su Internet, mettendo alla prova l'ontologia generale e mostrando come la conoscenza del dominio può essere sfruttata da un agente dedicato allo shopping.
- ❖ Per organizzare le gerarchie di categorie sono stati sviluppati sistemi di rappresentazione specializzati, come le **reti semantiche** e le **logiche descrittive**. L'**ereditarietà** è una forma importante di inferenza, che permette di dedurre le proprietà degli oggetti dalle categorie a cui appartengono.
- ❖ L'**ipotesi del mondo chiuso**, com'è implementata nei programmi logici, rappresenta un modo semplice di evitare di specificare una gran quantità di informazione negativa. Il modo migliore di interpretarla è come un **default** che può essere sovrascritto da informazione aggiuntiva.
- ❖ Le **logiche non monotone**, come la **circoscrizione** e la **logica di default**, hanno lo scopo di catturare il ragionamento di default in generale. La **programmazione con insiemi di risposta** velocizza l'inferenza non monotona, proprio come WALKSAT velocizza l'inferenza proposizionale.
- ❖ I **sistemi di mantenimento della verità** gestiscono efficientemente gli aggiornamenti e la revisione della conoscenza.

## Note storiche e bibliografiche

Secondo una teoria plausibile (Briggs, 1985), la ricerca sulla rappresentazione formale della conoscenza ebbe inizio nell'India classica con l'analisi della grammatica del sanscrito shastrico, che risale al primo millennio a. C. In Occidente, le definizioni dei termini da parte dei matematici della Grecia antica possono essere considerate le testimonianze più antiche. In effetti, lo sviluppo di terminologia tecnica in qualsiasi campo può essere considerata una forma di rappresentazione della conoscenza.

Le prime discussioni all'interno dell'IA tendevano a focalizzarsi sulla "rappresentazione del *problema*" piuttosto che sulla "rappresentazione della *conoscenza*" (v. ad esempio la trattazione di Amarel (1968) del problema dei Missionari e dei Cannibali). Negli anni '70 ebbe molta importanza lo sviluppo di "sistemi esperti" (chiamati anche "sistemi basati su conoscenza") che potevano, una volta fornita loro appropriata conoscenza del dominio, uguagliare o superare le prestazioni degli esperti umani su compiti molto precisi. Il primo sistema esperto, DENDRAL (Feigenbaum et al., 1971; Lindsay et al., 1980), era in grado di interpretare l'output di uno spettrometro di massa (uno strumento usato per analizzare la struttura dei composti chimici organici) con la stessa accuratezza di chimici esperti. Benché il suo successo fosse fondamentale per convincere la comunità dell'IA dell'importanza della rappresentazione della conoscenza, i formalismi usati da DENDRAL erano strettamente legati al dominio della chimica. Col tempo i ricercatori hanno cominciato a rivolgere l'attenzione verso formalismi di rappresentazione della conoscenza standardizzati e ontologie che potessero facilitare il processo di creazione di nuovi sistemi esperti. Così facendo, cominciarono ad avventurarsi in un territorio precedentemente esplorato dai filosofi della scienza e del linguaggio. La disciplina imposta dalla necessità che le teorie "funzionino" ha portato a sviluppi più rapidi e profondi di quanto questi problemi erano esclusivo appannaggio della filosofia (benché talvolta abbia anche portato alla ripetuta re-invenzione della ruota).

La creazione di tassonomie complete e di classificazioni risale ai tempi antichi. Aristotele (384–322 a. C.) enfatizzò fortemente gli schemi di classificazione e categorizzazione. Il suo *Organon*, una serie di opere dedicate alla logica raccolte dai suoi discepoli dopo la sua morte, includeva un trattato chiamato *Categorie* in cui cercava di costruire quella che oggi chiameremmo un'ontologia superiore. Aristotele introdusse anche le nozioni di *genere* e *specie* per la classificazione di livello inferiore, benché i termini non avessero il significato odierno, specificatamente biologico. Il nostro sistema di classificazione biologica, che include l'uso della "nomenclatura binomiale" (classificazione per genere e specie in senso tecnico), fu inventato dal biologo svedese Carlo Linneo, o Carl von Linne (1707–1778). I problemi associati ai tipi naturali e ai limiti sfumati tra le categorie sono stati considerati, tra gli altri, da Wittgenstein (1953), Quine (1953), Lakoff (1987) e Schwartz (1977).

L'interesse per le ontologie di grande scala sta aumentando. Il progetto CYC (Lenat, 1995; Lenat e Guha, 1990) ha rilasciato gratuitamente un'ontologia superiore con 6.000 concetti e 60.000 fatti, e mette a disposizione su licenza un'on-

tologia globale molto più grande. L'IEEE ha costituito il sottocomitato P1600.1, lo Standard Upper Ontology Working Group, e la Open Mind Initiative ha assoldato oltre 7.000 utenti Internet per inserire più di 400.000 fatti riguardanti concetti di buonsenso comune. Sul Web stanno emergendo standard come RDF, XML e Semantic Web (Berners-Lee et al., 2001), ma non sono ancora molto diffusi. Gli atti dei congressi sulla *Formal Ontology in Information Systems* (FOIS) contengono molti articoli interessanti sulle ontologie sia generali che specifiche di qualche dominio.

La tassonomia usata in questo capitolo è stata sviluppata dagli autori ed è basata in parte sulla loro esperienza sul progetto CYC e in parte sul lavoro di Hwang e Schubert (1993) e Davis (1990). Una discussione molto interessante del progetto generale di rappresentazione della conoscenza di senso comune compare nel "Naive Physics Manifesto" di Hayes (1978, 1985b).

La rappresentazione del tempo, del cambiamento, delle azioni e degli eventi è stata studiata estensivamente in filosofia e informatica teorica oltre che nell'IA. L'approccio più antico è quello della **logica temporale**, una logica specializzata in cui ogni modello descrive, anziché una struttura relazionale statica, una traiettoria completa attraverso il tempo (che può essere lineare o diramarsi). La logica include **operatori modali** che si possono applicare alle formule;  $\Box p$  significa " $p$  sarà sempre vero nel futuro";  $\Diamond p$  significa " $p$  sarà vero in qualche istante nel futuro". Lo studio della logica temporale ebbe inizio con Aristotele e la scuola stoica di Megara. Nei tempi moderni, Findlay (1941) è stato il primo a suggerire un calcolo formale per il ragionamento sul tempo, ma il lavoro più influente è considerato quello di Arthur Prior (1967). Tra i libri di testo ci sono quelli di Rescher e Urquhart (1971) e van Benthem (1983).

Per lungo tempo gli informatici teorici sono stati interessati alla formalizzazione delle caratteristiche dei programmi, considerati come sequenze di azioni computazionali. Burstall (1974) introdusse l'idea di usare operatori modali per ragionare sui programmi per computer. Poco dopo, Vaughan Pratt (1976) inventò la **logica dinamica**, i cui operatori modali indicano gli effetti dell'esecuzione dei programmi e di altre azioni (v. anche Harel, 1984). Nella logica dinamica, se  $\alpha$  è il nome di un programma, " $[\alpha]p$ " significa " $p$  sarebbe vero in tutti gli stati del mondo risultanti dall'esecuzione del programma  $\alpha$  nello stato corrente" e " $\langle\alpha\rangle p$ " significa " $p$  sarebbe vero in almeno uno degli stati del mondo risultanti dall'esecuzione del programma  $\alpha$  nello stato corrente". La logica dinamica è stata applicata all'analisi di programmi reali da Fischer e Ladner (1977). Pnueli (1977) introdusse l'idea di utilizzare la logica temporale classica per ragionare sui programmi.

Laddove la logica temporale integra il tempo direttamente nel linguaggio, le rappresentazioni del tempo nell'IA hanno solitamente preferito incorporare assiomi esplicativi riguardanti gli eventi e gli aspetti temporali nella base di conoscenza, non assegnando così al tempo una posizione speciale nella logica. In certi casi quest'approccio consente di ottenere una maggior chiarezza e flessibilità. Inoltre, una conoscenza temporale espressa in logica del primo ordine può essere integrata più facilmente con altra conoscenza già accumulata.

La prima trattazione del tempo e delle azioni in IA è dovuta al calcolo delle situazioni di John McCarthy (1963). Il primo sistema di IA a utilizzare estensivamente il ragionamento generale sulle azioni usando la logica del primo ordine è stato QA3 (Green, 1969b). Kowalski (1979b) sviluppò l'idea di reificare le proposizioni nel calcolo delle situazioni.

Il problema del frame fu identificato per la prima volta da McCarthy e Hayes (1969). Molti ricercatori considerarono il problema insolubile con la logica del primo ordine, cosa che diede un grande abbrivio alla ricerca nel campo delle logiche non monotone. Filosofi da Dreyfus (1972) a Crockett (1994) hanno citato il problema del frame come uno dei sintomi dell'inevitabile fallimento dell'intera IA. La soluzione parziale al problema di rappresentazione del frame che utilizza gli assiomi di stato successore è dovuta a Ray Reiter (1991); una soluzione al problema inferenziale del frame può essere fatta risalire al lavoro di Holldobler e Schneeberger (1990) su quello che divenne poi noto con il nome di calcolo dei fluenti (Thielscher, 1999). La discussione presentata in questo capitolo è basata in parte sulle analisi di Lin e Reiter (1997) e Thielscher (1999). I libri di Shanahan (1997) e Reiter (2001b) offrono trattamenti completi e moderni del ragionamento sulle azioni nel calcolo delle situazioni.

La soluzione parziale al problema del frame ha risvegliato l'interesse nell'approccio dichiarativo al ragionamento sulle azioni, che dall'inizio degli anni '70 era stato eclissato dai sistemi di pianificazione specializzati (v. Capitolo 11). Sotto la denominazione di **robotica cognitiva** sono stati fatti molti progressi nella rappresentazione logica del tempo e delle azioni. Il linguaggio GOLOG sfrutta l'intera potenza espressiva della programmazione logica per descrivere azioni e piani (Levesque et al., 1997a) ed è stato esteso per gestire azioni concorrenti (Giacomo et al., 2000), ambienti stocastici (Boutilier et al., 2000) e la percezione (Reiter, 2001a).

Il calcolo degli eventi è stato introdotto da Kowalski e Sergot (1986) per gestire il tempo continuo, e ne sono state proposte molte variazioni (Sadri e Kowalski, 1995). Shanahan (1999) ne presenta una buona, sintetica panoramica. Per la stessa ragione James Allen ha introdotto gli intervalli di tempo (Allen, 1983, 1984), sostenendo che fossero molto più naturali delle situazioni per ragionare su eventi prolungati e concorrenti. Peter Ladkin (1986a, 1986b) ha proposto la nozione di intervalli temporali "concavi" (cioè con presenza di interruzioni; sostanzialmente sono equivalenti a un'unione di normali intervalli "convessi") e ha applicato alla rappresentazione del tempo le tecniche dell'algebra astratta. Allen (1991) investiga in modo sistematico la grande varietà di tecniche disponibili per la rappresentazione del tempo. Shoham (1987) descrive la reificazione degli eventi e propone un nuovo schema di sua invenzione per realizzarla. Ci sono molti punti in comune tra l'ontologia basata su eventi presentata in questo capitolo e l'analisi degli eventi del filosofo Donald Davidson (1980). Lo stesso si può dire delle *storie* nell'ontologia dei liquidi di Pat Hayes (1985a).

La questione dell'ontologia delle sostanze ha una lunga storia. Platone ha proposto di considerarle entità astratte completamente distinte dagli oggetti fisici;

robotica cognitiva

mereologia

avrebbe detto *FattoDi(Burro<sub>3</sub>, Burro)* piuttosto che *Burro<sub>3</sub> ∈ Burro*. Questo conduce a una gerarchia di sostanze in cui, ad esempio, *BurroSalato* è una sostanza più specifica di *Burro*. La posizione che abbiamo adottato nel capitolo, e cioè che le sostanze sono categorie di oggetti, fu difesa strenuamente da Richard Montague (1973) ed è stata adottata dal progetto CYC. Copeland (1993) la sottopone a una critica molto severa, ma non priva di punti deboli. L'approccio alternativo che abbiamo menzionato, in cui il burro è un solo oggetto che consiste in tutti gli oggetti burrosi dell'universo, è stata proposta originariamente dal logico polacco Leśniewski (1916). La sua **mereologia** (il nome deriva dal termine greco per “parte”) usava la relazione parte-tutto in sostituzione della teoria matematica degli insiemi, allo scopo di eliminare del tutto entità astratte come gli insiemi stessi. Leonard e Goodman (1940) forniscono un'esposizione di più facile lettura delle sue idee, mentre *The Structure of Appearance* (1977), dello stesso Goodman, le applica a vari problemi di rappresentazione della conoscenza. Benché alcuni suoi aspetti siano alquanto scomodi – ad esempio, la necessità di ricorrere a un meccanismo di ereditarietà separato basato sulle relazioni parti-tutto – l'approccio mereologico fu capace di guadagnarsi il supporto di Quine (1960). Harry Bunt (1985) ha scritto un'analisi approfondita del suo utilizzo nella rappresentazione della conoscenza.

logica modale

Gli oggetti e gli stati mentali sono stati studiati intensamente sia in filosofia che in IA. Il metodo classico per ragionare sulla conoscenza in filosofia è la **logica modale**, che arricchisce la logica del primo ordine con operatori modali come *B* (crede) e *K* (conosce), che prendono come argomenti intere *formule* anziché termini. La teoria della dimostrazione per la logica modale limita la sostituzione all'interno dei contesti modali, ottenendo così l'opacità referenziale. La logica modale della conoscenza fu inventata da Jaakko Hintikka (1962); Saul Kripke (1963) ne definì la semantica in termini di **mondi possibili**. Informalmente, un mondo è possibile per un agente se è consistente con tutto ciò che l'agente conosce: da questo si possono derivare regole di inferenza che coinvolgono l'operatore *K*. Robert C. Moore mette in relazione la logica modale della conoscenza con uno stile di ragionamento che si riferisce direttamente ai mondi possibili della logica del primo ordine (Moore, 1980, 1985). La logica modale può sembrare una disciplina arcana e intimidire, ma ha già trovato applicazioni significative nel campo del ragionamento sull'informazione nei sistemi distribuiti. Il libro *Reasoning about Knowledge* di Fagin et al. (1995) fornisce un'introduzione dettagliata all'approccio modale. Il congresso biennale *Theoretical Aspects of Reasoning About Knowledge* (TARK) tratta le applicazioni della teoria della conoscenza in IA, nell'economia e nei sistemi distribuiti.

La teoria sintattica degli oggetti mentali è stata studiata inizialmente da Kaplan e Montague (1960), che hanno mostrato che può portare a dei paradossi se non viene gestita con attenzione. Negli ultimi anni è diventata popolare in seno all'IA, anche perché tende a modellare naturalmente le credenze come configurazioni fisiche di un computer o di un cervello. Konolige (1982) e Haas (1986) l'hanno uti-

lizzata per descrivere motori di inferenza di potenza limitata, e Morgenstern (1987) ha mostrato come potrebbe essere applicata alla descrizione delle precondizioni di conoscenza nella pianificazione. Nel Capitolo 12 esamineremo metodi di pianificazione di azioni di osservazione basati sulla teoria sintattica. Ernie Davis (1990) offre un eccellente confronto delle teorie della conoscenza sintattica e modale.

Il filosofo greco Porfirio (c. 234–305 a. C.), commentando le *Categorie* di Aristotele, disegnò quella che si potrebbe definire la prima rete semantica. Charles S. Peirce (1909) sviluppò i grafi esistenziali come primo formalismo di rete semantica basato sulla logica moderna. Ross Quillian (1961), spinto dall'interesse verso la memoria umana e l'elaborazione del linguaggio, diede inizio alla ricerca sulle reti semantiche all'interno dell'IA. Un importante articolo di Marvin Minsky (1975) presentò una versione di rete semantica chiamata *frame*; un frame è la rappresentazione di un oggetto o categoria, con attributi e relazioni con altri oggetti e categorie. Benché l'articolo sia servito a spostare l'interesse verso la rappresentazione della conoscenza in sé, fu criticato come una riproposizione di idee riciclate dalla programmazione a oggetti, come l'ereditarietà o l'uso dei valori di default (Dahl et al., 1970; Birtwistle et al., 1973). Non è chiaro quanto gli articoli successivi sulla programmazione orientata agli oggetti siano stati influenzati dai primi studi sulle reti semantiche nell'IA.

La questione della semantica sorse in modo prepotente in relazione alle reti semantiche di Quillian (e di quelli che avevano seguito il suo approccio) data la presenza ubiqua di “collegamenti È-UN” estremamente vaghi: lo stesso si può dire di altri formalismi di rappresentazione della conoscenza, come MERLIN (Moore e Newell, 1973), con le sue misteriose operazioni “flat” e “cover”. Il famoso articolo di Woods (1975) “What's In a Link?” portò all'attenzione dei ricercatori la necessità di definire una semantica precisa per i formalismi di rappresentazione della conoscenza. Brachman (1979) elaborò il discorso e propose alcune soluzioni. L'articolo di Patrick Hayes (1979) “The Logic of Frames” andò ancora più a fondo, sostenendo che “gran parte dei ‘frame’ non è altro che nuova sintassi per la logica del primo ordine”. Drew McDermott's (1978b) in “Tarskian Semantics, or, No Notation without Denotation!” argomentò che l'approccio alla semantica usato nella logica del primo ordine, basato sulla teoria dei modelli, dovrebbe essere applicato a tutti i formalismi per la rappresentazione della conoscenza. Quest'idea rimane controversa; è da notare che lo stesso McDermott ha poi ritrattato la sua posizione in “A Critique of Pure Reason” (Mc-Dermott, 1987). NETL (Fahlman, 1979) era un sistema sofisticato basato su reti semantiche i cui collegamenti È-UN (chiamati “copie virtuali” o VC) erano basati più sulla nozione di “ereditarietà” tipica dei sistemi a frame o della programmazione a oggetti che sulla relazione di sottoinsieme ed erano definiti in modo molto più preciso dei collegamenti di Quillian, che risalivano all'epoca precedente al lavoro di Woods. NETL è un caso particolarmente interessante, perché originariamente doveva essere implementato su hardware parallelo per superare la difficoltà di recuperare informazioni da reti

semantiche molto grandi. David Touretzky (1986) sottopone l'ereditarietà a una rigorosa analisi matematica; Selman e Levesque (1993) discutono la complessità dell'ereditarietà in presenza di eccezioni, mostrando che la maggior parte delle formulazioni sono NP-complete.

Lo sviluppo delle logiche descrittive è l'ultimo passo di una lunga linea di ricerca che ha lo scopo di trovare sottoinsiemi utili della logica del primo ordine per cui l'inferenza sia computazionalmente trattabile. Hector Levesque e Ron Brachman (1987) hanno mostrato che alcuni costrutti logici – in particolare alcuni usi della disgiunzione e della negazione – erano i principali responsabili dell'intrattabilità dell'inferenza logica. Sulla base del sistema KL-ONE (Schmolze e Lipkis, 1983), sono stati sviluppati diversi sistemi che incorporano i risultati dell'analisi teorica della complessità, tra i quali i più notevoli sono KRYPTON (Brachman et al., 1983) e CLASSIC (Borgida et al., 1989). Il risultato è stato un netto incremento della velocità di inferenza e una comprensione molto più profonda dell'interazione tra complessità ed espressività nei sistemi di ragionamento. Calvanese et al. (1999) presenta un riassunto dello stato dell'arte. Andando contro la tendenza comune, Doyle e Patil (1991) hanno sostenuto che restringere l'espressività del linguaggio rende impossibile risolvere certi problemi, o incoraggia gli utenti a superare le restrizioni con mezzi non logici.

I tre formalismi principali per la gestione dell'inferenza non monotona – circoscrizione (McCarthy, 1980), logica di default (Reiter, 1980) e logica modale non monotona (McDermott e Doyle, 1980) – furono tutti presentati in un numero speciale di AI Journal. La programmazione basata su insiemi di risposta può essere considerata un'estensione della negazione come fallimento o come un raffinamento della circoscrizione; la sottostante teoria della semantica del modello stabile fu introdotta da Gelfond e Lifschitz (1988); i sistemi più importanti sono DLV (Eiter et al., 1998) e SMODELS (Niemelä et al., 2000). L'esempio del disco rigido è preso dal manuale utente di SMODELS (Syrjänen, 2000). Lifschitz (2001) discute l'applicazione alla pianificazione della programmazione basata su sistemi di risposta. Brewka et al. (1997) forniscono una buona panoramica dei diversi approcci alla logica non monotona. Clark (1978) tratta l'approccio della negazione come fallimento alla programmazione logica e il completamento di Clark. Van Emden e Kowalski (1976) mostrano che ogni programma Prolog privo di negazioni ha un modello minimo unico. In anni recenti c'è stato un rinnovato interesse nell'applicazione delle logiche non monotone ai sistemi di rappresentazione della conoscenza su grande scala. I sistemi BENINQ per la gestione dei benefici assicurativi rappresentarono forse la prima applicazione commerciale di successo dell'ereditarietà non monotona (Morgenstern, 1998). Lifschitz (2001) discute l'applicazione della programmazione basata su insiemi di risposta alla pianificazione. Una varietà di sistemi di ragionamento non monotoni basati sulla programmazione logica sono documentati negli atti dei congressi su *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

Lo studio dei sistemi di mantenimento della verità ebbe inizio con TMS (Doyle, 1979) e RUP (McAllester, 1980), che erano entrambi essenzialmente dei JTMS. L'approccio ATMS fu descritto in una serie di articoli da Johan de Kleer (1986a, 1986b, 1986c). *Building Problem Solvers* (Forbus e de Kleer, 1993) spiega nei particolari come si possono usare i TMS nelle applicazioni di IA. Nayak e Williams (1997) mostrano come un TMS efficiente rende praticabile la pianificazione in tempo reale delle operazioni di un veicolo spaziale della NASA.

Per ovvie ragioni, questo capitolo non copre *ogni* area della rappresentazione della conoscenza. I tre argomenti principali che abbiamo omesso sono i seguenti.

- ◆ **Fisica qualitativa:** la fisica qualitativa è una branca della rappresentazione della conoscenza che si occupa specificatamente della costruzione di una teoria logica e non numerica degli oggetti fisici e dei processi. Il termine è stato coniato da Johan de Kleer (1975), benché si possa dire che la ricerca abbia avuto inizio con il sistema BUILD di Fahlman (1974), un sofisticato pianificatore per la costruzione di complesse torri fatte di blocchi. Fahlman scoprì durante la progettazione che la maggior parte dello sforzo (nella sua stima l'80%) era rivolta alla modellazione della fisica del mondo per calcolare la stabilità di vari sotto-aggregati di blocchi piuttosto che alla pianificazione in sé. Di conseguenza delineò un ipotetico processo di "fisica intuitiva" per spiegare perché i bambini piccoli possono risolvere i problemi di BUILD senza avere accesso alla sua veloce aritmetica in virgola mobile, necessaria per la modellazione fisica. Hayes (1985a) usa le "storie" – sezioni quadridimensionali di spazio-tempo analoghe agli eventi di Davidson – per costruire una fisica intuitiva dei liquidi abbastanza complessa. Hayes fu il primo a dimostrare che una vasca tappata straboccherà se il rubinetto rimane aperto e che una persona che cade in un lago si ritroverà completamente bagnata. De Kleer e Brown (1985) e Ken Forbus (1985) cercarono di costruire qualcosa di simile a una teoria generale del mondo fisico, basandosi su astrazioni qualitative delle equazioni fisiche. In anni recenti, la fisica qualitativa si è sviluppata al punto che è possibile analizzare una varietà impressionante di sistemi fisici complessi (Sacks e Joskowicz, 1993; Yip, 1991). Le tecniche qualitative sono state applicate alla costruzione di progetti innovativi per orologi, tergilicristalli e robot a sei zampe (Subramanian, 1993; Subramanian e Wang, 1994). La raccolta *Readings in Qualitative Reasoning about Physical Systems* (Weld e de Kleer, 1990) fornisce una buona introduzione.
- ◆ **Ragionamento spaziale:** il ragionamento necessario per navigare nel mondo del wumpus e in quello dello shopping è banale in confronto alla ricca struttura spaziale del mondo reale. I primi tentativi seri di catturare un ragionamento "di senso comune" riguardante lo spazio si possono trovare nel lavoro di Ernest Davis (1986, 1990). Il calcolo dei collegamenti tra regioni di Cohn et al. (1997) supporta una forma di ragionamento spaziale qualitativo e ha condotto a nuovi tipi di sistemi di informazione geografica. Come nel caso

fisica qualitativa

ragionamento spaziale

ragionamento  
psicologico

della fisica qualitativa, un agente può andare molto lontano, per così dire, senza ricorrere a una rappresentazione metrica completa. Quando una tale rappresentazione diventa necessaria, si possono utilizzare tecniche sviluppate nel campo della robotica (v. Capitolo 25, nel 2° vol.).

- ♦ **Ragionamento psicologico:** il ragionamento psicologico richiede lo sviluppo di una *psicologia* funzionante per agenti artificiali, utilizzabile per ragionare su se stessi e sugli altri agenti. Spesso si basa sulla cosiddetta “psicologia popolare”, quella che si ritiene che le persone usino per ragionare sugli esseri umani. Le teorie psicologiche che i ricercatori forniscono ai loro agenti artificiali per ragionare sugli altri agenti sono spesso basate sulla descrizione del progetto degli agenti logici stessi. Oggi come oggi il ragionamento psicologico è utilizzato soprattutto nel contesto della comprensione del linguaggio naturale, in cui la valutazione delle intenzioni dell’oratore ha un’importanza fondamentale.

Le fonti più aggiornate sono costituite dagli atti dei congressi internazionali *Principles of Knowledge Representation and Reasoning. Readings in Knowledge Representation* (Brachman e Levesque, 1985) e *Formal Theories of the Commonsense World* (Hobbs e Moore, 1985) sono due eccellenti antologie; la prima si concentra principalmente sugli articoli di importanza storica sui linguaggi e i formalismi di rappresentazione, la seconda sul processo di accumulo della conoscenza stessa. Davis (1990), Stefik (1995), e Sowa (1999) sono libri di testo introduttivi.

## Esercizi

- 10.1 Scrivete formule per definire gli effetti dell’azione *Scocca* nel mondo del wumpus. Descrivete i suoi effetti sul wumpus e ricordate che scoccare consuma l’unica freccia dell’agente.
- 10.2 All’interno del calcolo delle situazioni, scrivete un assioma per associare l’istante 0 alla situazione  $S_0$  e un altro assioma per associare l’istante  $t$  con qualsiasi situazione sia derivata da  $S_0$  con una sequenza di  $t$  azioni.
- 10.3 In questo esercizio considereremo il problema di pianificare un cammino che porti un robot da una città a un’altra. L’azione base intrapresa dal robot sarà *Vai(x, y)*, che lo porta dalla città  $x$  alla città  $y$  a patto che ci sia una strada diretta che le collega. *StradaDiretta(x, y)* è vero se e solo se esiste una strada diretta da  $x$  a  $y$ ; potete presumere che tutti questi fatti siano già presenti nella KB (v. mappa a pag. 86). Il robot parte da Arad e deve raggiungere Bucarest.
  - a. Scrivete una descrizione logica adeguata della situazione iniziale del robot.
  - b. Scrivete un’interrogazione logica adeguata le cui soluzioni potranno fornire cammini possibili verso l’obiettivo.

- c. Scrivete una formula che descrive l'azione *Vai*.
- d. Ora supponete che seguire una strada diretta tra due città consumi un quantitativo di carburante pari alla distanza tra le due città. Il robot parte con il serbatoio pieno. Arricchite la vostra rappresentazione per includere queste considerazioni: la vostra descrizione delle azioni dev'essere tale che la query specificata in precedenza possa ancora restituire piani attuabili.
- e. Descrivete la situazione iniziale e scrivete una o più nuove regole per descrivere l'azione *Vai*.
- f. Ora supponete che alcuni dei nodi della mappa contengano anche dei benzinali, in corrispondenza dei quali il robot potrà fare il pieno. Estendete la vostra rappresentazione e scrivete tutte le regole necessarie per descrivere i benzinali, compresa l'azione *RiempিSerbatoio*.

10.4 Investigate i possibili modi di estendere il calcolo degli eventi per gestire eventi *simultanei*. È possibile evitare l'esplosione combinatoria degli assiomi?

10.5 Rappresentate le seguenti sette formule utilizzando ed estendendo le rappresentazioni che abbiamo sviluppato nel capitolo.

- a. L'acqua è liquida tra 0 e 100 gradi.
- b. L'acqua bolle a 100 gradi.
- c. L'acqua nella borraccia di Giovanni è gelata.
- d. La Perrier è un tipo d'acqua.
- e. Giovanni ha della Perrier nella borraccia.
- f. Tutti i liquidi hanno un punto di congelamento.
- g. Un litro d'acqua pesa più di un litro di alcool.

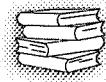
Ora ripetete l'esercizio usando una rappresentazione basata sull'approccio mereologico in cui, ad esempio, *Acqua* sia un oggetto che contiene tutte le parti d'acqua esistenti al mondo.

10.6 Scrivete le definizioni di:

- a. *ScomposizioneEsaustivaInParti*
- b. *PartizioneInParti*
- c. *PartiDisgiunte*

Le definizioni dovrebbero essere analoghe a quelle di *ScomposizioneEsaustiva*, *Partizione* e *Disgiunte*. Pensate che *PartizioneInParti(s, MucchioDi(s))* sia vero? Se pensate di sì, dimostratelo; in caso contrario fornite un controesempio e definite condizioni sufficienti perché sia vero.

10.7 Scrivete un insieme di formule che permetta di calcolare il prezzo di un singolo pomodoro (o altro oggetto) partendo dal prezzo al chilo. Estendete la teoria per permettere il calcolo del prezzo di un sacchetto di pomodori.



- 10.8 Uno schema alternativo per la rappresentazione delle misure prevede l'applicazione della funzione di unità a un oggetto "lunghezza" astratto. Secondo tale schema si scriverebbe  $\text{Pollici}(\text{Lunghezza}(L_1)) = 1.5$ . Come si comporta questo schema rispetto a quello che abbiamo visto nel capitolo? I punti da considerare includono gli assiomi di conversione, i nomi delle quantità astratte (come "50 dollari") e i confronti tra misure astratte in unità diverse (50 pollici è più di 50 centimetri).
- 10.9 Costruite una rappresentazione del cambio valutario che permetta variazioni quotidiane nei fattori di cambio.
- 10.10 Questo esercizio considera le relazioni tra categorie di eventi e gli intervalli di tempo in cui si verificano.
- Definite il predicato  $T(c, i)$  in termini di *Durante* e  $\in$ .
  - Spiegate precisamente perché non sono necessarie due notazioni diverse per descrivere categorie di eventi congiuntive.
  - Date una definizione formale di  $T(\text{UnaDi}(p, q), i)$  e  $T(\text{UnaOAltra}(p, q), i)$ .
  - Spiegate perché è sensato avere due forme di negazioni di eventi, analoghe alle due forme di disgiunzione. Chiamatele *Non* e *Mai* e date loro definizioni formali.
- 10.11 Definite il predicato *Fisso*, dove  $\text{Fisso}(\text{Posizione}(x))$  significa che la posizione di  $x$  è fissa nel tempo.
- 10.12 Definite i predicati *Precede*, *Segue*, *Durante* e *Sovrapposti*, usando il predicato *Consecutivi* e le funzioni *Inizio* e *Fine*, ma non la funzione *Tempo* o il predicato  $<$ .
- 10.13 Nel Paragrafo 10.5 abbiamo usato i predicati *Link* e *LinkText* per descrivere i collegamenti tra le pagine web. Usando tra gli altri i predicati *InTag* e *GetPagina*, scrivete le definizioni di *Link* e *LinkText*.
- 10.14 Una parte del processo di shopping che non abbiamo trattato è il controllo della compatibilità tra gli oggetti acquistati. Ad esempio, se un cliente ordina un computer, le periferiche sono quelle giuste? E se l'acquisto riguarda una macchina fotografica digitale, è associata alla giusta scheda di memoria e tipo di batterie? Scrivete una base di conoscenza il cui compito sarà decidere se un insieme di oggetti è compatibile e che potrà, nel caso, suggerire sostituzioni o acquisti aggiuntivi. Assicuratevi che la KB funzioni con almeno una linea di prodotti e sia facilmente estensibile ad altre linee.
- 10.15 Aggiungete regole per estendere la definizione del predicato *Nome(s, c)* in modo che una stringa come "computer portatile" corrisponda ai nomi delle categorie appropriate di più negozi online. Cercate di rendere la definizione il più generale possibile: per metterla alla prova prendete in esame dieci siti di shopping e guardate il nome che danno a tre diverse categorie. Ad esempio, nel caso dei computer portatili, potremmo trovare i nomi

“portatili”, “laptop”, “computer portatili”, “notebook”, “laptop e notebook” e “PC portatili”. Alcuni di questi casi possono essere trattati con esplicativi fatti *Nome*, altri dovranno essere gestiti con regole per la gestione dei plurali, delle congiunzioni etc.

- 10.16 Trovare una soluzione completa al problema della corrispondenza inesatta tra la descrizione dell’acquirente e quella dei negozi è molto complesso e richiede un sistema capace di applicare tecniche avanzate di elaborazione del linguaggio naturale e di recupero di informazione, che tratteremo nel 2° volume nei Capitoli 22 e 23. Un piccolo passo avanti è permettere all’utente di specificare i valori minimi e massimi di certi attributi. Chiediamo all’utente di usare la seguente grammatica per descrivere i prodotti:

*Descrizione* → *Categoria* [Connettore *Modificatore*]\*

*Connettore* → “con” | “e” | “,”

*Modificatore* → *Attributo* | *Attributo Op Valore*

*Op* → “=” | “>” | “<”

*Categoria* rappresenta un nome di prodotto, *Attributo* una caratteristica come “CPU” o “prezzo”, *Valore* un valore obiettivo per l’attributo. Così, la query “computer con una CPU di almeno 2.5 GHz di costo inferiore a 1000 dollari” dovrà essere scritta “computer con CPU > 2.5 GHz e prezzo < \$1000”. Implementate un agente per lo shopping che accetta descrizioni in questo linguaggio.

- 10.17 La nostra descrizione dello shopping su Internet ha omesso un passo abbastanza importante: l’effettivo *acquisto* del prodotto. Fornite una descrizione in logica formale di un acquisto usando il calcolo degli eventi. In altre parole, dovete definire la sequenza di eventi che si verifica quando un acquirente esegue un acquisto con la carta di credito, compreso il successivo addebitamento e la consegna del prodotto.
- 10.18 Descrivete l’evento che si verifica quando qualcuno baratta un oggetto con un altro. Descrivete poi l’acquisto come un baratto in cui uno dei due oggetti scambiati è una somma di denaro.
- 10.19 I due esercizi precedenti presumevoano l’esistenza di una nozione, per quanto primitiva, di proprietà. All’inizio, per esempio, l’acquirente *possiede* il denaro necessario all’acquisto sotto forma di contante. Questa struttura comincia a scricchiolare quando i propri soldi sono in banca, perché in tal caso non esiste più un oggetto specifico come una collezione di banconote in proprio possesso. Il quadro si complica ulteriormente quando entrano in gioco gli anticipi, i prestiti, gli affitti e il leasing. Investigate i vari concetti legati alla proprietà sia dal punto di vista legale che del senso comune, e proponete uno schema che possa rappresentarli formalmente.



**10.20** Dovete creare un sistema per consigliare a studenti universitari di informatica quali corsi frequentare per soddisfare i requisiti del piano di studi (fate riferimento alla vostra stessa università). Per prima cosa scegliete un vocabolario per la rappresentazione di tutta l'informazione necessaria, quindi rappresentatela. Fatto questo scrivete un'interrogazione appropriata affinché il sistema restituisca come soluzione un piano di studi legale. Dovreste permettere che la soluzione sia adattabile alle necessità di studenti diversi: il sistema dovrebbe chiedere quali esami sono già stati sostenuti ed evitare di generare piani di studio che includono quei corsi o corsi equivalenti.

Suggerite diversi modi in cui sarebbe possibile migliorare il sistema: ad esempio, sarebbe possibile prendere in considerazione le preferenze degli studenti, il carico di lavoro, la qualità dei docenti e così via. Spiegate come si potrebbe esprimere logicamente ognuno di questi tipi di conoscenza. Quanto facilmente il vostro sistema potrebbe incorporare quest'informazione per trovare il *miglior* piano di studi per uno studente?

**10.21** La Figura 10.1 mostra i livelli superiori di una gerarchia che contiene tutto. Estendetela affinché includa il maggior numero possibile di categorie reali. Un buon modo di far ciò è coprire tutti gli aspetti della vita di tutti i giorni, includendo oggetti ed eventi. Cominciate al risveglio e proseguite ordinatamente annotando tutto quello che vedete, toccate, fate e pensate. Una piccola selezione casuale potrebbe comprendere: musica, notizie, latte, camminare, guidare, benzina, mensa, parlare, Prof. Martena, curry di pollo, lingua, 7 euro, il giornale e così via. Dovreste produrre sia un singolo schema di gerarchia (su un grande foglio di carta) che una lista degli oggetti e delle categorie con le relazioni soddisfatte dai membri di ognuna di esse. Ogni oggetto deve appartenere a una categoria, e ogni categoria a una gerarchia.

**10.22** (Adattato da un esempio di Doug Lenat). La vostra missione è catturare in forma logica abbastanza conoscenza da rispondere a una serie di domande riguardanti la seguente semplice frase:

Ieri Vincenzo è andato al supermarket SBAM di Via Lomellina e ha comprato due chili di pomodori e un chilo di manzo macinato.

Cominciate a rappresentare il contenuto della frase come una serie di asserzioni. Dovreste scrivere formule che hanno una struttura logica semplice e diretta (ad esempio affermando che gli oggetti hanno certe proprietà, che tra loro esistono determinate relazioni, che tutti gli oggetti che soddisfano una proprietà devono soddisfarne un'altra). Le seguenti domande dovrebbero aiutarvi a cominciare:

- ♦ Quali classi, oggetti e relazioni sono necessari? Quali sono i padri, i "fratelli" e così via nelle gerarchie? Tra le altre cose, dovete definire gli eventi e disporre di un ordinamento temporale.

- ◆ Quale sarebbe il posto delle vostre entità in una gerarchia più generale?
- ◆ Quali vincoli esistono e quali sono le relazioni tra di essi?
- ◆ Quanto dev'essere dettagliato ogni concetto?

La vostra base di conoscenza dev'essere capace di rispondere a una lista di domande che daremo tra poco. Alcune di esse trattano direttamente i fatti narrati esplicitamente nella storia, ma la maggior parte richiederà all'agente di possedere conoscenza di fondo. Dovrete considerare quali cose si trovano in un supermarket, che cosa significa acquistarle, a cosa serviranno gli oggetti acquistati e così via. Cercate di rendere la vostra rappresentazione il più generale possibile. Per fare un esempio banale, non dite "la gente compra cibo alla SBAM", perché questo non vi aiuterà nel caso qualcuno faccia la spesa in un altro supermercato. Non dite "Joe cucinò gli spaghetti con i pomodori e il manzo macinato", perché questo non vi aiuterà a trattare nessun caso se non quello specificato. Inoltre, state attenti a non trasformare le domande in risposte: ad esempio, la domanda (c) chiede "Vincenzo ha comprato della carne?" e non "Vincenzo ha comprato un chilo di manzo macinato?".

Abbozzate le catene di ragionamento che porteranno alla soluzione delle domande. Facendo ciò, senza dubbio creerete concetti aggiuntivi, farete nuove asserzioni e così via. Se possibile, usate un sistema di ragionamento automatico per dimostrare la sufficienza della vostra base di conoscenza. Molte delle cose che scriverete saranno solo approssimativamente corrette nel mondo reale, ma non preoccupatevene troppo; l'idea è di estrarre il buon senso comune che rende possibile rispondere a queste domande. Uno svolgimento davvero completo di quest'esercizio è *estremamente* difficile, probabilmente al di là del presente stato dell'arte della rappresentazione della conoscenza. Tuttavia, dovreste essere in grado di identificare un insieme consistente di assiomi per rispondere alle domande limitate poste qui sotto.

- a. Vincenzo è un bambino o un adulto? [Adulto]
- b. Vincenzo ha almeno due pomodori? [Sì]
- c. Vincenzo ha comprato della carne? [Sì]
- d. Se Mariolina stava comprando dei pomodori nello stesso momento di Vincenzo, lui l'ha vista? [Sì]
- e. I pomodori sono prodotti nel supermercato? [No]
- f. Cosa farà Vincenzo con i pomodori? [Li mangerà]
- g. La SBAM vende anche deodoranti? [Sì]
- h. Vincenzo ha portato soldi al supermercato? [Sì]
- i. Al ritorno dal supermercato SBAM Vincenzo ha meno soldi? [Sì]

- 10.23 Apportate le necessarie aggiunte e cambiamenti alla base di conoscenza dell'esercizio precedente in modo che possa rispondere alle domande qui sotto. Mostrate che la KB restituisce effettivamente le risposte corrette e

includete nel resoconto una discussione delle modifiche, spiegando perché erano necessarie, se erano piccole o grandi e così via.

- a. Ci sono altre persone nella SBAM mentre Vincenzo è lì? [Sì – cassiere!]
- b. Vincenzo è vegetariano? [No]
- c. Chi possiede il deodorante contenuto nella SBAM? [l'azienda SBAM]
- d. Vincenzo aveva un etto di manzo macinato? [Sì]
- e. Il benzinaio poco distante ha della benzina? [Sì]
- f. I pomodori stanno nel bagagliaio della macchina di Vincenzo? [Sì]

**10.24** Come ricorderete, l'informazione codificata nelle reti semantiche mediante il meccanismo dell'ereditarietà può essere espressa in logica con adeguate formule di implicazione. In questo esercizio prenderemo in esame l'efficienza di tali formule per realizzare l'ereditarietà.

- a. Considerate il contenuto informativo di un catalogo di auto usate: ad esempio, una Fiat Uno del 1991 vale 200 euro. Supponete che tutta quest'informazione (per 11.000 modelli di vetture) sia codificata per mezzo di regole logiche, in modo analogo a quanto abbiamo presentato nel capitolo. Scrivete tre di queste regole, inclusa quella che riguarda le Fiat Uno del 1991. Come usereste queste regole per determinare il valore di una macchina *particolare* (ad esempio "Kit", che è una Fiat Uno del 1991) con un dimostratore di teoremi basato sulla concatenazione all'indietro come Prolog?
- b. Confrontate l'efficienza temporale del metodo di concatenazione all'indietro rispetto all'ereditarietà utilizzata dalle reti semantiche.
- c. Spiegate in che modo la concatenazione in avanti permette a un sistema basato sulla logica di risolvere lo stesso problema in modo efficiente, presumendo che la KB contenga solo le 11.000 regole che riguardano il prezzo delle vetture.
- d. Descrivete una situazione in cui né la concatenazione in avanti né quella all'indietro permetterà la gestione efficiente di una query sul prezzo di una particolare macchina.
- e. Potete suggerire una soluzione che permetta di risolvere efficientemente questo tipo di interrogazione con un sistema logico, in tutti i casi? [*Suggerimento:* ricordate che due macchine che appartengono alla stessa categoria hanno lo stesso prezzo].

**10.25** Si potrebbe supporre che la distinzione sintattica tra i collegamenti la cui etichetta è racchiusa in un rettangolo e quelli in cui non lo è sia superflua, perché questi ultimi sono sempre associati a categorie: di conseguenza un algoritmo di ereditarietà potrebbe semplicemente dare per scontato che un collegamento privo di rettangolo attaccato a una categoria si applichi a tutti i suoi membri. Mostrate che quest'argomentazione è fallace, fornendo esempi di possibili errori.

# Pianificazione

C  
o  
n  
s  
i  
g  
n  
a  
t  
u  
r  
e  
P  
a  
r  
t  
e  
c  
i  
p  
a  
z  
i  
o  
n  
e  
Q  
u  
a  
l  
i  
t  
a  
t  
e  
C  
o  
n  
s  
i  
g  
n  
a  
t  
u  
r  
e  
P  
a  
r  
t  
e  
c  
i  
p  
a  
z  
i  
o  
n  
e  
Q  
u  
a  
l  
i  
t  
a  
t  
e

## Capitolo 11

# Pianificazione

*In cui vediamo come un agente può sfruttare la struttura di un problema per costruire complessi piani d'azione.*

Il compito di trovare una sequenza di azioni capace di raggiungere un obiettivo prende il nome di **pianificazione**. Fin qui abbiamo visto due esempi di agenti capaci di pianificare: il risolutore di problemi basato sulla ricerca (Capitolo 3) e il pianificatore logico (Capitolo 10). Questo capitolo si occupa principalmente della *scalabilità* delle tecniche di pianificazione: gli approcci visti finora, infatti, non sono in grado di gestire problemi complessi.

Il Paragrafo 11.1 sviluppa un linguaggio espressivo ma attentamente vincolato per la rappresentazione di problemi di pianificazione, incluse le azioni e gli stati. Il linguaggio è molto vicino alle rappresentazioni proposizionali e del primo ordine delle azioni, che abbiamo esaminato nei Capitoli 7 e 10. Il Paragrafo 11.2 illustra come gli algoritmi di ricerca in avanti e all'indietro possano trarre vantaggio da questa rappresentazione, grazie soprattutto a euristiche accurate derivate automaticamente dalla sua stessa struttura. Tutto questo ricorda la costruzione di euristiche utili per i problemi di soddisfacimento di vincoli del Capitolo 5. I Paragrafi da 11.3 a 11.5 descrivono algoritmi di pianificazione che vanno oltre la ricerca in avanti e all'indietro, sfruttando ancora una volta la rappresentazione del problema. In particolare, esamineremo approcci che non si limitano a considerare esclusivamente sequenze di azioni totalmente ordinate.

In questo capitolo considereremo solo ambienti completamente osservabili, deterministici, finiti, statici (in cui cioè si verificano cambiamenti solo quando l'agente agisce) e discreti (per quanto riguarda il tempo, le azioni, gli oggetti e gli effetti). Questi ambienti contraddistinguono quella che viene chiamata **pianificazione classica**. La pianificazione non classica, al contrario, considera ambienti parzialmente osservabili o stocastici e utilizza algoritmi e tecniche di progettazione diverse, che tratteremo nei Capitoli 12 e, nel 2° volume, 17.

pianificazione classica

## 11.1 Il problema della pianificazione

Vediamo quello che può succedere quando un normale agente risolutore di problemi, che usa algoritmi standard di ricerca come quella in profondità, A\* e così via, si trova ad affrontare problemi reali di grandi dimensioni. Questo ci aiuterà a progettare agenti pianificatori migliori.

La difficoltà più evidente è che l'agente può essere sopraffatto dal numero delle azioni irrilevanti. Considerate il compito di acquistare una copia di questo libro in un negozio online. Supponiamo che ci sia un'azione di acquisto per ogni codice ISBN a 10 cifre, per un totale di 10 miliardi di azioni. L'algoritmo di ricerca dovrà esaminare gli stati risultanti da tutti i 10 miliardi di azioni per trovare quella che soddisfa l'obiettivo, che consiste nel possedere una copia del libro con ISBN 0137903952. Un agente pianificatore sensato, d'altra parte, dovrebbe essere in grado di risalire da una descrizione esplicita dell'obiettivo come  $Ha(ISBN0137903952)$  direttamente all'azione  $Acquista(ISBN0137903952)$ . Per far ciò, l'agente deve semplicemente possedere la conoscenza generale del fatto che  $Acquista(x)$  ha come risultato  $Ha(x)$ . Data questa conoscenza e l'obiettivo, il pianificatore può decidere con un solo passo di unificazione che l'azione da eseguire è  $Acquista(ISBN0137903952)$ .

La difficoltà successiva è trovare una buona **funzione euristica**. Supponiamo che l'obiettivo dell'agente sia comprare quattro libri diversi. In questo caso ci saranno  $10^{40}$  piani di quattro passi, ragion per cui non sarà neppure proponibile eseguire una ricerca senza un'euristica accurata. Per un essere umano è ovvio che una buona stima euristica del costo di uno stato è il numero di libri ancora da acquistare; sfortunatamente questa intuizione non può essere condivisa dall'agente risolutore di problemi, che vede il test obiettivo come una scatola nera che per ogni stato restituisce vero o falso. L'agente difetta quindi di autonomia; per ogni nuovo problema dovrà chiedere a un utente umano di fornirgli una funzione euristica. D'altra parte, se l'agente pianificatore ha accesso a una rappresentazione esplicita dell'obiettivo come congiunzione di sotto-obiettivi può sfruttare una singola euristica *indipendente dal dominio*: il numero di congiunti non soddisfatti. Per il problema di acquisto di libri, l'obiettivo sarebbe  $Ha(A) \wedge Ha(B) \wedge Ha(C) \wedge Ha(D)$ , e lo stato contenente  $Ha(A) \wedge Ha(C)$  avrebbe costo 2. In questo modo l'agente ottiene automaticamente l'euristica giusta per questo problema e per molti altri. Nel corso del capitolo vedremo come si possono costruire euristiche più sofisticate che considerano le azioni possibili oltre la struttura dell'obiettivo.

Infine, il risolutore di problemi potrebbe essere inefficiente perché non può avvantaggiarsi della tecnica di **scomposizione dei problemi**. Considerate la spedizione rapida di un insieme di pacchi verso più destinazioni sparse per tutta l'Australia. La cosa più sensata è trovare l'aeroporto più vicino per ogni destinazione e dividere il problema globale in più sottoproblemi, uno per ogni aeroporto. A seconda della destinazione finale si potrà poi scomporre ulteriormente il problema che riguarda i pacchetti che transitano per lo stesso aeroporto. Nel Capitolo 5 ab-

biamo visto che la capacità di effettuare questo tipo di scomposizione contribuisce all'efficienza della risoluzione dei problemi di soddisfacimento di vincoli. La stessa cosa si applica ai pianificatori: nel caso pessimo, trovare il piano migliore per spedire  $n$  pacchetti può richiedere un tempo  $O(n!)$ , che si riduce a  $O((n/k)! \times k)$  se il problema può essere scomposto in  $k$  parti uguali.

Come abbiamo già notato nel Capitolo 5, i problemi perfettamente scomponibili sono gradevoli ma rari.<sup>1</sup> La progettazione di molti sistemi di pianificazione – in particolar modo quella dei pianificatori a ordinamento parziale descritti nel Paragrafo 11.3 – si basa sull'assunto che la maggior parte dei problemi reali sono quasi scomponibili. Questo significa che il pianificatore può lavorare indipendentemente sui sotto-obiettivi, ma l'integrazione dei sottopiani risultanti potrà richiedere lavoro aggiuntivo. In alcuni casi quest'assunto può non valere, perché il raggiungimento di un sotto-obiettivo può annullarne un altro. Queste interazioni tra sotto-obiettivi sono ciò che rende difficili rompicapi come quello a 8 tasselli.

quasi scomponibili

## Il linguaggio dei problemi di pianificazione

La discussione qui sopra suggerisce che la rappresentazione dei problemi di pianificazione – stati, azioni e obiettivi – dovrebbe permettere agli algoritmi di trarre vantaggio dalla struttura logica del problema. La chiave sta nel trovare un linguaggio sufficientemente espressivo da descrivere una grande varietà di problemi, ma abbastanza ristretto da essere utilizzabile da parte di algoritmi efficienti. In questo paragrafo per prima cosa delineeremo il linguaggio di rappresentazione dei pianificatori classici, chiamato STRIPS;<sup>2</sup> più avanti ne indicheremo alcune possibili variazioni.

**Rappresentazione degli stati.** I pianificatori scompongono il mondo in condizioni logiche e rappresentano uno stato come una congiunzione di letterali positivi. Utilizzeremo per lo più letterali proposizionali; ad esempio, *Povero  $\wedge$  Sconosciuto* potrebbe rappresentare lo stato di un agente sfortunato. Useremo anche letterali del primo ordine: *Posizione(Aereo<sub>1</sub>, Melbourne)  $\wedge$  Posizione(Aereo<sub>2</sub>, Sydney)* potrebbe rappresentare uno stato del problema di consegna di pacchi. Nelle descrizioni di stato, i letterali del primo ordine devono essere **ground** e privi di **funzioni**: letterali come *Posizione(x, y)* o *Posizione(Padre(Fred), Sydney)* non sono permessi. Verrà sempre l'**ipotesi del mondo chiuso**, il che significa che tutte le condizioni non menzionate in uno stato saranno da considerare false.

**Rappresentazione degli obiettivi.** Un obiettivo è uno stato parzialmente specificato, rappresentato da una congiunzione di letterali positivi ground come

<sup>1</sup> Notate che anche la consegna di un singolo pacco non è perfettamente scomponibile: ci possono essere casi in cui è meglio assegnare dei pacchi a un aeroporto più distante se questo permette di risparmiare un volo. Nonostante questo, la maggior parte delle compagnie di spedizioni preferiscono la semplicità computazionale e organizzativa delle soluzioni scomposte.

<sup>2</sup> STRIPS sta per STanford Research Institute Problem Solver.

soddisfacimento di obiettivi

*Ricco  $\wedge$  Famoso o Posizione( $P_2$ , Tahiti).* Uno stato proposizionale  $s$  soddisfa un obiettivo  $g$  se contiene tutti gli atomi di  $g$  (e possibilmente altri). Ad esempio, lo stato  $Ricco \wedge Famoso \wedge Tristissimo$  soddisfa l'obiettivo  $Ricco \wedge Famoso$ .

**Rappresentazione delle azioni.** Un'azione è specificata per mezzo delle precondizioni che devono valere prima della sua esecuzione e degli effetti che ne scaturiscono. Ad esempio, l'azione di pilotare un aereo da un posto all'altro si scrive:

*Azione(Vola( $p$ ,  $da$ ,  $a$ ),*

*PRECOND: Posizione( $p$ ,  $da$ )  $\wedge$  Aereo( $p$ )  $\wedge$  Aeroporto( $da$ )  $\wedge$  Aeroporto( $a$ )*

*EFFETTO:  $\neg$ Posizione( $p$ ,  $da$ )  $\wedge$  Posizione( $p$ ,  $a$ )*

schema di azione

Più propriamente questo prende il nome di **schema di azione**, il che significa che rappresenta una molteplicità di azioni distinte che possono essere derivate istanziando le variabili  $p$ ,  $da$  e  $a$  con diverse costanti. In generale uno schema di azione è composto da tre parti.

- ◆ Il nome dell'azione e la lista di parametri – ad esempio *Vola( $p$ ,  $da$ ,  $a$ )* – servono a identificare l'azione.
- ◆ La **precondizione** è una congiunzione di letterali positivi privi di funzioni che specifica ciò che dev'essere vero in uno stato prima che l'azione possa essere eseguita. Tutte le variabili nella precondizione devono anche apparire nella lista dei parametri dell'azione.
- ◆ L'**effetto** è una congiunzione di letterali privi di funzioni che descrive come cambia lo stato quando l'azione viene eseguita. L'effetto asserisce che ogni letterale positivo  $P$  sarà vero nello stato che risulta dall'esecuzione dell'azione; ogni letterale negativo  $\neg P$  sarà falso. Anche le variabili nell'effetto devono apparire nella lista di parametri dell'azione.

effetto

Per migliorare la leggibilità, alcuni sistemi di pianificazione dividono l'effetto in una **lista di aggiunte** per i letterali positivi e una **lista di cancellazioni** per quelli negativi.

Avendo definito la sintassi per la rappresentazione dei problemi di pianificazione, possiamo ora definirne la semantica. Il modo più semplice per farlo è descrivere come le azioni modifichino gli stati (un metodo alternativo consiste nello specificare una traduzione diretta in assiomi di stato successore, la cui semantica deriva dalla logica del primo ordine: v. Esercizio 11.3). Prima di tutto, diciamo che un'azione è **applicabile** in ogni stato che soddisfa la precondizione; in caso contrario l'azione non ha alcun effetto. Per uno schema di azione del primo ordine, verificare l'applicabilità richiede una sostituzione  $\theta$  per le variabili della precondizione. Supponiamo ad esempio che uno stato corrente sia descritto da

$$\begin{aligned} &\text{Posizione}(P_1, JFK) \wedge \text{Posizione}(P_2, SFO) \wedge \text{Aereo}(P_1) \wedge \text{Aereo}(P_2) \\ &\wedge \text{Aeroporto}(JFK) \wedge \text{Aeroporto}(SFO) . \end{aligned}$$

Questo stato soddisfa la precondizione

$$\text{Posizione}(p, da) \wedge \text{Aereo}(p) \wedge \text{Aeroporto}(da) \wedge \text{Aeroporto}(a)$$

lista di aggiunte  
lista di cancellazioni

applicabile

Con la sostituzione  $\{p/P_1, da/JFK, a/SFO\}$  (tra le altre; v. Esercizio 11.2). Ne consegue che l'azione concreta  $Vola(P_1, JFK, SFO)$  è applicabile.

Partendo dallo stato  $s$ , il **risultato** dell'esecuzione di un'azione applicabile  $a$  è lo stato  $s'$  che è identico a  $s$  tranne il fatto che ogni letterale positivo  $P$  nell'effetto di  $a$  è aggiunto a  $s'$  e ogni letterale negativo  $\neg P$  è rimosso da  $s'$ . Quindi, dopo  $Vola(P_1, JFK, SFO)$ , lo stato corrente diventa

$$\begin{aligned} & Posizione(P_1, SFO) \wedge Posizione(P_2, SFO) \wedge Aereo(P_1) \wedge Aereo(P_2) \\ & \wedge Aeroporto(JFK) \wedge Aeroporto(SFO) . \end{aligned}$$

Notate che un effetto positivo già presente in  $s$  non viene aggiunto nuovamente, e se un effetto negativo non è in  $s$  quella parte dell'effetto è ignorata. Questa definizione segue il cosiddetto **assunto di STRIPS**: i letterali non menzionati nell'effetto non sono mai modificati. In questo modo, STRIPS evita il **problema di rappresentazione del frame** descritto nel Capitolo 10.

Possiamo finalmente definire la **soluzione** di un problema di pianificazione. Nella sua forma più semplice è costituita da una sequenza di azioni che, una volta eseguite, porteranno in uno stato che soddisfa l'obiettivo. Nel seguito di questo capitolo permetteremo alle soluzioni di essere insiemi parzialmente ordinati di azioni, a patto che ogni sequenza di azioni che rispetta l'ordinamento parziale sia una soluzione.

risultato

assunto di Strips

soluzione

## Espressività ed estensioni

Le varie restrizioni imposte alla rappresentazione STRIPS sono state scelte nella speranza di rendere gli algoritmi di pianificazione più semplici ed efficienti, senza che la descrizione dei problemi reali risulti troppo difficile. Una delle restrizioni più importanti è che i letterali debbano essere *privi di funzioni*. Grazie a essa si può essere certi che ogni schema di azione per un dato problema possa essere proposizionalizzato, ovvero trasformato in una collezione finita di rappresentazioni di azioni puramente proposizionali e prive di variabili (v. il Capitolo 9 per approfondire quest'argomento). Ad esempio, nel dominio dei trasporti aerei e per un problema con 10 aerei e 5 aeroporti, potremmo tradurre lo schema  $Vola(p, da, a)$  in  $10 \times 5 \times 5 = 250$  azioni puramente proposizionali. I pianificatori dei Paragrafi 11.4 e 11.5 operano direttamente sulle descrizioni proposizionalizzate. Se permetessimo l'uso di simboli di funzione il numero di stati e azioni potrebbe diventare infinito.

In anni recenti è diventato chiaro che STRIPS non è abbastanza espressivo per alcuni domini reali: di conseguenza, ne sono state sviluppate molte varianti. La Figura 11.1 descrive brevemente una delle più importanti, l'**Action Description Language** o **ADL**, confrontandolo con il linguaggio base di STRIPS. In ADL, l'azione  $Vola$  potrebbe essere scritta come segue:

*Azione*( $Vola(p : Aereo, da : Aeroporto, a : Aeroporto)$  ,  
*PRECOND*:  $Posizione(p, da) \wedge (da \neq a)$   
*EFFETTO*:  $\neg Posizione(p, da) \wedge Posizione(p, a))$  .

ADL

Linguaggio STRIPS	Linguaggio ADL
solo letterali positivi negli stati: <i>Povero</i> $\wedge$ <i>Sconosciuto</i>	letterali positivi e negativi negli stati: $\neg$ <i>Ricco</i> $\wedge$ $\neg$ <i>Famoso</i>
ipotesi del mondo chiuso: i letterali non menzionati sono falsi	ipotesi del mondo aperto: i letterali non menzionati sono sconosciuti
l'effetto $P \wedge \neg Q$ significa: aggiungi <i>P</i> e cancella <i>Q</i>	l'effetto $P \wedge \neg Q$ significa: aggiungi <i>P</i> e $\neg Q$ e cancella $\neg P$ e <i>Q</i>
solo letterali ground negli obiettivi: <i>Ricco</i> $\wedge$ <i>Famoso</i>	variabili quantificate negli obiettivi: $\exists x$ <i>Posizione</i> ( <i>P</i> <sub>1</sub> , <i>x</i> ) $\wedge$ <i>Posizione</i> ( <i>P</i> <sub>2</sub> , <i>x</i> ) è l'obiettivo che <i>P</i> <sub>1</sub> e <i>P</i> <sub>2</sub> si trovino nella stessa posizione
gli obiettivi sono congiunzioni: <i>Ricco</i> $\wedge$ <i>Famoso</i>	gli obiettivi ammettono congiunzione e disgiunzione: $\neg$ <i>Povero</i> $\wedge$ ( <i>Famoso</i> $\vee$ <i>Intelligente</i> )
gli effetti sono congiunzioni	sono permessi effetti condizionali: <i>when P : E</i> significa che <i>E</i> è un effetto solo se <i>P</i> è soddisfatta
l'uguaglianza non è supportata	il predicato di uguaglianza ( <i>x = y</i> ) è predefinito
nessun supporto dei tipi	le variabili possono avere tipi, come in ( <i>p : Aereo</i> )

**Figura 11.1** Confronto tra i linguaggi STRIPS e ADL per la rappresentazione di problemi di pianificazione. In entrambi i casi, gli obiettivi si comportano come precondizioni di un'azione priva di parametri.

La notazione  $p : Aereo$  nella lista dei parametri è un'abbreviazione per  $Aereo(p)$  nella precondizione; questo non aggiunge potere espressivo ma può facilitare la lettura (e limita il numero di possibili azioni proposizionali che possono essere costruite). La precondizione ( $da \neq a$ ) esprime il fatto che un volo non può atterrare nello stesso aeroporto di partenza: in STRIPS non avrebbe potuto essere espressa in modo così succinto.

I diversi formalismi usati in IA per la pianificazione sono stati sistematizzati con una sintassi standard chiamata Planning Domain Definition Language (PDDL). Questo linguaggio comune permette ai ricercatori di scambiarsi problemi di riferimento e confrontare i propri risultati. PDDL include sottolinguaggi per STRIPS, ADL e per le reti gerarchiche che tratteremo nel prossimo capitolo.

Le notazioni di STRIPS e ADL sono adeguate a molti domini reali: nei prossimi sottoparagrafi vedremo qualche esempio. Rimangono tuttavia alcune significative restrizioni; la più evidente è che non possono rappresentare in modo naturale le **ramificazioni** di azioni. Ad esempio, quando un aereo vola cambia anche la posizione delle persone, dei pacchi e delle palline di polvere in esso contenute. Possiamo rappresentare questi cambiamenti come effetti diretti del volo, ma sembra più naturale rappresentare la posizione del contenuto dell'aereo come una conse-

---

*Init(Posizione(C<sub>1</sub>, SFO)  $\wedge$  Posizione(C<sub>2</sub>, JFK)  $\wedge$  Posizione(P<sub>1</sub>, SFO)  $\wedge$  Posizione(P<sub>2</sub>, JFK)*

$\wedge$  Merci(C<sub>1</sub>)  $\wedge$  Merci(C<sub>2</sub>)  $\wedge$  Aereo(P<sub>1</sub>)  $\wedge$  Aereo(P<sub>2</sub>)

$\wedge$  Aeroporto(JFK), Aeroporto(SFO))

*Obiettivo(Posizione(C<sub>1</sub>, JFK)  $\wedge$  Posizione(C<sub>2</sub>, SFO))*

*Azione(Carica(c, p, a),*

PRECOND: Posizione(c, a)  $\wedge$  Posizione(p, a)  $\wedge$  Merci(c)  $\wedge$  Aereo(p)  $\wedge$  Aeroporto(a)

EFFETTO:  $\neg$ Posizione(c, a)  $\wedge$  In(c, p))

*Azione(Scarica(c, p, a),*

PRECOND: In(c, p)  $\wedge$  Posizione(p, a)  $\wedge$  Merci(c)  $\wedge$  Aereo(p)  $\wedge$  Aeroporto(a)

EFFETTO: Posizione(c, a)  $\wedge$   $\neg$ In(c, p))

*Azione(Vola(p, da, a),*

PRECOND: Posizione(p, da)  $\wedge$  Aereo(p)  $\wedge$  Aeroporto(da)  $\wedge$  Aeroporto(a)

EFFETTO:  $\neg$ Posizione(p, da)  $\wedge$  Posizione(p, a))

---

**Figura 11.2** Un problema STRIPS che riguarda il trasporto aereo di merci tra due aeroporti.

guenza logica di quella dell'aereo stesso. Vedremo altri esempi di questi vincoli di stato nel Paragrafo 11.5. I sistemi classici di pianificazione non cercano neppure di affrontare il problema della **qualificazione**, cioè la possibilità che si verifichino circostanze non rappresentate tali da causare il fallimento di un'azione. Questo è un argomento che tratteremo nel prossimo capitolo.

vincoli di stato

## Esempio: trasporto aereo di merci

La Figura 11.2 presenta un problema di trasporto aereo che coinvolge il carico/scarico di merci su aerei e il volo di questi ultimi da un posto all'altro. Il problema può essere definito con tre azioni: *Carica*, *Scarica* e *Vola*. Le azioni hanno effetto su due predicati: *In(c, p)* significa che le merci *c* si trovano sull'aereo *p*; *Posizione(x, a)* significa che l'oggetto *x* (un aereo o delle merci) sono all'aeroporto *a*. Notate che le merci non sono in alcuna *Posizione* quando si trovano *In* un aereo, cosicché *Posizione* in effetti significa "disponibile per l'uso in una data posizione". Ci vuole una certa esperienza con la definizione delle azioni per gestire simili dettagli in modo consistente. Il piano seguente è una soluzione del problema:

[*Carica(C<sub>1</sub>, P<sub>1</sub>, SFO)*, *Vola(P<sub>1</sub>, SFO, JFK)*, *Scarica(C<sub>1</sub>, P<sub>1</sub>, JFK)*,

*Carica(C<sub>2</sub>, P<sub>2</sub>, JFK)*, *Vola(P<sub>2</sub>, JFK, SFO)*, *Scarica(C<sub>2</sub>, P<sub>2</sub>, SFO)*].

La nostra rappresentazione è puro STRIPS: in particolare, nulla impedisce a un aereo di decollare e atterrare allo stesso aeroporto. In ADL, per impedirlo si potrebbero usare letterali di uguaglianza.

## Esempio: il problema della ruota di scorta

Considerate il problema di cambiare una ruota bucata. Più precisamente, l'obiettivo consiste nell'avere una ruota di scorta buona montata in maniera corretta sull'asse della macchina, mentre lo stato iniziale ha una ruota bucata sull'asse e una ruota di scorta nel bagagliaio. Per semplicità la nostra versione del problema è molto astratta e non prevede bulloni incastriati o altre complicazioni. Le azioni sono solo quattro: prendere la ruota di scorta dal bagagliaio, rimuovere quella bucata dall'asse, montare la ruota di scorta sull'asse e lasciare la macchina incustodita tutta la notte. Presumiamo che la macchina sia parcheggiata in un quartiere particolarmente malfamato, per cui l'effetto di lasciarla incustodita è la sparizione di tutte le ruote.

La descrizione ADL del problema è mostrata nella Figura 11.3: notate che è puramente proposizionale. C'è un aspetto che STRIPS non avrebbe potuto rappresentare, e cioè la precondizione negata  $\neg\text{Posizione}(\text{Bucata}, \text{Asse})$  dell'azione  $\text{Monta}(\text{Scorta}, \text{Asse})$ . Per evitarla si sarebbe potuto usare al suo posto  $\text{Libero}(\text{Asse})$ , come vedremo nel prossimo esempio.

## Esempio: il mondo dei blocchi

Uno dei più famosi domini di pianificazione è il **mondo dei blocchi**. Questo dominio consiste in un insieme di blocchi di forma cubica appoggiati a un tavolo.<sup>3</sup> I blocchi possono essere impilati ma solo un blocco può stare direttamente sopra un altro. Un braccio robotico può prendere un blocco e spostarlo in un'altra posizione, sul tavolo oppure su un altro blocco. Il braccio può tenere un solo blocco alla volta, per cui non può afferrarne uno che ne ha sopra un altro. Lo scopo sarà sempre quello di costruire una o più pile di blocchi, descritte specificando esattamente quali blocchi si trovano sopra a quali altri. Un obiettivo, per esempio, potrebbe essere quello di avere il blocco  $A$  su quello  $B$  e il blocco  $C$  su quello  $D$ .

Useremo  $\text{On}(b, x)$  per indicare che il blocco  $b$  si trova su  $x$ , dove  $x$  può essere un altro blocco o il tavolo. L'azione di muovere il blocco  $b$  da sopra  $x$  a sopra  $y$  si scriverà  $\text{Move}(b, x, y)$ . Ora, una delle precondizioni per muovere  $b$  è che non ci sia un blocco su di esso. Nella logica del primo ordine scriveremmo  $\neg\exists x \text{ On}(x, b)$  o, alternativamente,  $\forall x \neg\text{On}(x, b)$ . In ADL queste potrebbero essere espresse come precondizioni. Possiamo rimanere nel linguaggio STRIPS, comunque, introducendo il nuovo predicato  $\text{Libero}(x)$ , che è vero quando non c'è nulla sopra  $x$ .

L'azione  $\text{Move}$  sposta il blocco  $b$  da  $x$  a  $y$  se sia  $b$  che  $y$  sono liberi. Alla fine della mossa  $x$  sarà libero, ma  $y$  no. Una descrizione formale di  $\text{Move}$  in STRIPS è

*Azione*( $\text{Move}(b, x, y)$ ,

*PRECOND:*  $\text{On}(b, x) \wedge \text{Libero}(b) \wedge \text{Libero}(y)$ ,

*EFFETTO:*  $\text{On}(b, y) \wedge \text{Libero}(x) \wedge \neg\text{On}(b, x) \wedge \neg\text{Libero}(y)$ ).

---

<sup>3</sup> Il mondo dei blocchi utilizzato nella ricerca sulla pianificazione è molto più semplice della versione di SHRDLU, mostrata a pag. 29.

---

*Init(Posizione(Bucata, Asse)  $\wedge$  Posizione(Scorta, Bagagliaio))*

*Obiettivo(Posizione(Scorta, Asse))*

*Azione(Rimuovi(Scorta, Bagagliaio))*

PRECOND: *Posizione(Scorta, Bagagliaio)*

EFFETTO :  $\neg$ *Posizione(Scorta, Bagagliaio)  $\wedge$  Posizione(Scorta, Pavimento)*)

*Azione(Rimuovi(Bucata, Asse))*

PRECOND: *Posizione(Bucata, Asse)*

EFFETTO:  $\neg$ *Posizione(Bucata, Asse)  $\wedge$  Posizione(Bucata, Pavimento)*)

*Azione(Monta(Scorta, Asse))*

PRECOND: *Posizione(Scorta, Pavimento)  $\wedge$   $\neg$ Posizione(Bucata, Asse)*

EFFETTO:  $\neg$ *Posizione(Scorta, Pavimento)  $\wedge$  Posizione(Scorta, Asse)*)

*Azione(AbbandonaDiNotte,*

PRECOND:

EFFETTO:  $\neg$ *Posizione(Scorta, Pavimento)  $\wedge$   $\neg$ Posizione(Scorta, Asse)  $\wedge$   $\neg$ Posizione(Scorta, Bagagliaio)*  
 $\wedge$   $\neg$ *Posizione(Bucata, Pavimento)  $\wedge$   $\neg$ Posizione(Bucata, Asse)*)

---

**Figura 11.3** Il semplice problema della ruota bucata.

Sfortunatamente, quest'azione non gestisce propriamente *Libero* quando  $x$  o  $y$  è il tavolo. Quando  $x = \text{Tavolo}$ , l'azione ha come effetto *Libero(Tavolo)*, ma il tavolo non dovrebbe svuotarsi; quando invece  $y = \text{Tavolo}$  risulta una precondizione *Libero(Tavolo)*, ma il tavolo non dev'essere libero per potervi spostare un blocco. Per correggere quest'errore occorre fare due cose. Per prima cosa introduciamo un'altra azione per spostare un blocco  $b$  da  $x$  al tavolo:

*Azione(MoveSulTavolo(b, x),*

PRECOND: *On(b, x)  $\wedge$  Libero(b)),*

EFFETTO: *On(b, Tavolo)  $\wedge$  Libero(x)  $\wedge$   $\neg$ On(b, x)) .*

In secondo luogo, daremo a *Libero(b)* l'interpretazione “c'è uno spazio libero in cima a  $b$  abbastanza ampio da appoggiarvi un blocco”. Sotto quest'interpretazione, *Libero(Tavolo)* sarà sempre vero. L'unico problema è che nulla impedisce al pianificatore di usare *Move(b, x, Tavolo)* al posto di *MoveSulTavolo(b, x)*. Possiamo tollerare questo problema: infatti porterà a uno spazio di ricerca più grande del necessario, ma non sarà la causa di risposte scorrette. Alternativamente, potremmo introdurre il predicato *Blocco* e aggiungere *Blocco(b)  $\wedge$  Blocco(y)* alla precondizione di *Move*.

*Init*(*On*(*A*, *Tavolo*)  $\wedge$  *On*(*B*, *Tavolo*)  $\wedge$  *On*(*C*, *Tavolo*)

$\wedge$  *Blocco*(*A*)  $\wedge$  *Blocco*(*B*)  $\wedge$  *Blocco*(*C*)

$\wedge$  *Libero*(*A*)  $\wedge$  *Libero*(*B*)  $\wedge$  *Libero*(*C*)

*Obiettivo*(*On*(*A*, *B*)  $\wedge$  *On*(*B*, *C*))

*Azione*(*Move*(*b*, *x*, *y*),

PRECOND: *On*(*b*, *x*)  $\wedge$  *Libero*(*b*)  $\wedge$  *Libero*(*y*)  $\wedge$  *Blocco*(*b*)  $\wedge$  (*b*  $\neq$  *x*)  $\wedge$  (*b*  $\neq$  *y*)  $\wedge$  (*x*  $\neq$  *y*)

EFFETTO: *On*(*b*, *y*)  $\wedge$  *Libero*(*x*)  $\wedge$   $\neg$ *On*(*b*, *x*)  $\wedge$   $\neg$ *Libero*(*y*))

*Azione*(*MoveSulTavolo*(*b*, *x*),

PRECOND: *On*(*b*, *x*)  $\wedge$  *Libero*(*b*)  $\wedge$  *Blocco*(*b*)  $\wedge$  (*b*  $\neq$  *x*)

EFFETTO: *On*(*b*, *Tavolo*)  $\wedge$  *Libero*(*x*)  $\wedge$   $\neg$ *On*(*b*, *x*))

**Figura 11.4** Un problema di pianificazione nel mondo dei blocchi: la costruzione di una torre di tre blocchi. Una soluzione è costituita dalla sequenza [*Move*(*B*, *Tavolo*, *C*), *Move*(*A*, *Tavolo*, *B*)].

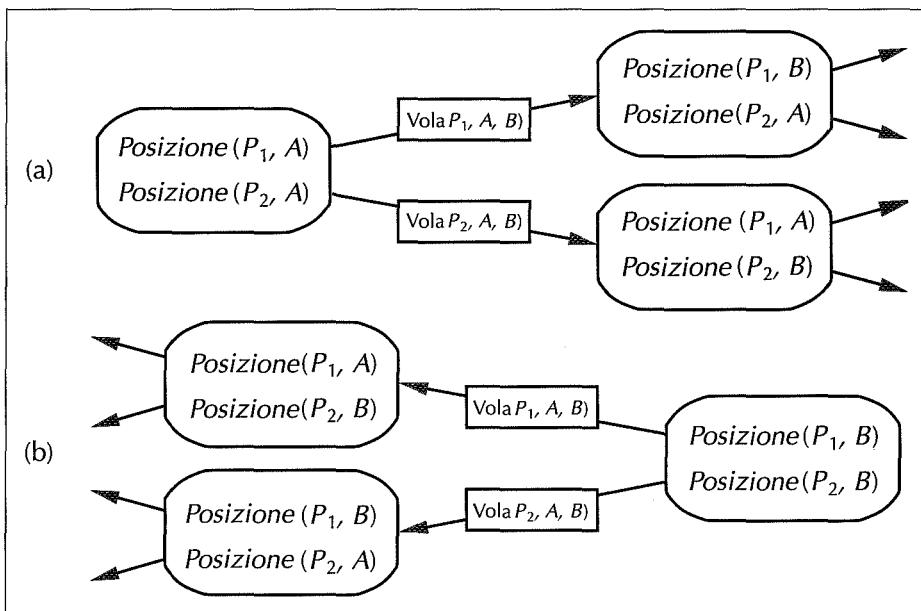
Rimane ancora il problema della presenza di azioni spurie come *Move*(*B*, *C*, *C*), che dovrebbe essere un'operazione nulla, ma ha effetti contraddittori. Di solito problemi come questo si ignorano, perché raramente causano la produzione di piani errati. L'approccio corretto è aggiungere precondizioni di disuguaglianza come quelle mostrate nella Figura 11.4.

## 11.2 Pianificazione con ricerca nello spazio degli stati

Passiamo ora a considerare gli algoritmi di pianificazione; l'approccio più semplice è usare la ricerca nello spazio degli stati. Dato che la descrizione delle azioni in un problema di pianificazione specifica sia le precondizioni che gli effetti, è possibile svolgere la ricerca in entrambe le direzioni: in avanti dallo stato iniziale o all'indietro dall'obiettivo, come si vede nella Figura 11.5. È anche possibile sfruttare le rappresentazioni esplicite delle azioni e degli obiettivi per derivare automaticamente euristiche efficaci.

### Ricerca in avanti nello spazio degli stati

La pianificazione con ricerca in avanti nello spazio degli stati è simile all'approccio alla risoluzione dei problemi che abbiamo visto nel Capitolo 3. Talvolta viene chiamata **pianificazione di progressione**, a causa della direzione del movimento. Si comincia dallo stato iniziale del problema e si considerano sequenze diverse di azioni finché non se ne trova una che raggiunge uno stato obiettivo. La formula-



**Figura 11.5** Due approcci per la ricerca di un piano. (a) La ricerca in avanti (progressione) nello spazio degli stati, che parte dallo stato iniziale e usa le azioni del problema per muoversi verso uno stato obiettivo. (b) La ricerca all'indietro (regressione) nello spazio degli stati: una ricerca negli stati-credenza (v. pag. 112) che parte dallo stato (o dagli stati) obiettivo e usa l'inverso delle azioni per muoversi all'indietro verso lo stato iniziale.

zione dei problemi di pianificazione come problemi di ricerca nello spazio degli stati è la seguente.

- ◆ Lo **stato iniziale** della ricerca è quello iniziale del problema di pianificazione. In generale, ogni stato sarà composto da un insieme di letterali positivi ground; tutti i letterali che non vi compaiono sono considerati falsi.
- ◆ Le **azioni applicabili** a uno stato sono tutte quelle le cui precondizioni sono soddisfatte. Lo stato successore che risulta da un'azione è generato aggiungendo i letterali positivi dell'effetto e cancellando quelli negativi (nel caso del primo ordine, dovremo applicare l'unificatore dalle precondizioni ai letterali dell'effetto). Notate che una singola funzione successore si applica a tutti i problemi di pianificazione: questa è una conseguenza dell'adozione di una rappresentazione esplicita delle azioni.
- ◆ Il **test obiettivo** controlla se lo stato corrente soddisfa l'obiettivo del problema di pianificazione.
- ◆ Il **costo di passo** di ogni azione tipicamente è 1. Sebbene sia facile permettere che ad azioni differenti corrispondano costi diversi, ciò viene fatto raramente dai pianificatori STRIPS.

Ricordate che, in assenza di simboli di funzione, lo spazio degli stati di un problema di pianificazione è finito: ne consegue che ogni algoritmo completo di ricerca su grafo, come A\*, sarà anche un algoritmo di pianificazione completo.

Dagli albori della ricerca sulla pianificazione (intorno al 1961) fino a tempi molto recenti (circa 1998) si è sempre pensato che la ricerca in avanti nello spazio degli stati fosse troppo inefficiente per essere usata nella pratica. Non è difficile trovare ragioni per pensarla così: è sufficiente rileggere l'inizio del Paragrafo 11.1. Prima di tutto, la ricerca in avanti non risolve il problema delle azioni irrilevanti, cosicché in ogni stato si devono considerare tutte le azioni applicabili. In secondo luogo, in mancanza di una buona euristica l'approccio rallenta molto velocemente al crescere delle dimensioni del problema. Considerate un problema di trasporto aereo con 10 aeroporti, 5 aerei per aeroporto e 20 pezzi di merci. L'obiettivo è spostare tutte le merci dall'aeroporto *A* a quello *B*. La soluzione è semplice: si possono caricare tutti i 20 pezzi su uno degli aerei in *A*, volare in *B* e scaricare tutto. Trovare questa soluzione può comunque risultare difficile, perché il fattore di ramificazione è enorme: ognuno dei 50 aerei può volare in 9 altri aeroporti, e ognuno dei 20 pacchetti può essere scaricato (se si trova su un aereo) o caricato in uno qualsiasi degli aerei presenti (se si trova in un aeroporto). In media, le azioni possibili sono circa 1000; questo significa che la profondità dell'albero di ricerca fino alla soluzione ovvia avrà circa 1000<sup>41</sup> nodi. È chiaro che sarà necessaria un'euristica molto accurata per far sì che un tipo di ricerca come questo risulti efficiente. Discuteremo alcune euristiche dopo aver considerato la ricerca all'indietro.

## Ricerca all'indietro nello spazio degli stati

Abbiamo descritto brevemente la ricerca all'indietro nello spazio degli stati nel Capitolo 3, come parte della ricerca bidirezionale. Avevamo già notato allora che la ricerca all'indietro può essere difficile da implementare quando gli stati obiettivo non sono enumerati esplicitamente, ma descritti mediante un insieme di costanti. In particolare, non sempre è evidente come si può generare una descrizione dei possibili predecessori di un insieme di stati obiettivo. Come vedremo, la rappresentazione STRIPS rende questo compito molto facile, perché gli insiemi di stati possono essere descritti specificando semplicemente quali letterali devono essere veri.

Il vantaggio principale della ricerca all'indietro è che permette di considerare solo le **azioni rilevanti**. Un'azione è rilevante per un obiettivo congiuntivo se permette di ottenere uno dei congiunti. Ad esempio, l'obiettivo nel nostro problema di trasporto aereo con 10 aeroporti è di avere 20 pezzi di merci all'aeroporto *B* o, più precisamente,

$$\text{Posizione}(C_1, B) \wedge \text{Posizione}(C_2, B) \wedge \dots \wedge \text{Posizione}(C_{20}, B).$$

Ora considerate il congiunto  $\text{Posizione}(C_1, B)$ . Lavorando a ritroso, possiamo cercare le azioni che hanno quest'effetto. Ce n'è solo una:  $\text{Scarica}(C_1, p, B)$ , dove l'aereo *p* non è specificato.

Notate che ci possono essere anche molte azioni *irrilevanti* che portano a uno stato obiettivo. Ad esempio, possiamo far volare un aereo vuoto da *JFK* a *SFO*; quest'azione conduce a uno stato obiettivo partendo da un predecessore in cui l'aereo si trovava in *JFK* e tutti i congiunti dell'obiettivo erano già soddisfatti. Una ricerca all'indietro che permette azioni irrilevanti sarà ancora completa, ma molto meno efficiente. Se una soluzione esiste, sarà comunque possibile trovarla con una ricerca all'indietro che permette solo azioni rilevanti. Restringere il processo a queste ultime significa che la ricerca all'indietro avrà spesso un fattore di ramificazione molto inferiore di quella in avanti. Ad esempio, il nostro problema di trasporto merci ha circa 1000 azioni in avanti dallo stato iniziale, ma solo 20 che procedono all'indietro dall'obiettivo.

La ricerca all'indietro spesso viene chiamata **pianificazione di regressione**. La questione principale è: quali sono gli stati in cui l'applicazione di una data azione conduce a un obiettivo? Quando si calcola la descrizione di questi stati si dice che si fa **regredire** l'obiettivo attraverso le azioni. Per vedere come farlo, considerate ancora l'esempio del trasporto aereo. Abbiamo l'obiettivo

$$\text{Posizione}(C_1, B) \wedge \text{Posizione}(C_2, B) \wedge \dots \wedge \text{Posizione}(C_{20}, B)$$

e l'azione rilevante *Scarica*( $C_1, p, B$ ) che soddisfa il primo congiunto. L'azione funzionerà solo se le sue precondizioni sono soddisfatte, di conseguenza ogni stato predecessore deve includerle: *In*( $C_1, p$ )  $\wedge$  *Posizione*( $p, B$ ). Inoltre, il sotto-obiettivo *Posizione*( $C_1, B$ ) non dovrebbe essere vero nello stato predecessore.<sup>4</sup> La descrizione del predecessore è quindi

$$\text{In}(C_1, p) \wedge \text{Posizione}(p, B) \wedge \text{Posizione}(C_2, B) \wedge \dots \wedge \text{Posizione}(C_{20}, B).$$

Oltre a richiedere che le azioni rendano vero qualche letterale desiderato, occorre richiedere anche che esse *non falsifichino* altri letterali desiderati. Un'azione che soddisfa questa restrizione è chiamata **consistente**: ad esempio, l'azione *Carica*( $C_2, p$ ) non sarebbe consistente con l'obiettivo corrente, perché negherebbe il letterale *Posizione*( $C_2, B$ ).

Date le definizioni di rilevanza e consistenza, possiamo descrivere il processo generale di costruzione dei predecessori per la ricerca all'indietro. Data una descrizione di obiettivo  $G$ , sia  $A$  un'azione rilevante e consistente. Il predecessore corrispondente si ottiene come segue:

- ♦ gli effetti positivi di  $A$  che appaiono in  $G$  sono cancellati;
- ♦ si aggiunge ogni letterale nella precondizione di  $A$ , a meno che non sia già presente.

pianificazione di regressione

consistente

<sup>4</sup> Se il sotto-obiettivo fosse vero nello stato predecessore, l'azione porterebbe ancora a uno stato obiettivo. D'altra parte tale azione sarebbe irrilevante, perché non *rende* vero l'obiettivo.

Per eseguire la ricerca si può usare uno qualsiasi degli algoritmi di ricerca standard: il processo termina quando viene generata una descrizione di predecessore soddisfatta dallo stato iniziale del problema di pianificazione. Nel caso del primo ordine, il soddisfacimento potrebbe richiedere una sostituzione delle variabili nella descrizione del predecessore. Ad esempio, la descrizione del predecessore del paragrafo precedente è soddisfatta dallo stato iniziale

$$In(C_1, P_{12}) \wedge Posizione(P_{12}, B) \wedge Posizione(C_2, B) \wedge \dots \wedge Posizione(C_{20}, B)$$

con la sostituzione  $\{p/P_{12}\}$ . La sostituzione dev'essere applicata alle azioni che portano dallo stato all'obiettivo, producendo la soluzione  $[Scarica(C_1, P_{12}, B)]$ .

## Euristiche per la ricerca nello spazio degli stati

Come abbiamo detto, né la ricerca in avanti né quella all'indietro sono efficienti senza una buona funzione euristica. Ricorderete dal Capitolo 4 che una funzione euristica stima la distanza da uno stato all'obiettivo; nella pianificazione STRIPS il costo di un'azione è sempre 1, per cui la distanza è il numero di azioni. L'idea di base è considerare gli effetti delle azioni e gli obiettivi che devono essere raggiunti, e cercare di stimare quante azioni saranno necessarie per ottenerli tutti. Determinare il numero esatto è NP-difficile, ma nella maggior parte dei casi è possibile trovare stime ragionevoli senza troppa fatica. Potrebbe anche essere possibile derivare un'euristica ammissibile, una cioè che non sbaglia mai per eccesso. In questo caso si potrebbe usare la ricerca A\* per trovare soluzioni ottime.

Ci sono due possibili approcci: il primo è derivare un **problema rilassato** dalla specifica del problema (v. Capitolo 4). La soluzione ottima del problema rilassato – che ci auguriamo sia molto facile da risolvere – fornisce un'euristica ammissibile del problema originale. Il secondo approccio è fingere che un algoritmo *divide et impera* puro possa funzionare. Questa prende il nome di ipotesi di **indipendenza dei sotto-obiettivi**: il costo della soluzione di una congiunzione di sotto-obiettivi è approssimato dalla somma dei costi della soluzione di ogni sotto-obiettivo *ottenuta in modo indipendente*. L'ipotesi di indipendenza dei sotto-obiettivi può essere ottimista o pessimista. È ottimista quando ci sono interazioni negative tra i sottopiani per ogni sotto-obiettivo: ad esempio quando un'azione in un piano cancella un obiettivo già ottenuto. È pessimista, e quindi non ammissibile, quando i sottopiani contengono azioni ridondanti: ad esempio, quando due azioni potrebbero essere sostituite da un'azione singola nel piano globale.

Vediamo come si possono derivare problemi di pianificazione rilassati. Dato che sono disponibili rappresentazioni esplicite delle precondizioni e degli effetti, il processo opererà modificando tali rappresentazioni (confrontate quest'approccio con quello dei problemi di ricerca, in cui la funzione successore è una scatola nera). La tecnica più semplice è rilassare il problema *rimuovendo tutte le precondizioni* dalle azioni. Ogni azione risulterà sempre applicabile, e ogni letterale potrà essere otte-

nuto in un passo (sempre che ci sia un'azione applicabile: in caso contrario, l'obiettivo è impossibile da raggiungere). Tutto ciò significa quasi che il numero di passi richiesto per risolvere una congiunzione di obiettivi corrisponde al numero di obiettivi non ancora soddisfatti: abbiamo detto “quasi”, comunque, perché (1) potrebbero esserci due azioni ognuna delle quali cancella il letterale obiettivo raggiunto dall'altra e (2) qualche azione potrebbe raggiungere più obiettivi. Se combiniamo il problema rilassato con l'ipotesi di indipendenza dei sotto-obiettivi, entrambe queste eccezioni svaniscono e l'euristica risultante corrisponde esattamente al numero di obiettivi non soddisfatti.

In molti casi si può ottenere un'euristica più accurata considerando almeno le interazioni positive che sorgono dalle azioni che raggiungono più obiettivi. Prima di tutto rilassiamo ulteriormente il problema *rimuovendo gli effetti negativi* (v. Esercizio 11.6). Quindi contiamo il numero minimo di azioni richieste affinché l'unione degli effetti positivi di tali azioni soddisfi l'obiettivo. Considerate ad esempio

*Obiettivo*( $A \wedge B \wedge C$ )

*Azione*( $X$ , EFFETTO:  $A \wedge P$ )

*Azione*( $Y$ , EFFETTO:  $B \wedge C \wedge Q$ )

*Azione*( $Z$ , EFFETTO:  $B \wedge P \wedge Q$ ) .

La copertura minima dell'obiettivo  $\{A, B, C\}$  è data dalle azioni  $\{X, Y\}$ , per cui l'euristica dell'insieme di copertura restituisce un costo di 2. Questo è un miglioramento rispetto all'ipotesi di indipendenza dei sotto-obiettivi, che fornisce un valore euristico di 3. C'è un piccolo, irritante inconveniente: il problema dell'insieme di copertura è NP-difficile. Tuttavia, è garantito che un semplice algoritmo greedy di copertura restituirà un valore compreso in un fattore  $\log n$  del vero valore minimo, dove  $n$  è il numero di letterali dell'obiettivo, e normalmente nella pratica l'algoritmo greedy funzionerà molto meglio di così. Sfortunatamente, non sarà più garantita l'ammissibilità dell'euristica.

È anche possibile generare insiemi rilassati rimuovendo gli effetti negativi senza eliminare le precondizioni. In altre parole, se un'azione ha l'effetto  $A \wedge \neg B$  nel problema originale, avrà solo l'effetto  $A$  nel problema rilassato. Questo significa che non ci si deve preoccupare delle interazioni negative tra i sottopiani, perché nessuna azione potrà cancellare i risultati ottenuti da un'altra azione. Il costo della soluzione del risultante problema rilassato fornisce quella che viene chiamata euristica della **lista di cancellazione vuota** (*empty-delete-list*). L'euristica è molto accurata, ma per calcolarla è necessario di fatto eseguire un (semplice) algoritmo di pianificazione: in pratica, la ricerca nel problema rilassato è spesso abbastanza veloce da giustificarne l'uso.

Le euristiche descritte possono essere usate sia in progressione che in regressione. Mentre scriviamo, i sistemi più veloci sono i pianificatori in progressione che utilizzano l'euristica della lista di cancellazione vuota. Questo probabilmente

lista di cancellazione  
vuota

cambierà man mano che vengono esplorate nuove euristiche e tecniche di ricerca. Dato che la pianificazione è esponenzialmente difficile<sup>5</sup> nessun algoritmo potrà essere efficiente per tutti i problemi, ma grazie ai metodi euristici descritti in questo capitolo oggi si possono risolvere molti più problemi di importanza pratica di quanto fosse possibile anche solo pochi anni fa.

## 11.3 Pianificazione con ordinamento parziale

Quelle in avanti e all'indietro nello spazio degli stati sono forme particolari di ricerche di pianificazione *con ordinamento totale*, e considerano solo sequenze di azioni strettamente lineari collegate direttamente allo stato iniziale o all'obiettivo. Questo significa che non possono trarre vantaggio dalla scomposizione del problema: invece di lavorare separatamente su ogni sottoproblema, le loro decisioni devono sempre riguardare sequenze di azioni che li coinvolgono tutti. Sarebbe preferibile utilizzare un approccio che tratta indipendentemente sotto-obiettivi diversi, li risolve per mezzo di più sottopiani e infine integra questi ultimi in un piano finale.

Un approccio siffatto ha anche il vantaggio di essere più flessibile sull'ordine con cui *costruisce* il piano: può considerare prima le decisioni "evidenti" o "importanti", senza essere obbligato a seguire un rigido ordine cronologico. Ad esempio, un agente pianificatore che si trovi a Berkeley e desideri andare a Montecarlo potrà per prima cosa cercare un volo da San Francisco a Parigi; una volta ottenuta l'informazione sugli orari di partenza e arrivo potrà porsi il problema dei trasferimenti da e per gli aeroporti.

Durante una ricerca, il principio generale di rimandare una scelta viene chiamata strategia del **minimo impegno**. Non è possibile darne una definizione formale; peraltro è chiaro che qualche grado di impegno è necessario, altrimenti la ricerca non farebbe alcun progresso. Nonostante la sua natura informale, il minimo impegno è un concetto utile per analizzare quando dovrebbero essere prese le decisioni in un problema di ricerca.

Il nostro primo esempio concreto sarà molto più semplice della pianificazione di una vacanza. Considerate il problema di infilarsi un paio di scarpe. Possiamo descriverlo formalmente come problema di pianificazione nel modo che segue:

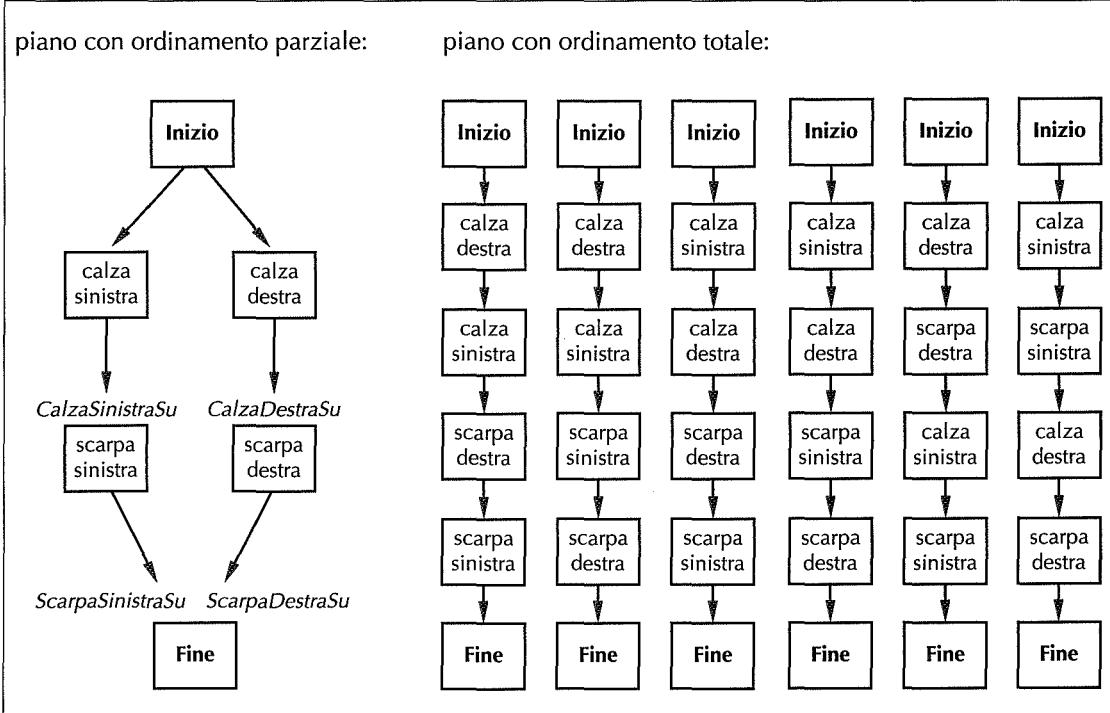
```

Obiettivo(ScarpaDestraSu \wedge ScarpaSinistraSu)
Init()
Azione(ScarpaDestra, PRECOND: CalzaDestraSu, EFFETTO: ScarpaDestraSu)
Azione(CalzaDestra, EFFETTO: CalzaDestraSu)
Azione(ScarpaSinistra, PRECOND: CalzaSinistraSu, EFFETTO: ScarpaSinistraSu)
Azione(CalzaSinistra, EFFETTO: CalzaSinistraSu).

```

minimo impegno

<sup>5</sup> Tecnicamente la pianificazione nello stile di STRIPS è PSPACE-completa, a meno che le azioni non abbiano solo precondizioni positive e un solo letterale di effetto (Bylander, 1994).



**Figura 11.6** Un piano con ordinamento parziale per infilarsi calze e scarpe, e le sei corrispondenti linearizzazioni in piani totalmente ordinati.

Un pianificatore dovrebbe essere in grado di determinare che il primo congiunto dell’obiettivo può essere soddisfatto con la sequenza di due azioni *CalzaDestra* seguita da *ScarpaDestra* e il secondo con la sequenza *CalzaSinistra* seguita da *ScarpaSinistra*. Le due sequenze potranno essere poi combinate nel piano finale. Nel far questo, il pianificatore manipolerà le due sottosequenze in modo indipendente, senza impegnarsi sull’ordinamento delle rispettive azioni. Un algoritmo di pianificazione che può aggiungere due azioni a un piano senza specificare quale delle due è eseguita per prima prende il nome di **pianificatore con ordinamento parziale**. La Figura 11.6 mostra il piano con ordinamento parziale che rappresenta la soluzione al problema delle due scarpe. Notate che la soluzione ha la forma di un *grafo* di azioni, e non di una sequenza. Notate anche la presenza delle azioni “finte” *Inizio* e *Fine*, che marcano appunto l’inizio e la fine del piano. Chiamarle azioni semplifica le cose, perché possiamo dire che ogni passo di un piano è costituito da un’azione. La soluzione con ordinamento parziale corrisponde a sei possibili piani con ordinamento totale; ognuno di essi prende il nome di **linearizzazione** del piano parzialmente ordinato.

pianificatore con  
ordinamento parziale

linearizzazione

La pianificazione con ordinamento parziale può essere implementata come una ricerca nello spazio dei piani parzialmente ordinati (che d'ora in poi, per semplicità, chiameremo solo “piani”). Cominceremo così con un piano vuoto; quindi considereremo diversi modi di raffinarlo finché non saremo riusciti a identificare un piano completo che risolve il problema. In questa ricerca le azioni non sono effettuate nel mondo, ma sui piani stessi: aggiungendo un passo, stabilendo l'ordine tra due azioni e così via.

Per la pianificazione con ordinamento parziale definiremo l'algoritmo POP. Tradizionalmente viene scritto come un programma autonomo, ma noi lo formuliamo invece come un'istanza di un problema di ricerca; questo ci permetterà di focalizzarci sui passi di raffinamento del piano senza preoccuparci del modo in cui l'algoritmo esplora lo spazio. In effetti, una volta formulato il problema di ricerca è possibile applicare diversi metodi di ricerca euristica o non informata.

Ricordate che gli stati del nostro problema di ricerca saranno sempre dei piani, nella maggior parte incompleti. Per evitare confusione con gli stati del mondo, parleremo sempre di piani e non di stati. Ogni piano ha i seguenti quattro componenti: i primi due ne definiscono i passi, gli altri due servono come supporto per determinare le possibili estensioni.

- ◆ Un insieme di **azioni** che costituiscono i passi del piano, prese dall'insieme di azioni del problema originale di pianificazione. Il piano “vuoto” contiene solo le azioni *Inizio* e *Fine*. *Inizio* non ha precondizioni e ha come effetto tutti i letterali dello stato iniziale del problema di pianificazione. *Fine* non ha effetti e ha come precondizioni i letterali obiettivo del problema di pianificazione.
- ◆ Un insieme di **vincoli di ordinamento**. Ogni vincolo ha la forma  $A \prec B$ , che si legge “*A* prima di *B*” e significa che l'azione *A* dev'essere eseguita in qualche punto del tempo prima di *B*, ma non necessariamente immediatamente prima. I vincoli di ordinamento devono costituire un ordinamento parziale proprio. Qualsiasi ciclo, come  $A \prec B$  e  $B \prec A$ , rappresenta una contraddizione: non è quindi possibile aggiungere a un piano vincoli di ordinamento che portano a situazioni cicliche.
- ◆ Un insieme di **collegamenti causali**. Un collegamento causale tra due azioni del piano *A* e *B* si scrive  $A \xrightarrow{p} B$  e si legge “*A* raggiunge *p* per *B*”. Ad esempio, il collegamento causale

$$\text{CalzaDestra} \xrightarrow{\text{CalzaDestraSu}} \text{ScarpaDestra}$$

asserisce che *CalzaDestraSu* è un effetto dell'azione *CalzaDestra* e una precondizione di *ScarpaDestra*. Inoltre asserisce che *CalzaDestraSu* deve rimanere vero dall'istante in cui viene eseguita l'azione *CalzaDestra* a quello dell'azione *ScarpaDestra*. In altre parole, il piano non può essere esteso con

vincoli di ordinamento

collegamenti causali  
raggiunge

l'aggiunta di una nuova azione  $C$  che **confligge** con il collegamento causale. Un'azione  $C$  **confligge** con  $A \xrightarrow{p} B$  se  $C$  ha l'effetto  $\neg p$  e se potrebbe (secondo i vincoli di ordinamento) verificarsi dopo  $A$  e prima di  $B$ . Alcuni autori chiamano i collegamenti causali **intervalli di protezione**, perché  $A \xrightarrow{p} B$  protegge  $p$  dall'essere negato in tutto l'intervallo che va da  $A$  a  $B$ .

confligge

- ♦ Un insieme di **precondizioni aperte**. Una precondizione è aperta se non è soddisfatta da nessuna azione del piano. Il compito di un pianificatore sarà di ridurre l'insieme delle precondizioni aperte a un insieme vuoto, senza introdurre contraddizioni.

precondizioni aperte

Ad esempio, il piano finale della Figura 11.6 ha i seguenti componenti (non mostriamo i vincoli di ordinamento che impongono che ogni azione segua *Inizio* e venga prima di *Fine*):

Azioni:  $\{\text{CalzaDestra}, \text{ScarpaDestra}, \text{CalzaSinistra}, \text{ScarpaSinistra}, \text{Inizio}, \text{Fine}\}$

Ordinamenti:  $\{\text{CalzaDestra} \prec \text{ScarpaDestra}, \text{CalzaSinistra} \prec \text{ScarpaSinistra}\}$

Collegamenti:  $\{\text{CalzaDestra} \xrightarrow{\text{CalzaDestraSu}} \text{ScarpaDestra},$   
 $\text{CalzaSinistra} \xrightarrow{\text{CalzaSinistraSu}} \text{ScarpaSinistra},$   
 $\text{ScarpaDestra} \xrightarrow{\text{ScarpaDestraSu}} \text{Fine},$   
 $\text{ScarpaSinistra} \xrightarrow{\text{ScarpaSinistraSu}} \text{Fine}\}$

Precondizioni aperte:  $\{\}$ .

Definiamo **piano consistente** un piano che non contiene cicli nei vincoli di ordinamento né conflitti con i collegamenti causali. Un piano consistente senza nessuna precondizione aperta è una **soluzione**. *Ogni linearizzazione di una soluzione con ordinamento parziale è una soluzione ordinata la cui esecuzione a partire dallo stato iniziale raggiungerà uno stato obiettivo*. Questo significa che possiamo estendere il concetto di “esecuzione di un piano” anche ai piani parzialmente ordinati. Un piano parzialmente ordinato viene eseguito scegliendo ripetutamente *una qualsiasi* delle possibili azioni successive. Come vedremo nel Capitolo 12, questa flessibilità durante l'esecuzione del piano risulta molto utile quando il mondo si rifiuta di cooperare. Un ordinamento flessibile facilita anche l'integrazione di più piani in un singolo piano globale, perché ognuno dei piani più piccoli può riordinare le proprie azioni per evitare conflitti.

piano consistente

Ora possiamo formulare il problema di ricerca risolto da POP. Cominceremo con una formulazione adatta ai problemi di pianificazione proposizionali, rimanendo a dopo le complicazioni del primo ordine. Come sempre la definizione include lo stato iniziale, le azioni e il test obiettivo.

- ♦ Il piano iniziale contiene *Inizio* e *Fine*, il vincolo di ordinamento  $\text{Inizio} \prec \text{Fine}$  e nessun collegamento causale, e ha tutte le precondizioni di *Fine* come precondizioni aperte.



- ◆ La funzione successore sceglie arbitrariamente una precondizione aperta  $p$  di un'azione  $B$  e genera un piano successore per ogni possibile modo consistente di scegliere un'azione  $A$  che raggiunge  $p$ . La consistenza è assicurata come segue.
  1. Il collegamento causale  $A \xrightarrow{p} B$  e il vincolo di ordinamento  $A \prec B$  sono aggiunti al piano. L'azione  $A$  può esistere già nel piano o essere nuova. Se è nuova, va aggiunta al piano con i due vincoli  $Inizio \prec A$  e  $A \prec Fine$ .
  2. Si risolvono i conflitti tra il nuovo collegamento causale e tutte le azioni esistenti e tra l'azione  $A$  (se è nuova) e tutti i collegamenti causalì esistenti. Un conflitto tra  $A \xrightarrow{p} B$  e  $C$  è risolto obbligando  $C$  ad avvenire all'esterno dell'intervallo di protezione, mediante l'aggiunta di  $B \prec C$  o  $C \prec A$ . Se l'uno, l'altro o entrambi hanno come risultato un piano consistente, si aggiungono i rispettivi stati successore.
- ◆ Il test obiettivo verifica se un piano è una soluzione del problema originale di pianificazione. Dato che i piani generati sono tutti consistenti, il test deve solo controllare se ci sono ancora precondizioni aperte.

Ricordate che le azioni considerate dagli algoritmi di ricerca così formulati sono passi di raffinamento di un piano e non azioni reali del dominio. Il costo di cammino è quindi irrilevante, perché la sola cosa che importa è il costo totale delle azioni reali nel piano. Nonostante questo, è possibile specificare una funzione di costo di cammino che riflette il costo reale del piano: è sufficiente sommare 1 per ogni azione reale aggiunta al piano e 0 per tutti gli altri passi di raffinamento. In questo modo  $g(n)$ , dove  $n$  è un piano, sarà uguale al numero di azioni reali in esso contenute. È anche possibile usare una stima euristica  $h(n)$ .

A prima vista si potrebbe pensare che la funzione successore debba includere i successori di *ogni*  $p$  aperta, e non solo una di esse. Tuttavia questo sarebbe ridondante e inefficiente, per la stessa ragione per cui gli algoritmi di soddisfacimento di vincoli non includono i successori per ogni possibile variabile: l'ordine in cui consideriamo le precondizioni aperte (come quello in cui consideriamo le variabili di un CSP) è commutativo (v. pag. 184). Possiamo quindi scegliere un ordinamento arbitrario e avere ancora un algoritmo completo. Scegliere l'ordinamento giusto può portare a una ricerca più veloce, ma alla fine tutti gli ordinamenti porteranno allo stesso insieme di soluzioni candidate.

## Un esempio di pianificazione con ordinamento parziale

Ora vediamo come POP risolve il problema della ruota bucata del Paragrafo 11.1: la sua descrizione è riportata nella Figura 11.7.

La ricerca comincia dal piano iniziale, che contiene un'azione *Inizio* con effetto *Posizione(Scorta, Bagagliaio)  $\wedge$  Posizione(Bucata, Asse)* e un'azione *Fine* con la sola precondizione *Posizione(Scorta, Asse)*. Per generare i successori dobbiamo prendere una precondizione aperta e scegliere tra le azioni possibili quelle in grado

*Init(Posizione(Bucata, Asse)  $\wedge$  Posizione(Scorta, Bagagliaio))*

*Obiettivo(Posizione(Scorta, Asse))*

*Azione(Rimuovi(Scorta, Bagagliaio))*

PRECOND: *Posizione(Scorta, Bagagliaio)*

EFFETTO :  $\neg$ *Posizione(Scorta, Bagagliaio)  $\wedge$  Posizione(Scorta, Pavimento)*)

*Azione(Rimuovi(Bucata, Asse))*

PRECOND: *Posizione(Bucata, Asse)*

EFFETTO:  $\neg$ *Posizione(Bucata, Asse)  $\wedge$  Posizione(Bucata, Pavimento)*)

*Azione(Monta(Scorta, Asse))*

PRECOND: *Posizione(Scorta, Pavimento)  $\wedge$  \neg Posizione(Bucata, Asse)*

EFFETTO:  $\neg$ *Posizione(Scorta, Pavimento)  $\wedge$  Posizione(Scorta, Asse)*)

*Azione(AbandonaDiNotte,*

PRECOND:

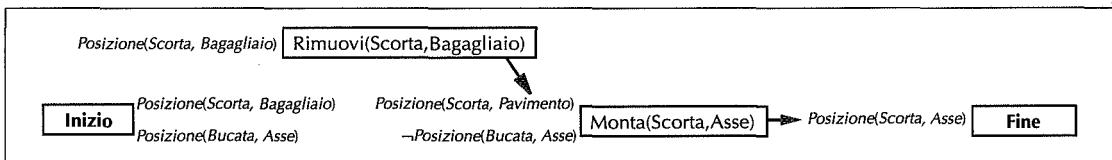
EFFETTO:  $\neg$ *Posizione(Scorta, Pavimento)  $\wedge$  \neg Posizione(Scorta, Asse)  $\wedge$  \neg Posizione(Scorta, Bagagliaio)*

$\wedge$   $\neg$ *Posizione(Bucata, Pavimento)  $\wedge$  \neg Posizione(Bucata, Asse)*)

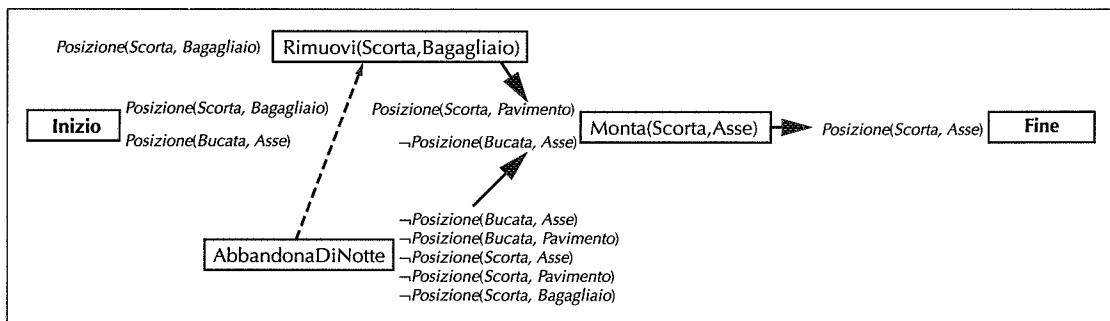
**Figura 11.7** La descrizione del semplice problema della ruota bucata.

di soddisfarla. Per adesso non ci preoccuperemo delle funzioni euristiche che possono aiutarci in queste decisioni: le nostre scelte, quindi, saranno apparentemente arbitrarie. La sequenza di eventi è la seguente.

1. Prendiamo l'unica precondizione aperta di *Fine, Posizione(Scorta, Asse)*. Scegliamo l'unica azione applicabile *Monta(Scorta, Asse)* .
  2. Prendiamo la precondizione *Posizione(Scorta, Pavimento)* di *Monta(Scorta, Asse)*. Scegliamo l'unica azione applicabile per soddisfarla, *Rimuovi(Scorta, Bagagliaio)*.
- Il piano risultante è mostrato nella Figura 11.8.



**Figura 11.8** Il piano incompleto con ordinamento parziale per il problema della ruota bucata, dopo la scelta delle azioni per le prime due precondizioni. I box rappresentano azioni, con le precondizioni a sinistra e gli effetti a destra (questi ultimi tuttavia sono omessi, tranne nel caso dell'azione *Inizio*). Le frecce scure rappresentano collegamenti causali che “proteggono” la proposizione puntata dalla freccia.



**Figura 11.9** Il piano dopo aver scelto *AbbandonaDiNotte* per raggiungere l'obiettivo  $\neg\text{Posizione}(\text{Bucata}, \text{Asse})$ . Per evitare un conflitto con il collegamento causale da *Rimuovi(Scorta, Bagagliaio)* che protegge *Posizione(Scorta, Pavimento)*, *AbbandonaDiNotte* è vincolata in modo che accada prima di *Rimuovi(Scorta, Bagagliaio)*, come evidenzia la freccia tratteggiata.

3. Prendiamo la precondizione  $\neg\text{Posizione}(\text{Bucata}, \text{Asse})$  di *Monta(Scorta, Asse)*. Per fare i bastioni contrari, scegliamo l'azione *AbbandonaDiNotte* invece di *Rimuovi(Bucata, Asse)*. Notate che *AbbandonaDiNotte* ha anche l'effetto  $\neg\text{Posizione}(\text{Scorta}, \text{Pavimento})$ , il che significa che configlia con il collegamento causale

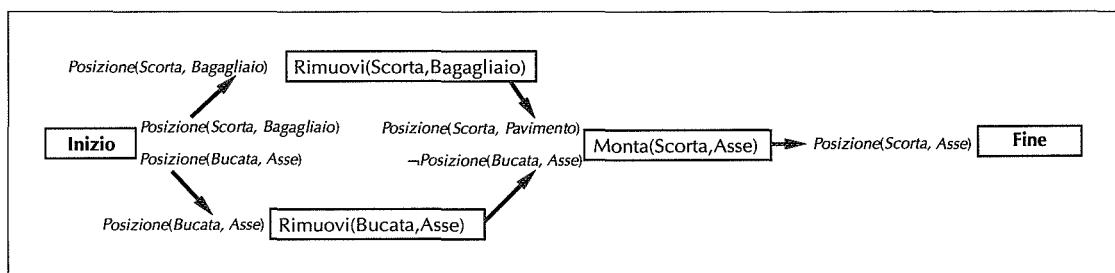
$$\text{Rimuovi}(\text{Scorta}, \text{Bagagliaio}) \xrightarrow{\text{Posizione}(\text{Scorta}, \text{Pavimento})} \text{Monta}(\text{Scorta}, \text{Asse}).$$

Per risolvere il conflitto aggiungiamo un vincolo di ordinamento, spostando *AbbandonaDiNotte* prima di *Rimuovi(Scorta, Bagagliaio)*. Il piano risultante è mostrato nella Figura 11.9 (perché risolve il conflitto, e perché questo è l'unico modo per farlo?).

4. A questo punto la sola precondizione rimasta aperta è *Posizione(Scorta, Bagagliaio)* per l'azione *Rimuovi(Scorta, Bagagliaio)*. L'unica azione che può soddisfarla è *Inizio*, ma il collegamento causale da *Inizio* a *Rimuovi(Scorta, Bagagliaio)* è in conflitto con l'effetto  $\neg\text{Posizione}(\text{Scorta}, \text{Bagagliaio})$  di *AbbandonaDiNotte*. Questa volta non c'è modo di risolvere il conflitto con *AbbandonaDiNotte*: spostarlo prima di *Inizio* è impossibile, e non possiamo neppure spostarlo dopo *Rimuovi(Scorta, Bagagliaio)* perché c'è già un vincolo che impone che venga prima. Siamo quindi costretti a tornare sui nostri passi, eliminare l'azione *Rimuovi(Scorta, Bagagliaio)* e gli ultimi due collegamenti causali e ritornare nello stato della Figura 11.8. In sostanza, il pianificatore ha dimostrato che l'azione *AbbandonaDiNotte* non è un buon modo di cambiare la gomma bucata.

5. Consideriamo ancora la precondizione  $\neg\text{Posizione}(\text{Bucata}, \text{Asse})$  di *Monta(Scorta, Asse)*. Questa volta scegliamo *Rimuovi(Bucata, Asse)*.

6. Ancora una volta prendiamo la precondizione *Posizione(Scorta, Bagagliaio)* di *Rimuovi(Scorta, Bagagliaio)* e per soddisfarla scegliamo *Inizio*. Questa volta non ci sono conflitti.



**Figura 11.10** La soluzione finale del problema della ruota bucata. Notate che *Rimuovi(Scorta, Bagagliaio)* e *Rimuovi(Bucata, Asse)* possono essere eseguite in qualsiasi ordine, finché sono complete prima di *Monta(Scorta, Asse)*.

7. Prendiamo la precondizione *Posizione(Bucata, Asse)* di *Rimuovi(Bucata, Asse)* e per soddisfarla scegliamo ancora *Inizio*. Questo ci fornisce infine il piano completo e consistente – in altre parole, la soluzione – rappresentata nella Figura 11.10.

Benché quest'esempio sia molto semplice, illustra bene alcuni dei punti forti della pianificazione con ordinamento parziale. Prima di tutto, i collegamenti causali portano a una potatura anticipata delle porzioni dello spazio di ricerca che, per colpa di conflitti irrisolvibili, non contengono soluzioni. In secondo luogo, notate che la soluzione della Figura 11.10 è un piano con ordinamento parziale. In questo caso il vantaggio è piccolo, perché ci sono solo due possibili linearizzazioni; nonostante ciò un agente potrebbe avvantaggiarsi anche di una piccola dose di flessibilità: ad esempio, se la ruota dev'essere cambiata nel bel mezzo del traffico.

Quest'esempio mette anche in evidenza alcuni possibili miglioramenti. Ad esempio, c'è del lavoro duplicato: *Inizio* è collegato a *Rimuovi(Scorta, Bagagliaio)* prima che il conflitto causi il ritorno della ricerca sui propri passi. A questo punto il processo di backtracking scioglie il collegamento, anche se non è coinvolto nel conflitto. Quando la ricerca riprende, lo stesso collegamento dev'essere quindi nuovamente ripristinato. Questo comportamento è tipico del backtracking cronologico e può essere mitigato da un backtracking diretto dalle dipendenze.

## Pianificazione con ordinamento parziale con variabili libere

In questo paragrafo consideriamo le complicazioni che possono sorgere quando POP è usato con rappresentazioni delle azioni del primo ordine che includono variabili. Supponiamo di avere un problema nel mondo dei blocchi (v. Figura 11.4) con la precondizione aperta *On(A, B)* e l'azione

*Azione(Move(b, x, y))*,

PRECOND: *On(b, x) ∧ Libero(b) ∧ Libero(y)*,

EFFETTO: *On(b, y) ∧ Libero(x) ∧ ¬On(b, x) ∧ ¬Libero(y)* .

Quest'azione soddisfa  $On(A, B)$  perché l'effetto  $On(b, y)$  unifica con  $On(A, B)$  con la sostituzione  $\{b/A, y/B\}$ . Applichiamo quindi questa sostituzione all'azione, ottenendo

*Azione*( $Move(A, x, B)$ ) ,

PRECOND:  $On(A, x) \wedge Libero(A) \wedge Libero(B)$ ,

EFFETTO:  $On(A, B) \wedge Libero(x) \wedge \neg On(A, x) \wedge \neg Libero(B)$  .

La variabile  $x$  rimane libera. Questo significa che l'azione dice di muovere il blocco  $A$  prendendolo *da qualche parte*, ma senza specificare dove. Questo è un altro esempio del principio del minimo impegno: possiamo rimandare la scelta fino a quando un altro passo del piano la compie per noi. Ad esempio, supponiamo di avere nello stato iniziale  $On(A, D)$ . Allora l'azione *Inizio* può essere usata per ottenere  $On(A, x)$ , legando  $x$  a  $D$ . La strategia di aspettare nuove informazioni prima di scegliere  $x$  è spesso più efficiente di provare ogni valore possibile e tornare indietro con il backtracking ogni volta che si fallisce.

La presenza di variabili nelle precondizioni e nelle azioni complica il processo di rilevazione e di soluzione dei conflitti. Ad esempio, quando aggiungiamo al piano  $Move(A, x, B)$ , abbiamo bisogno del collegamento causale

$Move(A, x, B) \xrightarrow{On(A, B)} Fine$  .

Se c'è un'altra azione  $M_2$  con effetto  $\neg On(A, z)$ , allora  $M_2$  configge solo se  $z$  vale  $B$ . Per considerare questa possibilità occorre estendere la rappresentazione dei piani per includere un insieme di vincoli di disuguaglianza della forma  $z \neq X$ , dove  $z$  è una variabile e  $X$  è un'altra variabile o un simbolo di costante. In questo caso risolveremmo il conflitto aggiungendo  $z \neq B$ , che significa che le future estensioni del piano possono istanziare  $z$  assegnandogli qualsiasi valore, purché sia diverso da  $B$ . Ogni volta che applichiamo una sostituzione a un piano, occorre controllare che le disuguaglianze non contraddicono la sostituzione. Una sostituzione che include  $x/y$ , ad esempio, configge con il vincolo di disuguaglianza  $x \neq y$ . Tali conflitti non possono essere risolti, per cui il pianificatore dovrà tornare sui propri passi.

Nel Paragrafo 12.6 vedremo un esempio più esteso di pianificazione POP con variabili nel mondo dei blocchi.

## Euristiche per la pianificazione con ordinamento parziale

In confronto alla pianificazione con ordinamento totale, quella con ordinamento parziale ha il chiaro vantaggio di poter scomporre i problemi in sottoproblemi. Tuttavia c'è anche uno svantaggio: dato che non rappresenta direttamente gli stati, è più difficile stimare quanto un piano sia vicino al raggiungimento di un obiettivo. Oggi come oggi, i ricercatori hanno una comprensione meno chiara di come si possono calcolare euristiche accurate per la pianificazione con ordinamento parziale rispetto al caso totalmente ordinato.

L'euristica più banale consiste nel contare il numero di precondizioni aperte distinte, e può essere già migliorata sottraendo il numero di precondizioni aperte che corrispondono ai letterali dello stato *Inizio*. Come nel caso con ordinamento totale, questo sovrasta il costo quando ci sono azioni che raggiungono più obiettivi e lo sottostima quando ci sono interazioni negative tra i passi del piano. Nel prossimo paragrafo presenteremo un approccio che ci permette di ottenere euristiche molto più accurate da un problema rilassato.

La funzione euristica serve a scegliere quale piano raffinare. Una volta fatta questa scelta, l'algoritmo genera i successori in base alla selezione di una singola precondizione aperta su cui operare. Come nel caso della variabile negli algoritmi di soddisfacimento di vincoli, questa scelta ha un grande impatto sull'efficienza. L'euristica della **variabile più vincolata** può essere adattata dai CSP agli algoritmi di pianificazione, e sembra funzionare bene. L'idea è di scegliere la precondizione aperta che può essere soddisfatta nel *minor* numero di modi. Per quest'euristica ci sono due casi speciali: prima di tutto, se una precondizione aperta non può essere soddisfatta da nessuna azione, l'euristica la selezionerà; questa è un'ottima idea perché una determinazione anticipata dell'impossibilità di trovare una soluzione può farci risparmiare un sacco di lavoro. In secondo luogo, se una precondizione può essere soddisfatta in un solo modo è meglio selezionare proprio quella, dato che la decisione è obbligata e ci potrà fornire vincoli aggiuntivi per le scelte ancora da compiere. Benché il calcolo completo del numero di modi in cui si possa soddisfare ogni precondizione aperta sia costoso, e non sempre valga la pena di svolgerlo, gli esperimenti dimostrano che la gestione dei due casi speciali può causare notevoli aumenti di efficienza.

## 11.4 Grafi di pianificazione

Tutte le euristiche per la pianificazione suggerite sin qui possono soffrire di imprecisioni. Questo paragrafo mostra come usare una struttura dati speciale, chiamata **grafo di pianificazione**, per ottenere stime euristiche migliori che potranno essere applicate a tutte le tecniche di ricerca che abbiamo considerato. Alternativamente sarà possibile estrarre una soluzione direttamente dal grafo di pianificazione, usando un algoritmo specializzato come GRAPHPLAN.

Un grafo di pianificazione consiste in una sequenza di **livelli** che corrispondono ai passi temporali nel piano, in cui il livello 0 è lo stato iniziale. Ogni livello contiene un insieme di letterali e un insieme di azioni. Approssimativamente, i letterali sono tutti quelli che *potrebbero* essere veri in quel passo temporale, a seconda delle azioni eseguite nei passi precedenti. Sempre approssimativamente, le azioni sono tutte quelle le cui precondizioni *potrebbero* essere soddisfatte in quel passo. Abbiamo detto "approssimativamente" perché il grafo di pianificazione memoriz-

grafo di pianificazione

livelli

za solo un insieme ristretto delle possibili interazioni negative tra le azioni; di conseguenza potrebbe essere troppo ottimista circa il numero minimo di passi temporali richiesti affinché un letterale diventi vero. Nonostante questo, il numero di passi sul grafo di pianificazione fornisce una buona stima di quanto sia difficile soddisfare un dato letterale dallo stato iniziale. La cosa più importante è che la definizione del grafo rende possibile la sua costruzione in modo molto efficiente.

I grafi di pianificazione sono applicabili solo a problemi proposizionali, ovvero privi di variabili. Come abbiamo menzionato nel Paragrafo 11.1, sia le rappresentazioni STRIPS che quelle ADL possono essere proposizionalizzate. Nei problemi che comprendono un gran numero di oggetti, questo potrebbe causare una significativa esplosione del numero di schemi di azione. Nonostante questo i grafi di pianificazione si sono dimostrati uno strumento efficace per risolvere problemi difficili.

Illustreremo i grafi facendo ricorso a un semplice esempio (anche perché esempi più complessi darebbero origine a grafi più grandi di questa pagina), che ri-chiama un detto anglosassone analogo alla nostra “botte piena e moglie ubriaca”. La Figura 11.11 mostra un problema e la 11.12 il suo grafo di pianificazione. Cominciamo dal livello di stato  $S_0$ , che rappresenta lo stato iniziale del problema. Dopo di esso viene il livello di azione  $A_0$ , in cui poniamo tutte le azioni le cui precondizioni sono soddisfatte nel livello precedente. Ogni azione è collegata alle sue precondizioni in  $S_0$  e ai suoi effetti in  $S_1$ , cosa che in questo caso ci fa introdurre in  $S_1$  nuovi letterali non presenti in  $S_0$ .

Il grafo di pianificazione, oltre alle azioni, dev'essere in grado di rappresentare anche l'inazione. In altre parole, necessita dell'equivalente degli assiomi di frame del calcolo delle situazioni, che permettono a un letterale di rimanere vero da una situazione all'altra se nessuna azione lo modifica. In un grafo di pianificazione per far questo si usa un insieme di **azioni di persistenza**. Per ogni letterale positivo e negativo  $C$ , aggiungiamo al problema un'azione di persistenza con precondizione  $C$  ed effetto  $C$ . La Figura 11.12 mostra una sola azione “reale” in  $A_0$ , *Mangia(Torta)*, e due azioni di persistenza disegnate come piccoli quadratini bianchi.

azioni di persistenza

*Init(Ha(Torta))*

*Obiettivo(Ha(Torta)  $\wedge$  Mangiata(Torta))*

*Azione(Mangiata(Torta))*

PRECOND: *Ha(Torta)*

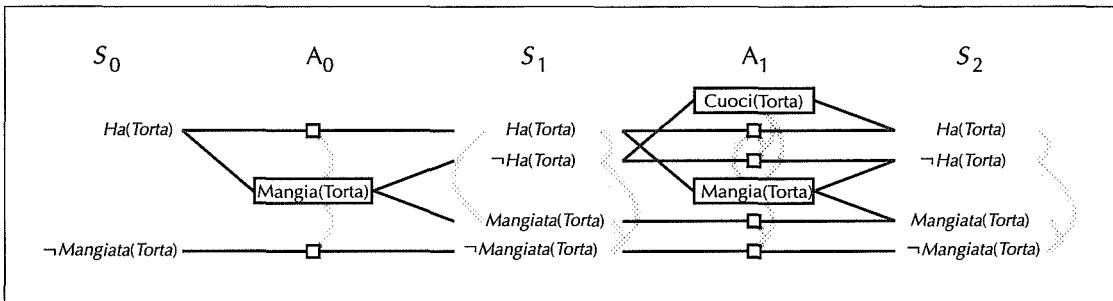
EFFETTO:  $\neg Ha(Torta) \wedge Mangiata(Torta)$

*Azione(Cuoci(Torta))*

PRECOND:  $\neg Ha(Torta)$

EFFETTO: *Ha(Torta)*

Figura 11.11 Il problema “ha la torta e l'ha anche mangiata”.



**Figura 11.12** Il grafo di pianificazione per il problema “ha la torta e l’ha anche mangiata” fino al livello  $S_2$ . I rettangoli indicano azioni (i piccoli quadratini le azioni di persistenza) e le righe dritte indicano precondizioni ed effetti. I collegamenti mutex sono disegnati come linee curve di colore grigio.

Il livello  $A_0$  contiene tutte le azioni che *potrebbero* verificarsi nello stato  $S_0$ , ma riporta anche – e questo è altrettanto importante – i conflitti tra azioni che impedirebbero loro di verificarsi insieme. Le linee grigie nella Figura 11.12 indicano collegamenti di **mutua esclusione** (o **mutex**). Ad esempio, *Mangia(Torta)* è mutuamente esclusivo con la persistenza sia di *Ha(Torta)* che di  $\neg Mangiata(Torta)$ . Vedremo tra breve come si calcolano i collegamenti mutex.

Il livello  $S_1$  contiene tutti i letterali che potrebbero risultare dalla scelta di un qualsiasi sottoinsieme delle azioni in  $A_0$ . Contiene inoltre collegamenti mutex che indicano i letterali che non possono apparire insieme, indipendentemente dalla scelta delle azioni. Ad esempio, *Ha(Torta)* e *Mangiata(Torta)* sono mutex: a seconda della scelta delle azioni in  $A_0$  potremmo avere uno o l’altro, ma non entrambi. In altre parole  $S_1$  rappresenta più stati, proprio come la ricerca di regressione nello spazio degli stati, e i collegamenti mutex sono vincoli che definiscono l’insieme degli stati possibili.

Continuiamo in questo modo alternando tra il livello di stato  $S_i$  e quello di azione  $A_i$ , finché non raggiungiamo un punto in cui due livelli consecutivi sono identici: possiamo dire che il grafo si è **livellato**. Ogni livello successivo sarà identico, per cui è del tutto inutile continuare le espansioni.

Alla fine avremo una struttura in cui ogni livello  $A_i$  contiene tutte le azioni applicabili in  $S_i$  insieme ai vincoli che specificano quali coppie di azioni non possono essere eseguite insieme. Ogni livello  $S_i$  contiene tutti i letterali che potrebbero risultare da qualsiasi possibile scelta di azioni in  $A_{i-1}$  insieme ai vincoli che specificano quali coppie di letterali sono impossibili. È importante notare che il processo di costruzione del grafo di pianificazione *non* richiede alcuna scelta tra le azioni, cosa che implicherebbe una ricerca combinatoria. Il grafo si limita invece a registrare l’impossibilità di alcune scelte per mezzo di collegamenti mutex. La complessità della costruzione del grafo di pianificazione è un polinomio di ordine basso nel numero di azioni e di letterali, laddove lo spazio degli stati è esponenziale nel numero di letterali.

mutua esclusione  
mutex

livellato

Definiamo ora i collegamenti mutex sia per le azioni che per i letterali. In un dato livello si ha una relazione mutex tra due *azioni* se si verifica una qualsiasi delle seguenti tre condizioni.

- ◆ *Effetti inconsistenti*: un'azione nega l'effetto dell'altra. Ad esempio, *Mangia(Torta)* e la persistenza di *Ha(Torta)* hanno effetti inconsistenti, perché sono in disaccordo sull'effetto *Ha(Torta)*.
- ◆ *Interferenze*: uno degli effetti di un'azione è la negazione di una precondizione dell'altra. Ad esempio, *Mangia(Torta)* interferisce con la persistenza di *Ha(Torta)* negando la sua precondizione.
- ◆ *Necessità in competizione*: una delle precondizioni di una delle azioni è mutuamente esclusiva con una precondizione dell'altra. Ad esempio, *Cuoci(Torta)* e *Mangia(Torta)* sono mutex perché competono sul valore della precondizione *Ha(Torta)*.

In un dato livello si ha una relazione mutex tra due *letterali* se uno è la negazione dell'altro oppure se ogni possibile coppia di azioni che potrebbe soddisfare i due letterali è mutuamente esclusiva. Questa condizione prende il nome di *supporto inconsistente*. Ad esempio, *Ha(Torta)* e *Mangiata(Torta)* sono mutex in  $S_1$  perché l'unico modo di ottenere *Ha(Torta)*, l'azione di persistenza, è mutex con l'unico modo di ottenere *Mangiata(Torta)*, e cioè *Mangia(Torta)*. In  $S_2$  i due letterali non sono mutex perché ci sono nuovi modi di soddisfarli, come *Cuoci(Torta)* e la persistenza di *Mangiata(Torta)*, che non sono mutex.

## Grafi di pianificazione per la stima euristica

Un grafo di pianificazione, una volta costruito, è una ricca fonte di informazioni su un problema. Ad esempio, un letterale che non compare nel livello finale del grafo non può essere raggiunto da alcun piano. Questa osservazione può essere sfruttata nella ricerca all'indietro assegnando a ogni stato che contiene un letterale irraggiungibile un costo  $h(n) = \infty$ . In modo analogo, nella pianificazione con ordinamento parziale, ogni piano con una precondizione aperta irraggiungibile ha  $h(n) = \infty$ .

Quest'idea può essere resa più generale: possiamo stimare il costo del raggiungimento di un letterale obiettivo in base al livello in cui appare per la prima volta nel grafo di pianificazione. Chiameremo questo il *costo di livello* dell'obiettivo. Nella Figura 11.12, *Ha(Torta)* ha un costo di livello di 0 e *Mangiata(Torta)* di 1. È facile dimostrare (v. Esercizio 11.9) che queste stime sono ammissibili per i singoli obiettivi. La stima potrebbe non essere molto buona, comunque, perché i grafi di pianificazione permettono più azioni per livello mentre l'euristica conta solo i livelli e non il numero di azioni. Per questa ragione, per il calcolo delle euristiche è pratica comune utilizzare un *grafo di pianificazione seriale*. Un grafo di pianificazione seriale permette l'esecuzione di una sola azione per ogni passo tem-



costo di livello



grafo di pianificazione seriale

porale; per far questo è sufficiente aggiungere collegamenti mutex tra ogni coppia di azioni eccetto quelle di persistenza. Spesso i costi di livello estratti dai grafi seriali sono stime molto affidabili dei costi reali.

Per stimare il costo di una congiunzione di obiettivi ci sono tre semplici approcci. L'euristica **max-level** prende semplicemente il massimo tra i costi di livello di tutti gli obiettivi: è ammissibile, ma non necessariamente accurata. L'euristica della **somma dei livelli**, seguendo l'ipotesi di indipendenza tra sotto-obiettivi, restituisce la somma dei costi di livello degli obiettivi; non è ammissibile, ma funziona bene nella pratica per i problemi molto scomponibili, ed è molto più accurata del numero di obiettivi insoddisfatti che abbiamo considerato nel Paragrafo 11.2. Per il nostro problema, la stima euristica dell'obiettivo congiuntivo  $Ha(Torta) \wedge Mangiata(Torta)$  sarà pari a  $0 + 1 = 1$ , laddove la risposta corretta sarebbe 2. Inoltre, se eliminassimo l'azione  $Cuoci(Torta)$  la stima rimarrebbe pari a 1, ma l'obiettivo congiuntivo sarebbe impossibile. Infine, l'euristica del **livello di insieme** cerca il livello in cui tutti i letterali dell'obiettivo congiuntivo appaiono sul grafo senza che alcuna coppia di essi sia mutuamente esclusiva. Quest'euristica restituisce il valore corretto di 2 nel nostro problema originale e di infinito per il problema senza  $Cuoci(Torta)$ . Il livello di insieme domina l'euristica max-level e funziona benissimo nelle attività in cui c'è molta interazione tra i sottopiani.

Come strumento per generare euristiche accurate, possiamo considerare il grafo di pianificazione come un problema rilassato risolvibile in modo efficiente. Per comprendere la natura del problema rilassato dobbiamo capire cosa significa esattamente che un letterale  $g$  compaia al livello  $S_i$  del grafo. Idealmente, ci piacerebbe che questo rappresenti una garanzia del fatto che esiste un piano con  $i$  livelli di azione che raggiunge  $g$ , e che l'assenza di  $g$  garantisca che tale piano non esiste. Sfortunatamente, ottenere una garanzia simile è difficile quanto risolvere il problema di pianificazione originale. Così, il grafo di pianificazione garantisce effettivamente la seconda parte (se  $g$  non compare, il piano non esiste), ma se invece appare, tutto quello che si può dire è che c'è un piano che *forse* raggiunge  $g$  e non ha "evidenti" difetti. Un difetto è evidente quando può essere rilevato prendendo due azioni o due letterali per volta – in altre parole, considerando le relazioni mutex. Difetti più sottili potrebbero coinvolgere tre, quattro o più azioni, ma l'esperienza ha dimostrato che non vale la pena di spendere lo sforzo computazionale necessario per rilevarli. Questo caso è analogo a quello dei problemi di soddisfacimento di vincoli in cui spesso è una buona idea calcolare la 2-consistenza prima di cercare una soluzione, ma è molto più raro che valga la pena di calcolare la 3-consistenza o quelle superiori (v. Paragrafo 5.2).

max-level

somma dei livelli

livello di insieme

## L'algoritmo GRAPHPLAN

Questo sottoparagrafo mostra come si può estrarre direttamente un piano dal grafo di pianificazione, anziché usarlo solo per derivare un'euristica. L'algoritmo GRAPHPLAN (Figura 11.13) prevede due passi principali, che si alternano all'interno di un ciclo. Prima di tutto verifica se tutti i letterali dell'obiettivo sono presenti nello stato corrente senza collegamenti mutex tra alcuna delle coppie: se è così il grafo corrente *potrebbe* contenere una soluzione, che l'algoritmo cerca di estrarre. In caso contrario, il grafo viene espanso aggiungendo le azioni per il livello corrente e i letterali di stato per quello successivo. Il processo continua finché si trova una soluzione o si appura che non ne esiste alcuna.

Seguiamo il funzionamento di GRAPHPLAN sul problema della ruota bucata del Paragrafo 11.1. La Figura 11.14 riporta l'intero grafo. La prima riga di GRAPHPLAN inizializza il grafo di pianificazione con un grafo a un livello ( $S_0$ ) che consiste nei cinque letterali dello stato iniziale del problema. Il letterale obiettivo *Posizione(Scorta, Asse)* non è presente in  $S_0$ , per cui non è necessario invocare ESTRAI-SOLUZIONE, dato che è sicuro che non ne esista ancora alcuna. L'algoritmo invoca invece ESPANDI-GRAFO, che aggiunge le tre azioni le cui precondizioni esistono al livello  $S_0$  (cioè, tutte le azioni tranne *Monta(Scorta, Asse)*) insieme alle azioni di persistenza per tutti i letterali in  $S_0$ . Gli effetti delle azioni sono aggiunti al livello  $S_1$ . A questo punto ESPANDI-GRAFO cerca eventuali relazioni mutex e le aggiunge al grafo.

*Posizione(Scorta, Asse)* non è presente neppure in  $S_1$ , così ancora una volta ESTRAI-SOLUZIONE non viene invocata. La chiamata a ESPANDI-GRAFO ha come risultato il grafo di pianificazione mostrato nella Figura 11.14. Ora che abbiamo tutte le azioni, vale la pena considerare alcuni esempi di relazioni mutex e le loro cause.

- ◆ *Effetti inconsistenti:* *Rimuovi(Scorta, Bagagliaio)* è mutex con *AbbandonaDiNotte* perché la prima ha come effetto *Posizione(Scorta, Pavimento)*, la seconda la sua negazione.
- ◆ *Interferenze:* *Rimuovi(Bucata, Asse)* è mutex con *AbbandonaDiNotte* perché la prima ha la precondizione *Posizione(Bucata, Asse)*, la seconda ha come effetto la sua negazione.
- ◆ *Necessità in competizione:* *Monta(Scorta, Asse)* è mutex con *Rimuovi(Bucata, Asse)* perché la prima ha come precondizione *Posizione(Bucata, Asse)*, la seconda la sua negazione.
- ◆ *Supporto inconsistente:* *Posizione(Scorta, Asse)* è mutex con *Posizione(Bucata, Asse)* in  $S_2$  perché l'unico modo di ottenere *Posizione(Scorta, Asse)* è attraverso *Monta(Scorta, Asse)*, che è mutex con l'azione di persistenza che rappresenta l'unico modo di ottenere *Posizione(Bucata, Asse)*. Le relazioni di mutua esclusione sono quindi in grado di rilevare il conflitto immediato che sorge dal tentativo di mettere due oggetti nello stesso punto allo stesso tempo.

**function** GRAPHPLAN(*problema*) **returns** una soluzione, o il fallimento

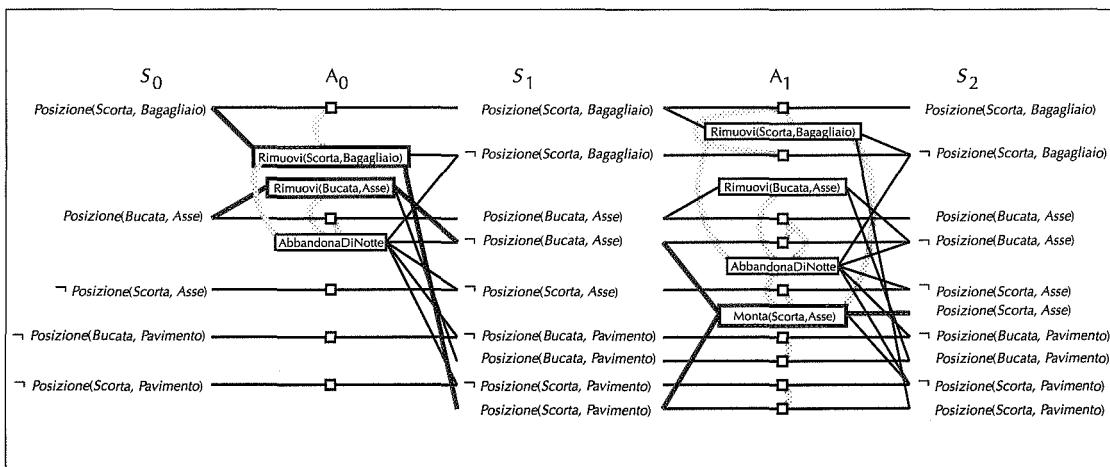
```

grafo ← GRAFO-INIZIALE-PIANIFICAZIONE(problema)
obiettivi ← OBIETTIVI[problema]
loop do
 if obiettivi sono tutti non-mutex nell'ultimo livello di grafo then do
 soluzione ← ESTRAI-SOLUZIONE(grafo, obiettivi, LUNGHEZZA(grafo))
 if soluzione ≠ fallimento then return soluzione
 else if NESSUNA-SOLUZIONE-POSSIBILE(grafo) then return fallimento
 grafo ← ESPANDI-GRAFO(grafo, problema)

```

**Figura 11.13** L'algoritmo GRAPHPLAN alterna un passo di estrazione di una soluzione e un passo di espansione del grafo. ESTRAI-SOLUZIONE verifica se può trovare un piano, partendo dalla fine e procedendo all'indietro. ESPANDI-GRAFO aggiunge le azioni del livello corrente e i letterali di stato di quello successivo.

Questa volta, quando rientriamo nel ciclo tutti i letterali dell'obiettivo sono presenti in  $S_2$ , e nessuno di essi è mutex con nessun altro. Questo significa che potrebbe esistere una soluzione, che ESTRAI-SOLUZIONE tenterà di trovare. In pratica ESTRAI-SOLUZIONE deve risolvere un CSP booleano che ha come variabili le



**Figura 11.14** Il grafo di pianificazione per il problema della ruota bucata dopo l'espansione fino al livello  $S_2$ . I collegamenti mutex sono indicati con linee grigie. Per semplicità abbiamo riportato solo alcuni dei mutex più rappresentativi. La soluzione è indicata da segmenti e rettangoli in grassetto.

azioni in ogni livello; il valore di ogni variabile può essere *dentro* o *fuori* dal piano. Per far questo possiamo usare un algoritmo standard per CSP, oppure possiamo definire ESTRAI-SOLUZIONE come un problema di ricerca in cui ogni stato contiene un puntatore a un livello del grafo di pianificazione e a un insieme di obiettivi non soddisfatti, come segue.

- ◆ Lo stato iniziale è l'ultimo livello del grafo di pianificazione,  $S_n$ , con l'insieme di obiettivi del problema di pianificazione originale.
- ◆ Le azioni disponibili in uno stato al livello  $S_i$  consistono nella selezione di un qualsiasi sottoinsieme privo di conflitti delle azioni in  $A_{i-1}$  i cui effetti soddisfano gli obiettivi dello stato. Lo stato risultante ha livello  $S_{i-1}$  e ha come insieme di obiettivi le precondizioni dell'insieme di azioni prescelto. Con “privi di conflitti” intendiamo dire un insieme di azioni tali che nessuna copia di azioni né di precondizioni è mutex.
- ◆ L'obiettivo è raggiungere uno stato al livello  $S_0$  tale che tutti gli obiettivi siano soddisfatti.
- ◆ Il costo di ogni azione è 1.

Per questo particolare problema, cominciamo in  $S_2$  con l'obiettivo *Posizione(Scorta, Asse)*. Per ottenere l'insieme obiettivo l'unica scelta è *Monta(Scorta, Asse)*. Questo ci porta a uno stato di ricerca in  $S_1$  con gli obiettivi *Posizione(Scorta, Pavimento)* e *¬Posizione(Bucata, Asse)*. Il primo dei due può essere ottenuto solo con *Rimuovi(Scorta, Bagagliaio)*, il secondo con *Rimuovi(Bucata, Asse)* oppure con *AbbandonaDiNotte*. Ma *AbbandonaDiNotte* è mutex con *Rimuovi(Scorta, Bagagliaio)*, per cui l'unica soluzione è scegliere *Rimuovi(Scorta, Bagagliaio)* e *Rimuovi(Bucata, Asse)*. Questo ci porta a uno stato di ricerca  $S_0$  con gli obiettivi *Posizione(Scorta, Bagagliaio)* e *Posizione(Bucata, Asse)*. Entrambi sono presenti nello stato, per cui abbiamo una soluzione: le azioni *Rimuovi(Scorta, Bagagliaio)* e *Rimuovi(Bucata, Asse)* in  $A_0$ , seguita da *Monta(Scorta, Asse)* in  $A_1$ .

Sappiamo che la pianificazione è PSPACE-completa e che la costruzione del grafo di pianificazione richiede un tempo polinomiale, per cui potrebbe darsi che, nel caso pessimo, l'estrazione della soluzione risulti intrattabile. Per questa ragione è necessario ricorrere a una funzione euristica che ci possa aiutare a scegliere le azioni durante la ricerca all'indietro. Un approccio che funziona bene in pratica consiste nell'utilizzare un algoritmo greedy basato sul costo di livello dei letterali. Per ogni insieme di obiettivi, procediamo nell'ordine seguente:

1. scegliamo prima il letterale che ha il costo di livello più alto;
2. per soddisfare quel letterale, scegliamo per prima l'azione con le precondizioni più facili da ottenere. Questo significa scegliere l'azione che minimizza la somma (o il massimo) dei costi di livello delle sue precondizioni.

## Terminazione di GRAPHPLAN

Fin qui abbiamo ignorato la questione della terminazione. Se un problema non ha soluzione, possiamo essere certi che GRAPHPLAN non andrà in ciclo infinito, estendendo a ogni iterazione il grafo di pianificazione? La risposta è sì, ma una dimostrazione completa va oltre gli scopi di questo libro. Qui accenneremo solo ai concetti principali, in particolare quelli che gettano luce sui grafi di pianificazione in generale.

Il primo passo è notare che alcune proprietà dei grafi di pianificazione crescono o diminuiscono monotonicamente. “ $X$  cresce monotonicamente” significa che l’insieme degli  $X$  al livello  $i+1$  è un superinsieme (non necessariamente proprio) dell’insieme al livello  $i$ . Le proprietà sono le seguenti.

- ◆ *I letterali crescono monotonicamente:* una volta che un letterale compare in un dato livello, comparirà anche in tutti i successivi. Questa è una conseguenza della persistenza delle azioni.
- ◆ *Le azioni crescono monotonicamente:* una volta che un’azione compare in un dato livello, comparirà anche in tutti i successivi. Questa è una conseguenza dell’incremento dei letterali: se le precondizioni di un’azione sono presenti in un livello, lo saranno anche nei successivi, e così l’azione.
- ◆ *Il numero delle relazioni mutex decresce monotonicamente:* se due azioni sono mutex in un dato livello  $A_i$ , saranno mutex anche in tutti i livelli precedenti in cui appaiono entrambe. Lo stesso vale per le relazioni di mutua esclusione tra letterali. Nelle figure potrebbe non sembrare sempre così, perché nel disegno dei grafi c’è una semplificazione: non sono riportati né i letterali che non possono valere nel livello  $S_i$  né le azioni che non possono essere eseguite nel livello  $A_i$ . Potete constatare che è vero che il numero di mutex decresce monotonicamente se considerate che quei letterali e quelle azioni invisibili sono mutex con tutto.

La dimostrazione è piuttosto complessa, ma può essere gestita caso per caso: se le azioni  $A$  e  $B$  sono mutex al livello  $A_i$ , ci sono tre possibili ragioni. Le prime due sono *effetti inconsistenti* e *interferenza*, che sono proprietà delle azioni stesse, ragion per cui se le azioni sono mutex in  $A_i$  lo saranno per forza in ogni livello del grafo. Il terzo caso, *necessità in competizione*, dipende dalle condizioni al livello  $S_i$ : quel livello deve contenere una precondizione di  $A$  che è mutuamente esclusiva con una precondizione di  $B$ . Ora, queste due precondizioni possono essere mutex se sono una la negazione dell’altra (nel qual caso lo saranno in ogni livello) oppure se tutte le azioni che raggiungono una di esse sono mutex con tutte le azioni che raggiungono l’altra. Ma sappiamo già che le azioni disponibili crescono monotonicamente, ragion per cui per induzione i mutex devono diminuire.

Dato che le azioni e i letterali crescono e le relazioni mutex diminuiscono, e dato che il numero di azioni e letterali è obbligatoriamente finito, ogni grafo di pianificazione è destinato a un certo punto a livellarsi: da quel punto in poi, tutti i livelli saranno identici. Una volta che un grafo si è livellato, se uno degli obiettivi

del problema è assente o se due obiettivi sono mutex si può dire che il problema non ha soluzioni, e possiamo interrompere l'algoritmo GRAPHPLAN e restituire un fallimento. Se quando il grafo si livella tutti gli obiettivi sono presenti e non mutex, ma ESTRAI-SOLUZIONE non riesce a trovare una soluzione, potrebbe essere necessario estendere ancora il grafo per un numero finito di livelli, ma si può dimostrare che a un certo punto sarà possibile terminare. Quest'ultimo aspetto è più complesso, comunque, e non lo tratteremo qui.

## 11.5 Pianificazione con la logica proposizionale

Abbiamo visto nel Capitolo 10 che la pianificazione può essere effettuata dimostrando un teorema del calcolo delle situazioni. Il teorema afferma che, dato lo stato iniziale e gli assiomi di stato successore che descrivono gli effetti delle azioni, l'obiettivo sarà verificato nella situazione che risulta da una certa sequenza di azioni. Già nel 1969 si era giunti alla conclusione che quest'approccio era troppo inefficiente per trovare piani interessanti. I recenti sviluppi negli algoritmi di ragionamento efficienti per il calcolo proposizionale (v. Capitolo 7) hanno portato a un rinnovato interesse verso la pianificazione attraverso il ragionamento logico.

L'approccio che esamineremo in questo paragrafo è basato sulla verifica di soddisfabilità di una formula logica piuttosto che sulla dimostrazione vera e propria di un teorema. Cercheremo modelli di formule proposizionali che hanno quest'aspetto:

$$\text{stato iniziale} \wedge \text{tutte le possibili descrizioni di azione} \wedge \text{obiettivo} .$$

La formula conterrà simboli proposizionali corrispondenti a ogni possibile occorrenza di azione; un modello che soddisfa la formula assegnerà *true* alle azioni che fanno parte di un piano corretto e *false* alle altre. Un assegnamento che corrisponde a un piano scorretto non sarà un modello perché risulterà inconsistente con l'asserzione che l'obiettivo è vero. Se il problema di pianificazione è insolubile, la formula non sarà soddisfacibile.

### Descrivere problemi di pianificazione in logica proposizionale

Il processo che seguiremo per tradurre problemi STRIPS in logica proposizionale sarà un esempio tipico (per così dire) del ciclo di rappresentazione della conoscenza: cominceremo con quello che sembra un insieme ragionevole di assiomi, scopriremo che permette l'esistenza di modelli indesiderati, e a questo punto aggiungeremo altri assiomi.

Cominciamo con un problema di trasporto aereo molto semplice. Nello stato iniziale (tempo 0), l'aereo  $P_1$  è in *SFO* e l'aereo  $P_2$  è in *JFK*. L'obiettivo è avere  $P_1$  in *JFK* e  $P_2$  in *SFO*; in altre parole gli aerei devono scambiarsi di posto. Per prima cosa ci serviranno simboli proposizionali distinti per le asserzioni in ogni istante temporale. Per indicare il passo temporale useremo degli apici, come nel Capitolo 7. Quindi, lo stato iniziale si scrivrà

$$\text{Posizione}(P_1, \text{SFO})^0 \wedge \text{Posizione}(P_2, \text{JFK})^0.$$

Ricordate sempre che  $\text{Posizione}(P_1, \text{SFO})^0$  è un simbolo atomico! Dato che la logica proposizionale non comprende l'ipotesi del mondo chiuso, occorre anche specificare tutte le proposizioni che *non* sono vere nello stato iniziale. Le proposizioni sconosciute nello stato iniziale si possono lasciare non specificate (**ipotesi del mondo aperto**). In questo esempio scriviamo:

$$\neg \text{Posizione}(P_1, \text{JFK})^0 \wedge \neg \text{Posizione}(P_2, \text{SFO})^0.$$

Lo stesso obiettivo dev'essere associato a un particolare istante temporale. Dato che non è possibile sapere a priori quanti passi occorreranno per raggiungerlo, possiamo provare ad asserire che l'obiettivo è già vero nello stato iniziale, con  $T = 0$ : questo significa asserire  $\text{Posizione}(P_1, \text{JFK})^0 \wedge \text{Posizione}(P_2, \text{SFO})^0$ . Se questo fallisce, possiamo riprovare con  $T = 1$ , e continuare allo stesso modo finché non raggiungiamo una lunghezza minima accettabile del piano. Per ogni valore di  $T$  la base di conoscenza includerà solo le formule che coprono i passi temporali da 0 a  $T$ . Per assicurare la terminazione dell'algoritmo imporremo un limite superiore arbitrario  $T_{\max}$ . L'algoritmo è presentato nella Figura 11.15. Un approccio alternativo, che non necessita di tentativi multipli, è discusso nell'Esercizio 11.17.

---

```

function SATPLAN(problema, T_{\max}) returns una soluzione, o il fallimento
 inputs: problema, un problema di pianificazione
 T_{\max} , un limite superiore della lunghezza del piano

 for $T = 0$ to T_{\max} do
 cnf, mapping \leftarrow TRADUCI-IN-SAT(problema, T)
 assegnamento \leftarrow RISOLTORE-SAT(cnf)
 if assegnamento non è null then
 return ESTRAI-SOLUZIONE(assegnamento, mapping)
 return fallimento

```

---

**Figura 11.15** L'algoritmo SATPLAN. Il problema di pianificazione è tradotto in una formula CNF in cui si asserisce che l'obiettivo è verificato in un istante temporale prefissato  $T$  e che include assiomi per ogni istante fino a  $T$  (i dettagli del processo di traduzione sono forniti nel testo). Se l'algoritmo di soddisfacibilità trova un modello, viene estratto un piano guardando i simboli proposizionali che si riferiscono ad azioni e che hanno valore *true*. Se non esiste alcun modello, il processo è ripetuto dopo aver spostato l'obiettivo un passo più avanti.

Il problema successivo è come codificare le descrizioni delle azioni in logica proposizionale. L'approccio più diretto è usare un simbolo proposizionale per ogni occorrenza di azione: ad esempio,  $Vola(P_1, SFO, JFK)^0$  è vero se l'aereo  $P_1$  vola da  $SFO$  a  $JFK$  nell'istante 0. Come nel Capitolo 7, scriveremo delle versioni proposizionali degli assiomi di stato successore sviluppati per il calcolo delle situazioni del Capitolo 10. Ad esempio, avremo

$$\begin{aligned} Posizione(P_1, JFK)^1 \Leftrightarrow \\ (Posizione(P_1, JFK)^0 \wedge \neg(Vola(P_1, JFK, SFO)^0 \wedge Posizione(P_1, JFK)^0)) \\ \vee (Vola(P_1, SFO, JFK)^0 \wedge Posizione(P_1, SFO)^0). \end{aligned} \quad (11.1)$$

La formula dice che l'aereo  $P_1$  si troverà in  $JFK$  nell'istante 1 se c'era già nell'istante 0 e non se ne è andato, oppure se era in  $SFO$  nell'istante 0 ed è volato in  $JFK$ . Ci servirà un assioma simile per ogni aereo, aeroporto e istante temporale. Inoltre, ogni aeroporto aggiuntivo aggiungerà un altro modo di raggiungere o partire da una posizione e aggiungerà quindi nuovi disgiunti alla parte destra di ogni assioma.

Una volta scritti tutti gli assiomi, per trovare un piano ci basta eseguire l'algoritmo di verifica di soddisfabilità. Dev'esserci per forza un piano che soddisfa l'obiettivo al tempo  $T = 1$ , e precisamente quello in cui gli aerei volano entrambi e si scambiano di posto. Supponiamo che la KB contenga

$$stato\ iniziale \wedge assiomi\ di\ stato\ successore \wedge obiettivo^1, \quad (11.2)$$

che asserisce che l'obiettivo è raggiunto nell'istante  $T = 1$ . Potete verificare che l'assegnamento in cui

$$Vola(P_1, SFO, JFK)^0 \text{ e } Vol(P_2, JFK, SFO)^0$$

Sono veri e tutti gli altri simboli di azione sono falsi è un modello della KB. Fin qui, tutto bene. Ma esistono altri possibili modelli che l'algoritmo di soddisfabilità potrebbe restituire? In effetti, sì. Questi altri modelli sono tutti piani accettabili? Purtroppo no. Considerate il piano alquanto sciocco specificato dai simboli di azione

$$Vola(P_1, SFO, JFK)^0 \text{ e } Vol(P_1, JFK, SFO)^0 \text{ e } Vol(P_2, JFK, SFO)^0.$$

Questo piano è sciocco perché l'aereo  $P_1$  comincia in  $SFO$ , per cui l'azione  $Vola(P_1, JFK, SFO)^0$  non è eseguibile. Nonostante questo, il piano è *effettivamente* un modello per la formula dell'Equazione (11.2)! In effetti, potete verificare che è consistente con quanto detto sin qui. Per capire il perché, dobbiamo guardare attentamente quello che dicono gli assiomi di stato successore (come l'Equazione (11.1)) sulle azioni le cui precondizioni non sono soddisfatte. Gli assiomi predico-

no correttamente che non succederà nulla quando un'azione simile è eseguita (v. Esercizio 11.15), ma *non* dicono che l'azione non può essere eseguita! Per evitare di generare piani che includono azioni illegali dobbiamo aggiungere **assiomi di precondizione** che affermano che l'occorrenza di un'azione richiede che le sue precondizioni siano soddisfatte.<sup>6</sup> Per esempio, dobbiamo scrivere

$$\text{Vola}(P_1, \text{JFK}, \text{SFO})^0 \Rightarrow \text{Posizione}(P_1, \text{JFK})^0.$$

Dato che  $\text{Posizione}(P_1, \text{JFK})^0$  è falso nello stato iniziale, quest'assioma assicura che anche ogni modello assegnerà il valore falso a  $\text{Vola}(P_1, \text{JFK}, \text{SFO})^0$ . Con l'aggiunta degli assiomi di precondizione c'è esattamente un modello che soddisfa tutti gli assiomi e raggiunge l'obiettivo nell'istante 1: quello in cui l'aereo  $P_1$  vola in *JFK* e l'aereo  $P_2$  vola in *SFO*. Notate che questa soluzione ha due azioni in parallelo, proprio come GRAPHPLAN o POP.

Quando aggiungiamo un terzo aeroporto, *LAX*, sorgono nuove sorprese. Ora ogni aereo ha due azioni legali in ogni stato. Quando eseguiamo l'algoritmo di soddisfabilità, troviamo che un modello con  $\text{Vola}(P_1, \text{SFO}, \text{JFK})^0$  e  $\text{Vola}(P_2, \text{JFK}, \text{SFO})^0$  e  $\text{Vola}(P_2, \text{JFK}, \text{LAX})^0$  soddisfa tutti gli assiomi. I nostri assiomi di stato successore e di precondizione permettono a un aereo di volare contemporaneamente in due posti diversi! Le precondizioni dei due voli di  $P_2$  sono soddisfatte nello stato iniziale; gli assiomi di stato successore dicono che  $P_2$  sarà in *SFO* e anche in *LAX* nell'istante 1; l'obiettivo è soddisfatto. È chiaro che dobbiamo aggiungere ulteriori assiomi per eliminare queste soluzioni spurie. Un approccio consiste nell'aggiungere **assiomi di esclusione tra azioni** che impediscono il verificarsi di azioni simultanee. Ad esempio, possiamo realizzare un'esclusione completa aggiungendo tutti i possibili assiomi della forma

$$\neg(\text{Vola}(P_2, \text{JFK}, \text{SFO})^0 \wedge \text{Vola}(P_2, \text{JFK}, \text{LAX})^0).$$

Questi assiomi assicurano che non si potranno mai verificare due azioni nello stesso istante. Così facendo eliminano tutti i piani spuri, ma obbligano anche i piani a essere totalmente ordinati. In questo modo si perde la flessibilità dei piani parzialmente ordinati; inoltre, aumentando il numero di passi temporali del piano, il tempo di calcolo potrebbe aumentare.

Invece di un'esclusione completa, potremmo richiederne solo una parziale: in altre parole, impedire le azioni simultanee solo se interferiscono l'una con l'altra. Le condizioni sono le stesse che per le azioni mutex: due azioni non possono avve-

assiomi di  
precondizione

assiomi di esclusione  
tra azioni

---

<sup>6</sup> Notate che l'aggiunta di assiomi di precondizione significa che non occorre includere le precondizioni delle azioni negli assiomi di stato successore.

nire simultaneamente se una nega la precondizione o l'effetto dell'altra. Ad esempio,  $Vola(P_2, JFK, SFO)^0$  e  $Vola(P_2, JFK, LAX)^0$  non possono accadere entrambe, perché ognuna nega la precondizione dell'altra; d'altra parte,  $Vola(P_1, SFO, JFK)^0$  e  $Vola(P_2, JFK, SFO)^0$  possono verificarsi insieme perché i due aerei non interferiscono. L'esclusione parziale elimina i piani spuri senza forzare un ordinamento totale.

Gli assiomi di esclusione potrebbero apparire uno strumento alquanto rozzo. Invece di richiedere che un aereo non possa volare in due aeroporti diversi allo stesso tempo, potremmo semplicemente specificare che nessun oggetto può trovarsi contemporaneamente in due posti diversi:

$$\forall p, x, y, t \quad x \neq y \Rightarrow \neg(Posizione(p, x)^t \wedge Posizione(p, y)^t).$$

vincoli di stato

Questo fatto, unito agli assiomi di stato successore, *implica* che un aereo non può volare in due aeroporti diversi contemporaneamente. Fatti come questo prendono il nome di **vincoli di stato**. Nella logica proposizionale, naturalmente, dovremo scrivere esplicitamente tutte le istanze ground di ogni vincolo di stato. Per il problema degli aeroporti, il vincolo di stato è sufficiente a eliminare tutti i piani spuri. Spesso i vincoli di stato sono molto più compatti degli assiomi di esclusione tra azioni, ma non sempre è facile derivarli dalla descrizione originale STRIPS di un problema.

Riassumendo, la pianificazione basata sulla soddisfacibilità richiede di trovare i modelli di una formula che contiene lo stato iniziale, l'obiettivo, gli assiomi di stato successore, gli assiomi di precondizione e gli assiomi di esclusione tra azioni oppure i vincoli di stato. Si può dimostrare che questa collezione di assiomi è sufficiente, nel senso che non ci saranno più "soluzioni" spurie. Ogni modello che soddisfa la formula proposizionale sarà un piano valido per il problema originale: ovvero, ogni linearizzazione di quel piano sarà una sequenza legale di azioni che raggiunge l'obiettivo.

## Complessità delle codifiche proposizionali

La principale limitazione dell'approccio proposizionale è la dimensione della base di conoscenza generata dal problema iniziale di pianificazione. Ad esempio, lo schema di azione  $Vola(p, a_1, a_2)$  diventa  $T \times |Aerei| \times |Aeroporti|^2$  simboli proposizionali differenti. In generale, il numero totale di simboli di azione è limitato da  $T \times |Act| \times |O|^P$ , dove  $|Act|$  è il numero di schemi di azione,  $|O|$  quello di oggetti nel dominio e  $P$  è l'arità (numero di argomenti) massima tra tutti gli schemi di azione. Il numero di clausole è ancora più grande. Ad esempio, con 10 passi temporali, 12 aerei e 30 aeroporti, l'assioma di esclusione tra azioni completo ha 583 milioni di clausole.

Dato che il numero di simboli di azione cresce con l'esponente dell'arità degli schemi, si potrebbe pensare di ridurre l'arità stessa. Per far questo possiamo prendere in prestito un'idea dalle reti semantiche (v. Capitolo 10). Queste ultime infatti usano solo predicati binari; quelli con più argomenti sono ridotti a un in-

sieme di predicati binari che descrivono ogni argomento separatamente. Applicando quest'idea a un simbolo di azione come  $Vola(P_1, SFO, JFK)^0$  otteniamo tre nuovi simboli:

$Vola_1(P_1)^0$ : l'aereo  $P_1$  vola al tempo 0

$Vola_2(SFO)^0$ : l'origine del volo è  $SFO$

$Vola_3(JFK)^0$ : la destinazione del volo è  $JFK$ .

Questo processo, chiamato **divisione dei simboli** (*symbol splitting*), elimina la necessità di usare un numero di simboli esponenziale: ora ne servono solo  $T \times |Act| \times P \times |O|$ .

divisione dei simboli

Da solo lo splitting può ridurre il numero dei simboli, ma non quello degli assiomi nella KB. Se ogni simbolo di azione in ogni clausola fosse semplicemente rimpiazzato da una congiunzione di tre simboli, la dimensione totale della KB rimarrebbe più o meno la stessa. Lo splitting dei simboli in effetti riduce le dimensioni della KB, perché alcuni simboli divisi saranno irrilevanti per certi assiomi e potranno essere omessi. Ad esempio, considerate l'assioma di stato successore dell'Equazione (11.1), modificato per includere  $LAX$  e omettere le precondizioni delle azioni (che saranno trattate separatamente per mezzo di assiomi di precondizione):

$$\begin{aligned} Posizione(P_1, JFK)^1 \Leftrightarrow \\ (Posizione(P_1, JFK)^0 \wedge \neg Vola(P_1, JFK, SFO)^0 \wedge \neg Vola(P_1, JFK, LAX)^0) \\ \vee Vola(P_1, SFO, JFK)^0 \vee Vola(P_1, LAX, JFK)^0. \end{aligned}$$

La prima condizione dice che  $P_1$  si troverà in  $JFK$  se era lì al tempo 0 e non è volato in alcuna altra città, non importa quale; la seconda dice che l'aereo sarà in  $JFK$  se è arrivato in volo da una qualsiasi altra città, non importa quale. Usando i simboli divisi, possiamo semplicemente omettere l'argomento il cui valore non ha importanza:

$$\begin{aligned} Posizione(P_1, JFK)^1 \Leftrightarrow (Posizione(P_1, JFK)^0 \wedge \neg(Vola_1(P_1)^0 \wedge Vola_2(JFK)^0)) \\ \vee (Vola_1(P_1)^0 \wedge Vola_3(JFK)^0). \end{aligned}$$

Notate che  $SFO$  e  $LAX$  non sono più neppure menzionati dall'assioma. In generale, i simboli di azione divisi permettono che la dimensione di ogni assioma di stato successore sia indipendente dal numero di aeroporti. Riduzioni simili si verificano con gli assiomi di precondizione e quelli di esclusione tra azioni (v. Esercizio 11.16). Nel caso descritto poco fa con 10 passi temporali, 12 aerei e 30 aeroporti, l'assioma completo di esclusione tra azioni sarà ridotto da 583 milioni a 9.360 clausole.

C'è tuttavia un inconveniente: la rappresentazione con simboli divisi non permette azioni parallele. Considerate le due azioni parallele  $Vola(P_1, SFO, JFK)^0$  e  $Vola(P_2, JFK, SFO)^0$ . Convertendole nella rappresentazione divisa, avremo

$$\begin{aligned} & Vola_1(P_1)^0 \wedge Vola_2(SFO)^0 \wedge Vola_3(JFK)^0 \wedge \\ & Vola_1(P_2)^0 \wedge Vola_2(JFK)^0 \wedge Vola_3(SFO)^0. \end{aligned}$$

Non è più possibile determinare che cos'è successo! Sappiamo che gli aerei  $P_1$  e  $P_2$  hanno volato, ma non possiamo più identificare l'origine e la destinazione di ogni volo. Questo significa che dovremo usare un assioma completo di esclusione tra azioni, con gli inconvenienti che abbiamo discusso.

I pianificatori basati sulla soddisfacibilità possono gestire problemi di grandi dimensioni: ad esempio, trovare una soluzione ottima in 30 passi a un problema nel mondo dei blocchi con dozzine di blocchi. La dimensione della codifica proposizionale e il costo della soluzione dipendono fortemente dal problema, ma nella maggior parte dei casi il collo di bottiglia è rappresentato dalla memoria necessaria alla memorizzazione degli assiomi proposizionali. Un risultato interessante è la scoperta che nella risoluzione di problemi di pianificazione gli algoritmi con backtracking come DPLL si comportano spesso meglio degli algoritmi di ricerca locale come WALKSAT. Questo è dovuto al fatto che la maggior parte degli assiomi proposizionali sono clausole di Horn, gestite in modo efficiente dalla tecnica di propagazione delle unità. Quest'osservazione ha portato allo sviluppo di algoritmi ibridi che combinano la ricerca locale con il backtracking e la propagazione delle unità.

## 11.6 Analisi degli approcci alla pianificazione

Oggi la pianificazione è un'area di grande interesse. Una delle ragioni è che combina le due aree principali dell'IA che abbiamo trattato fin qui: la *ricerca* e la *logica*. Un pianificatore infatti può essere considerato un programma che cerca una soluzione, ma anche uno che dimostra (in modo costruttivo) l'esistenza di una soluzione. Il fertile interscambio di idee tra le due aree ha portato a miglioramenti nelle prestazioni di diversi ordini di grandezza nell'ultimo decennio e a un incremento nell'uso dei pianificatori nelle applicazioni industriali. Sfortunatamente, non abbiamo ancora una comprensione chiara di quali tecniche funzionano meglio per ogni tipo di problema. È probabile che emergano nuove tecniche in grado di dominare i metodi esistenti.

La pianificazione è prima di tutto un esercizio nel controllo dell'esplosione combinatoria. Se in un dominio ci sono  $p$  proposizioni primitive, ci saranno  $2^p$  stati. In domini complessi,  $p$  può diventare molto grande. Considerate che gli oggetti nel dominio hanno proprietà (*Forma*, *Colore* etc.) e relazioni (*Sopra*, *Compreso Tra* etc.). Con  $d$  oggetti in un dominio che prevede relazioni ternarie, avremo  $2^{d^3}$  stati. Potremmo concludere che, nel caso pessimo, la pianificazione è un'impresa senza speranza.

L'approccio *divide et impera* può rappresentare un'arma potente contro tale pessimismo. Nel caso più favorevole, in cui il problema è perfettamente scomponibile, la velocità aumenta esponenzialmente. La possibilità di scomporre il problema, comunque, è impedita dalle interazioni negative tra le azioni. I pianificatori con ordinamento parziale gestiscono questo problema per mezzo dei

collegamenti causali, che sono un potente strumento di rappresentazione, ma sfortunatamente ogni conflitto dev'essere risolto con una scelta (di spostare l'azione in conflitto prima o dopo il collegamento), e le scelte possono moltiplicarsi in modo esponenziale. GRAPHPLAN evita queste scelte durante la fase di costruzione del grafo, usando collegamenti mutex per registrare i conflitti senza scegliere come risolverli. SATPLAN memorizza un insieme simile di relazioni di mutua esclusione, ma per far ciò usa la forma generale CNF anziché una specifica struttura dati. Quanto bene questa soluzione funzioni in pratica dipende dal risolutore SAT utilizzato.

Talvolta è possibile risolvere un problema in modo efficiente riconoscendo che le interazioni negative possono essere evitate. Diciamo che un problema ha **sotto-obiettivi serializzabili** se esiste un ordinamento tra i sotto-obiettivi tale che il pianificatore li può ottenere in quell'ordine senza essere obbligato a invalidare obiettivi già raggiunti. Ad esempio, nel mondo dei blocchi, se l'obiettivo è costruire una torre (come quella composta da *A* su *B*, che a sua volta si trova su *C*, che a sua volta è sul *Tavolo*), allora i sotto-obiettivi sono serializzabili dal basso verso l'alto: se per prima cosa otteniamo *C* sul *Tavolo*, non dovremo mai annullare quest'obiettivo man mano che lavoriamo per soddisfare gli altri. Un pianificatore che utilizza la tecnica dal basso verso l'alto potrà risolvere qualsiasi problema nel dominio nel mondo dei blocchi senza tornare mai sui suoi passi (benché possa non trovare sempre il piano più breve).

Passando a un esempio più complesso, per il pianificatore Remote Agent che comandava la navicella Deep Space One della NASA, fu determinato che le proposizioni relative al comando di un veicolo spaziale sono serializzabili. Questo non dovrebbe sorprendere, perché un veicolo spaziale è progettato dagli ingegneri per essere controllato nel modo più facile possibile (nel rispetto di tutti i vincoli). Sfruttando l'ordinamento serializzato degli obiettivi, il pianificatore Remote Agent è stato capace di eliminare la maggior parte della ricerca. In questo modo è risultato abbastanza veloce da controllare la navicella in tempo reale, una cosa precedentemente considerata impossibile.

C'è più di un modo di controllare l'esplosione combinatoria. Abbiamo visto nel Capitolo 5 che esistono molte tecniche per controllare il backtracking nei problemi di soddisfacimento di vincoli (CSP), come il backtracking guidato dalle dipendenze. Tutte queste tecniche possono essere applicate anche alla pianificazione. Estrarre una soluzione da un grafo di pianificazione, per esempio, può essere formulato come un CSP booleano le cui variabili indicano se una data azione deve verificarsi in un dato istante. Il CSP può essere risolto usando uno qualsiasi degli algoritmi visti nel Capitolo 5, come min-conflicts. Un metodo strettamente imparentato, usato nel sistema BLACKBOX, è convertire il grafo di pianificazione in un'espressione CNF ed estrarre un piano usando un risolutore SAT. Quest'approccio sembra funzionare meglio di SATPLAN, presumibilmente perché il grafo di pianificazione ha già eliminato dal problema molti degli stati e delle azioni impossibili; e meglio anche di GRAPHPLAN, presumibilmente perché una ricerca di

sotto-obiettivi  
serializzabili

soddisficiabilità come WALKSAT è molto più flessibile della rigida ricerca con backtracking utilizzata da GRAPHPLAN.

Non c'è dubbio che pianificatori come GRAPHPLAN, SATPLAN e BLACKBOX hanno fatto progredire il campo della pianificazione, alzando il livello delle prestazioni e favorendo la comprensione degli aspetti problematici di rappresentazione e combinatori. Questi metodi, comunque, sono intrinsecamente proposizionali e possono quindi esprimere domini limitati (ad esempio, problemi di logistica con solo qualche dozzina di oggetti e locazioni richiedono gigabyte di memoria per memorizzare le corrispondenti espressioni CNF). Probabilmente per ottenere ulteriori progressi sarà necessario ricorrere a rappresentazioni e algoritmi del primo ordine, benché strutture come i grafi di pianificazione continueranno a essere utili come fonte di euristiche.

## 11.7 Riepilogo

---

In questo capitolo abbiamo definito il problema della pianificazione negli ambienti deterministici e completamente osservabili. Abbiamo descritto le principali rappresentazioni dei problemi e diversi approcci algoritmici per la loro risoluzione. I punti da ricordare sono i seguenti.

- ◆ I sistemi di pianificazione sono algoritmi di risoluzione di problemi che operano su rappresentazioni esplicite proposizionali (o del primo ordine) degli stati e delle azioni. Queste rappresentazioni rendono possibile derivare euristiche efficaci e sviluppare algoritmi potenti e flessibili.
- ◆ Il linguaggio STRIPS descrive le azioni in termini di precondizioni ed effetti e gli stati iniziali e gli obiettivi come congiunzioni di letterali positivi. Il linguaggio ADL rilassa alcune di queste limitazioni permettendo l'uso di disgiunzioni, negazioni e quantificatori.
- ◆ La ricerca nello spazio degli stati può procedere in avanti (**progressione**) o all'indietro (**regressione**). Si possono derivare euristiche efficaci con l'ipotesi dell'indipendenza dei sotto-obiettivi o mediante diversi rilassamenti del problema di pianificazione.
- ◆ Gli algoritmi di pianificazione con ordinamento parziale (POP) esplorano lo spazio dei piani senza scegliere una sequenza totalmente ordinata di azioni. La loro ricerca procede all'indietro dall'obiettivo, aggiungendo azioni al piano per raggiungere via via ogni sotto-obiettivo. Sono particolarmente efficaci sui problemi che si prestano a un approccio *divide et impera*.
- ◆ Un **grafo di pianificazione** può essere costruito in modo incrementale, partendo dallo stato iniziale. Ogni livello contiene un superinsieme dei letterali e delle azioni che si potrebbero verificare in quel passo temporale e le relazioni di mutua esclusione, o **mutex**, tra letterali e azioni che non possono verifi-

carsi contemporaneamente. I grafi di pianificazione forniscono euristiche utili per i pianificatori nello spazio degli stati e con ordinamento parziale, e possono essere usati direttamente dall'algoritmo GRAPHPLAN.

- ♦ L'algoritmo GRAPHPLAN processa un grafo di pianificazione ed estrae un piano mediante una ricerca all'indietro, permettendo (in un certo grado) l'ordinamento parziale delle azioni.
- ♦ L'algoritmo SATPLAN traduce un problema di pianificazione in assiomi proposizionali e applica un algoritmo di soddisfabilità per trovare un modello che corrisponde a un piano valido. Sono state sviluppate molte rappresentazioni proposizionali differenti, con diversi livelli di compattezza ed efficienza.
- ♦ Ognuno dei principali approcci alla pianificazione ha i suoi estimatori, e non c'è ancora un consenso comune su quale sia l'approccio migliore. La competizione e il proficuo interscambio tra gli approcci hanno causato un incremento significativo nell'efficienza dei sistemi di pianificazione.

## Note storiche e bibliografiche

La pianificazione in IA scaturì dalle investigazioni nella ricerca nello spazio degli stati, nella dimostrazione di teoremi e nella teoria del controllo, e dalle necessità pratiche della robotica, dello scheduling e di altri domini. STRIPS (Fikes e Nilsson, 1971), il primo sistema di pianificazione importante, evidenzia l'interazione tra tutte queste influenze. STRIPS fu progettato come componente di pianificazione per il software del progetto del robot Shakey, allo SRI. La sua struttura di controllo generale fu modellata su quella di GPS, il General Problem Solver (Newell e Simon, 1961), un sistema di ricerca nello spazio degli stati che sfruttava l'analisi mezzi-finì. STRIPS utilizzò una versione del dimostratore di teoremi QA3 (Green, 1969b) come subroutine per stabilire la verità delle precondizioni delle azioni. Lifschitz (1986) offre definizioni precise e un'analisi del linguaggio STRIPS. Bylander (1992) dimostra che la pianificazione STRIPS semplice è PSPACE-completa. Fikes e Nilsson (1993) forniscono una retrospettiva storica sul progetto STRIPS e una panoramica delle sue relazioni con i sistemi di pianificazione più recenti.

La rappresentazione delle azioni di STRIPS è stata di gran lunga più influente del suo approccio algoritmico: quasi tutti i sistemi di pianificazione successivi hanno usato una qualche variazione del suo linguaggio. Sfortunatamente, la proliferazione di varianti ha reso inutilmente difficile il confronto fra di esse. Col tempo si è raggiunta una migliore comprensione delle limitazioni e dei compromessi dei vari formalismi. L'Action Description Language, o ADL (Pednault, 1986), ha rilassato alcune delle restrizioni del linguaggio STRIPS rendendo possibile la codifica di problemi più realistici. Nebel (2000) esamina alcuni schemi per la compilazione di ADL in STRIPS. Il Problem Domain Description Language, o PDDL (Ghallab et al., 1998), fu introdotto come una sintassi standard riconoscibile dal computer per la rappresentazione di STRIPS, ADL e altri linguaggi. PDDL è stato usato come

pianificazione lineare

interleaving

linguaggio standard per le gare di pianificazione che, a partire dal 1998, si sono tenute durante il congresso AIPS.

All'inizio degli anni '70 i pianificatori generalmente lavoravano su sequenze di azioni totalmente ordinate. La scomposizione dei problemi si otteneva calcolando un sottopiano per ogni sotto-obiettivo e poi concatenando in qualche ordine tutti i sottopiani. Quest'approccio, chiamato **pianificazione lineare** da Sacerdoti (1975), si rivelò presto incompleto: in effetti non può risolvere alcuni problemi molto semplici, come l'anomalia di Sussman (v. Esercizio 11.11), scoperta da Allen Brown durante la sperimentazione con il sistema HACKER (Sussman, 1975). Un pianificatore completo deve consentire l'**interleaving**, ovvero l'esecuzione alternata delle azioni di sottopiani diversi in una singola sequenza. La nozione di sotto-obiettivi serializzabili (Korf, 1987) corrisponde esattamente all'insieme di problemi per cui i pianificatori senza interleaving sono completi.

Una soluzione al problema dell'interleaving fu la pianificazione con regressione dell'obiettivo, una tecnica in cui i passi di un piano totalmente ordinato sono riordinati per evitare conflitti tra i sotto-obiettivi. Introdotto da Waldinger (1975), l'approccio fu usato anche da Warren (1974) per WARPLAN. WARPLAN è notevole anche per il fatto di essere il primo pianificatore scritto in un linguaggio di programmazione logica (Prolog) ed è uno dei migliori esempi della notevole economia che contraddistingue talvolta la programmazione logica: WARPLAN è costituito da 100 righe di codice, una piccola frazione delle dimensioni dei pianificatori dell'epoca. Anche INTERPLAN (Tate, 1975a, 1975b) permetteva l'interleaving arbitrario dei passi di un piano per superare l'anomalia di Sussman e problemi simili.

Le idee alla base della pianificazione con ordinamento parziale includono il rilevamento dei conflitti (Tate, 1975a) e la protezione delle condizioni raggiunte dalle interferenze (Sussman, 1975). La costruzione di piani parzialmente ordinati (allora chiamati **task network**, o *reti di attività*) fu tentata per la prima volta dal pianificatore NOAH (Sacerdoti, 1975, 1977) e dal sistema NONLIN di Tate (1975b, 1977).<sup>7</sup>

La pianificazione con ordinamento parziale dominò i vent'anni successivi di ricerca, eppure per gran parte di quel periodo non si può dire che fosse ampiamente compresa. TWEAK (Chapman, 1987) era basato su una ricostruzione logica e una semplificazione dello stato dell'arte della pianificazione in quel momento; la sua formulazione era così chiara da permettere la dimostrazione delle completezza e dell'intrattabilità (NP-difficoltà e indecidibilità) di varie formulazioni del problema di pianificazione. Il lavoro di Chapman portò a quella che si può definire la prima descrizione semplice e leggibile di un pianificatore completo con ordina-

<sup>7</sup> C'è un po' di confusione sulla terminologia: molti autori usano il termine **non lineare** con il significato di "parzialmente ordinato". Quest'uso è leggermente diverso da quello originale di Sacerdoti, che si riferiva ai piani con interleaving.

mento parziale (McAllester e Rosenblitt, 1991). Un'implementazione dell'algoritmo di McAllester e Rosenblitt chiamato SNLP (Soderland e Weld, 1991) fu ampiamente distribuita e permise a molti ricercatori di comprendere e sperimentare per la prima volta la pianificazione con ordinamento parziale. L'algoritmo POP descritto in questo capitolo è basato su SNLP.

Il gruppo di Weld sviluppò anche UCPOP (Penberthy e Weld, 1992), il primo pianificatore per problemi espressi in ADL. UCPOP incorporava l'euristica basata sul numero di obiettivi insoddisfatti. Era un po' più veloce di SNLP, ma raramente riusciva a trovare piani che comprendessero più di una dozzina di passi. Benché fossero sviluppate per UCPOP euristiche migliori (Joslin e Pollack, 1994; Gerevini e Schubert, 1996), la pianificazione con ordinamento parziale perse interesse negli anni '90 quando emersero metodi più veloci. Nguyen e Kambhampati (2001) suggeriscono che si debba pensare a una riabilitazione: con euristiche accurate derivate da un grafo di pianificazione, il loro pianificatore REPOP scala verso l'alto molto meglio di GRAPHPLAN ed è in grado di rivaleggiare con i pianificatori nello spazio degli stati più veloci.

Avrim Blum e Merrick Furst (1995, 1997) diedero nuova linfa al campo della pianificazione con il loro sistema GRAPHPLAN, che era interi ordini di grandezza più veloce dei pianificatori con ordinamento parziale del suo tempo. Presto seguirono altri sistemi di pianificazione basata su grafo, come IPP (Koehler et al., 1997), STAN (Fox e Long, 1998) e SGP (Weld et al., 1998). Una struttura dati molto simile al grafo di pianificazione era stata sviluppata poco prima da Ghallab e Laruelle (1994), per essere utilizzata dal pianificatore con ordinamento parziale IXTET per derivare euristiche accurate. Nguyen et al. (2001) offrono un'analisi molto esauriente delle euristiche derivate dai grafi di pianificazione. La nostra discussione dei grafi è basata in parte su questo lavoro e su note di Subbarao Kambhampati. Come abbiamo detto nel capitolo, un grafo di pianificazione può essere usato in molti modi per guidare la ricerca. Il vincitore della gara di pianificazione AIPS del 2002, LPG (Gerevini e Serina, 2002), cercava nei grafi di pianificazione usando una tecnica di ricerca locale ispirata a WALKSAT.

La pianificazione basata sulla soddisfacibilità e l'algoritmo SATPLAN furono proposti da Kautz e Selman (1992), che erano stati ispirati dal sorprendente successo dei metodi di ricerca locale greedy per i problemi di soddisfacibilità (v. Capitolo 7). Kautz et al. (1996) investigarono varie forme di rappresentazione proposizionale per gli assiomi STRIPS, trovando che le forme più compatte non portavano necessariamente a tempi più veloci. Un'analisi sistematica fu svolta da Ernst et al. (1997), che svilupparono anche un "compilatore" automatico per la generazione di rappresentazioni proposizionali di problemi PDDL. Il pianificatore BLACKBOX, che prende alcune idee sia da GRAPHPLAN che da SATPLAN, è stato sviluppato da Kautz e Selman (1998).

Il ritorno di interesse per la pianificazione nello spazio degli stati fu spronato dal programma UNPOP di Drew McDermott (1996), che fu il primo a suggerire un'euristica della distanza basata su un problema rilassato che ignorava le liste di

diagrammi binari di decisioni

cancellazione. Il nome UNPOP era una reazione al fatto che in quel periodo i ricercatori si concentravano quasi esclusivamente sulla pianificazione con ordinamento parziale; McDermott sospettava che gli altri approcci non godessero dell'attenzione che meritavano. L'Heuristic Search Planner (HSP) di Bonet e Geffner e i suoi sviluppi successivi (Bonet e Geffner, 1999) furono i primi sistemi a rendere pratica la ricerca nello spazio degli stati per problemi grandi. Oggi come oggi, il sistema di maggior successo che esegue ricerche nello spazio degli stati è FASTFORWARD (o FF) di Hoffmann (2000), vincitore della gara di pianificazione AIPS 2000. FF usa un'euristica basata su un grafo di pianificazione semplificato con un algoritmo di ricerca molto veloce che unisce ricerca in avanti e ricerca locale in modo innovativo.

Di recente, c'è stato molto interesse nella rappresentazione dei piani come **diagrammi binari di decisioni**, una descrizione compatta degli automi a stati finiti ampiamente studiata nella comunità di verifica dell'hardware (Clarke e Grumberg, 1987; McMillan, 1993). Esistono tecniche per dimostrare le proprietà dei diagrammi binari di decisione, tra cui quella di essere una soluzione di un problema di pianificazione. Cimatti et al. (1998) presentano un pianificatore basato su quest'approccio. Sono state usate anche altre rappresentazioni; ad esempio, Vossen et al. (2001) esaminano l'uso della programmazione intera applicata alla pianificazione.

Benché sia ancora presto per trarre conclusioni, cominciano a comparire interessanti confronti dei vari approcci alla pianificazione. Helmert (2001) analizza diverse classi di problemi di pianificazione e dimostra che gli approcci basati sui vincoli come GRAPHPLAN e SATPLAN sono più adatti ai domini NP-difficili, mentre quelli basati sulla ricerca si comportano meglio nei domini in cui è possibile trovare soluzioni accettabili senza backtracking. GRAPHPLAN e SATPLAN si trovano in difficoltà nei domini con molti oggetti, perché questo significa che devono creare molte azioni. In alcuni casi il problema può essere rimandato o evitato generando le azioni proposizionalizzate in modo dinamico, man mano che se ne presenta la necessità, anziché istanziarle tutte prima di cominciare la ricerca.

Weld (1994, 1999) fornisce due panoramiche eccellenti degli algoritmi di pianificazione moderni. È interessante notare i cambiamenti avvenuti nei cinque anni che separano i due lavori: il primo si concentra sulla pianificazione con ordinamento parziale, il secondo introduce GRAPHPLAN e SATPLAN. *Readings in Planning* (Allen et al., 1990) è una ricca antologia di molti dei migliori articoli nel campo, tra cui diverse buone panoramiche. Yang (1997) fornisce una visione d'insieme, lunga quanto un libro, delle tecniche di pianificazione con ordinamento parziale.

La ricerca sulla pianificazione è stata una parte centrale dell'IA fin dall'inizio, e gli articoli che la riguardano si trovano in tutte le riviste e i congressi. Ci sono anche congressi specializzati, come l'International Conference on AI Planning Systems (AIPS), l'International Workshop on Planning and Scheduling for Space, e la European Conference on Planning.

## Esercizi

11.1 Descrivete le differenze e i punti in comune tra la risoluzione di problemi e la pianificazione.

11.2 Dati gli assiomi della Figura 11.2, quali sono le istanze concrete applicabili di  $Vola(p, da, a)$  nello stato descritto da

$$\begin{aligned} & Posizione(P_1, JFK) \wedge Posizione(P_2, SFO) \wedge Aereo(P_1) \wedge Aereo(P_2) \\ & \wedge Aeroporto(JFK), Aeroporto(SFO) ? \end{aligned}$$

11.3 Investigiamo come si potrebbe tradurre un insieme di schemi STRIPS negli assiomi di stato successore del calcolo delle situazioni (v. Capitolo 10).

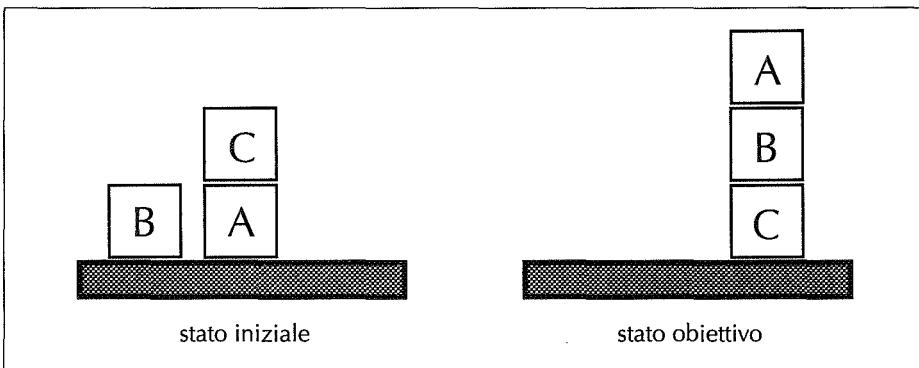
- ◆ Considerate lo schema di  $Vola(p, da, a)$ . Scrivete una definizione logica del predicato  $VolaPrecond(p, da, a, s)$ , che è vero se le precondizioni di  $Vola(p, da, a)$  sono soddisfatte nella situazione  $s$ .
- ◆ Ora, presumendo che  $Vola(p, da, a)$  sia l'unico schema di azione disponibile, scrivete un assioma di stato successore per  $Posizione(p, x, s)$  che catturi la stessa informazione dello schema di azione.
- ◆ Ora supponete che esista un altro modo di viaggiare: *Teletrasbordo* ( $p, da, a$ ), che ha la precondizione aggiuntiva  $\neg CurvaturaUno(p)$  e l'effetto aggiuntivo *CurvaturaUno*( $p$ ). Spiegate come dev'essere modificata la base di conoscenza per il calcolo delle situazioni.
- ◆ Infine, sviluppate una procedura generale e precisamente definita per eseguire la traduzione da un insieme di schemi STRIPS a un insieme di assiomi di stato successore.

11.4 Il problema della scimmia e delle banane è costituito da una scimmia chiusa in un laboratorio con delle banane appese al soffitto troppo in alto per essere raggiunte. È presente uno scatolone, che una volta scalato permetterà alla scimmia di raggiungere i succosi frutti. Inizialmente la scimmia si trova in *A*, le banane in *B*, lo scatolone in *C*. La scimmia e lo scatolone hanno un'altezza *Bassa*, ma una volta salito sullo scatolone il primate si troverà a un'altezza *Alta*, la stessa delle banane. Le azioni disponibili alla scimmia includono *Vai* da un posto all'altro, *Springi* un oggetto da un posto all'altro, *SaliSu* o *ScendiGiù* da un oggetto, e *Afferra* o *Lascia* un oggetto. Afferrare un oggetto ha come risultato che la scimmia lo sta portando, se i due si trovano nello stesso posto e alla stessa altezza.

- a. Scrivete la descrizione dello stato iniziale.
- b. Scrivete le definizioni delle sei azioni nello stile di STRIPS.
- c. Supponiamo che la scimmia voglia farsi beffe degli scienziati, che sono andati a farsi un caffè, prendendo le banane ma lasciando lo scatolone

nella sua posizione originale. Scrivete questo obiettivo in forma generale (cioè, non dando per scontato che la sua posizione iniziale sia in C) nel linguaggio del calcolo delle situazioni. Quest'obiettivo può essere risolto da un sistema STRIPS?

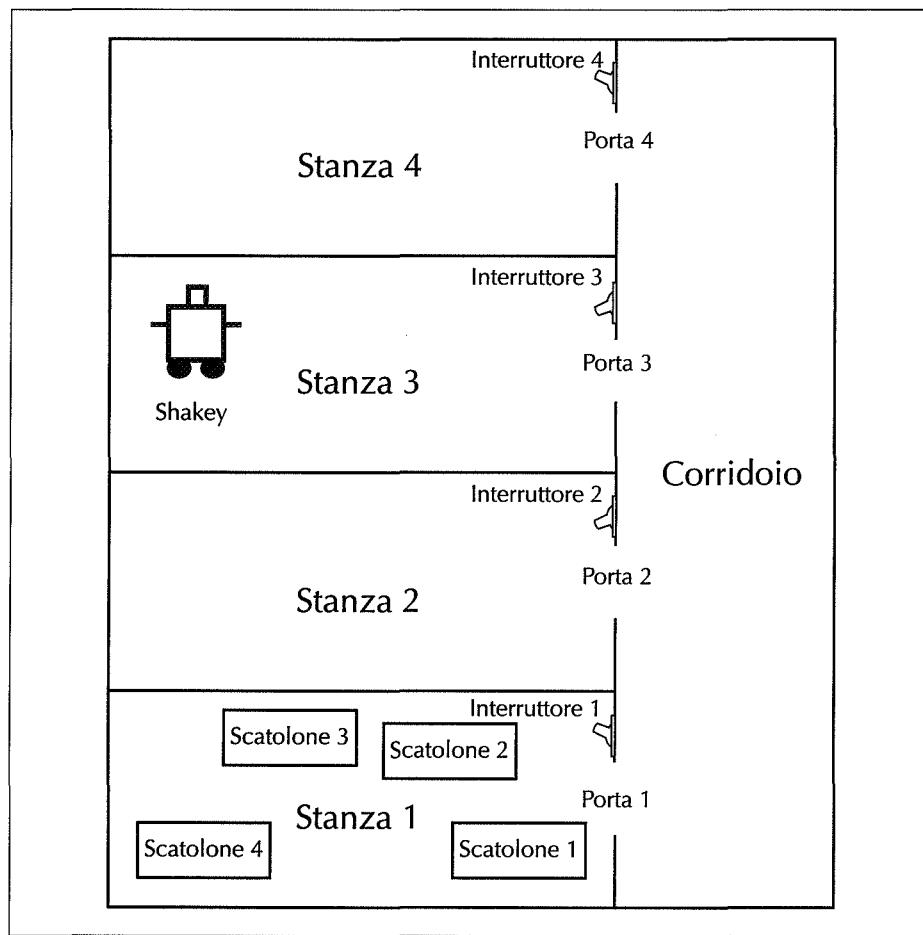
- d. Il vostro assioma per lo spostamento degli oggetti è probabilmente scorretto, perché un oggetto troppo pesante rimarrà fermo anche quando si applica l'operatore *Spring*. Questo è un esempio del problema di ramificazione o di qualificazione? Correggete la vostra descrizione del problema in modo da prendere in considerazione gli oggetti pesanti.
- 11.5 Spiegate perché il processo di generazione dei predecessori nella ricerca all'indietro non deve aggiungere i letterali che rappresentano effetti negativi dell'azione.
- 11.6 Spiegate perché l'eliminazione di tutti gli effetti negativi da ogni schema di azione in un problema STRIPS ha come risultato un problema rilassato.
- 11.7 Esaminate la definizione di ricerca bidirezionale del Capitolo 3.
- Sarebbe una buona idea applicare la ricerca bidirezionale nello spazio degli stati alla pianificazione?
  - E la ricerca bidirezionale nello spazio dei piani con ordinamento parziale?
  - Progettate una versione della pianificazione con ordinamento parziale in cui un'azione può essere aggiunta al piano se le sue precondizioni possono essere soddisfatte dagli effetti delle azioni già presenti nel piano. Spiegate come gestire i conflitti e i vincoli di ordinamento. Pensate che quest'algoritmo sia essenzialmente identico alla ricerca in avanti nello spazio degli stati?
  - Considerate un pianificatore con ordinamento parziale che combina il metodo della parte (c) con la tecnica standard di aggiungere azioni per soddisfare le precondizioni aperte. L'algoritmo risultante sarebbe lo stesso della parte (b)?
- 11.8 Costruite i livelli 0, 1 e 2 del grafo di pianificazione per il problema della Figura 11.2.
- 11.9 Dimostrate le seguenti asserzioni sui grafi di pianificazione.
- ♦ Un letterale che non compare nel livello finale del grafo non può mai essere soddisfatto.
  - ♦ Il costo di livello di un letterale in un grafo seriale non è maggiore del costo effettivo di un piano ottimo che lo soddisfa.



**Figura 11.16** Il problema di pianificazione nel mondo dei blocchi noto come “anomalia di Sussman”.

- 11.10** Abbiamo distinto nettamente i pianificatori in avanti e all’indietro nello spazio degli stati con quelli con ordinamento parziale, dicendo che questi ultimi eseguono la ricerca nello spazio dei piani. Spiegate come la ricerca nello spazio degli stati può essere considerata anch’essa una ricerca nello spazio dei piani, e indicate quali sono gli operatori di raffinamento del piano.
- 11.11** La Figura 11.16 illustra un problema nel mondo dei blocchi noto come **anomalia di Sussman**. Il problema è stato considerato anomalo perché i pianificatori dei primi anni ’70, privi totalmente della capacità di eseguire l’interleaving delle azioni, non potevano risolverlo. Scrivete una definizione del problema in notazione STRIPS e risolvetelo, a mano o usando un programma di pianificazione. Un pianificatore privo di interleaving, dati due sotto-obiettivi  $G_1$  e  $G_2$ , produce un piano per  $G_1$  concatenato a uno per  $G_2$  o viceversa. Spiegate perché un simile pianificatore non può risolvere questo problema.
- 11.12** Considerate il problema di infilarsi calze e scarpe definito nel Paragrafo 11.3. Applicate GRAPHPLAN al problema e mostrate la soluzione ottenuta. Ora aggiungete azioni per infilarsi cappotto e cappello. Calcolate il piano con ordinamento parziale che risolve il problema e mostrate che prevede 180 diverse linearizzazioni. Qual è il numero minimo di soluzioni diverse sul grafo di pianificazione necessarie per rappresentare tutte le 180 linearizzazioni?

anomalia di Sussman



**Figura 11.17** Il mondo di Shakey. Shakey può muoversi tra gli oggetti di una stanza, attraversare le porte che separano le stanze, arrampicarsi sugli oggetti che è possibile scalare, spingere quelli che è possibile spostare e azionare gli interruttori della luce.

**11.13** L'originale programma STRIPS era stato progettato per controllare il robot Shakey. La Figura 11.17 mostra una versione del mondo di Shakey, che consiste in quattro stanze allineate lungo un corridoio. Ogni stanza ha una porta e un interruttore della luce.

Le azioni nel mondo di Shakey includono spostarsi da un luogo all'altro, spingere oggetti mobili (come scatoloni), salire e scendere da oggetti rigidi (ancora una volta, scatoloni) e spegnere/accendere interruttori. Il robot stesso non raggiunse mai un grado di destrezza sufficiente a salire su

una scatola o azionare un interruttore, ma il pianificatore STRIPS fu capace di trovare piani che andavano oltre le capacità fisiche del robot. Le sei azioni di Shakey sono le seguenti.

- ◆  $Vai(x, y)$ , che richiede che Shakey si trovi in  $x$  e che  $x$  e  $y$  siano locazioni nella stessa stanza. Per convenzione, una porta che separa due stanze è considerata presente in entrambe.
- ◆ Spingere una scatola  $b$  dalla locazione  $x$  alla locazione  $y$  nella stessa stanza:  $Spingi(b, x, y)$ . Occorrerà definire un predicato *Scatolone* e costanti per tutte le scatole.
- ◆ Salire su una scatola:  $SalisU(b)$ ; scendere da una scatola:  $ScendiGiù(b)$ . Servirà il predicato *Sopra* e la costante *Pavimento*.
- ◆ Azionare un interruttore:  $Accendi(s)$  e  $Spegni(s)$ . Per accendere o spegnere la luce, Shakey deve trovarsi in cima a una scatolone in corrispondenza dell'interruttore.

Descrivete le sei azioni di Shakey e lo stato iniziale della Figura 11.17 in notazione STRIPS. Costruite un piano affinché Shakey spinga lo *Scatolone2* nella *Stanza2*.

**11.14** Abbiamo visto che i grafi di pianificazione possono gestire solo azioni proposizionali. Come possiamo usarli se il nostro problema ha variabili nell'obiettivo, come  $Posizione(P_1, x) \wedge Posizione(P_2, x)$ , dove  $x$  varia su un intervallo finito di posizioni? Come potreste codificare un problema simile in modo che sia possibile utilizzare i grafi di pianificazione? (Suggerimento: ricordate l'azione *Fine* della pianificazione POP. Quali precondizioni dovrebbe avere?)

**11.15** Finora abbiamo dato per scontato che le azioni siano eseguite solo nelle situazioni appropriate. Vediamo cosa possono dire gli assiomi di stato successore proposizionali, come l'Equazione (11.1), sulle azioni le cui precondizioni non sono soddisfatte.

a. Mostrate che gli assiomi predicono che non accadrà nulla quando un'azione viene eseguita in uno stato in cui le sue precondizioni non sono soddisfatte.

b. Considerate un piano  $p$  che contiene le azioni richieste per raggiungere un obiettivo ma anche azioni illegali. Si può dire che

$$\text{stato iniziale} \wedge \text{assiomi di stato successore} \wedge p \models \text{obiettivo?}$$

c. Con assiomi di stato successore del primo ordine nel calcolo delle situazioni (come nel Capitolo 10), è possibile dimostrare che un piano che contiene azioni illegali può raggiungere un obiettivo?

- 11.16 Fornendo esempi presi dal dominio degli aeroporti, spiegate come lo splitting o divisione dei simboli riduce la dimensione degli assiomi di precondizione e di quelli di esclusione tra azioni. Derivate una formula generale per la dimensione di ogni insieme di azioni in termini di numero di passi temporali, numero di schemi di azioni, arità di queste ultime e numero di oggetti.
- 11.17 Nell'algoritmo SATPLAN della Figura 11.15, ogni chiamata dell'algoritmo di soddisfacibilità ha un obiettivo  $g^T$ , con  $T$  che va da 0 a  $T_{\max}$ . Supponiamo che l'algoritmo di soddisfacibilità venga invece chiamato una volta sola, con l'obiettivo  $g^0 \vee g^1 \vee \dots \vee g^{T_{\max}}$ .
- a. Questo restituirà sempre un piano di lunghezza minore o uguale a  $T_{\max}$ , se esiste?
  - b. Quest'approccio introduce nuove “soluzioni” spurie?
  - c. Discutete come si potrebbe modificare un algoritmo di soddisfacibilità come WALKSAT in modo che, una volta fornитогli un obiettivo disgiuntivo in questo formato, possa trovare soluzioni brevi (se esistono).

## Capitolo 12

# Pianificazione e azione nel mondo reale

*In cui vediamo come rappresentazioni più espressive e architetture agente più interattive danno origine a pianificatori utili nel mondo reale.*

Il capitolo precedente ha introdotto le nozioni fondamentali, le rappresentazioni e gli algoritmi per la pianificazione. I pianificatori usati nel mondo reale per compiti come lo scheduling delle osservazioni del telescopio spaziale Hubble, il funzionamento delle fabbriche e la gestione logistica militare sono più complessi di quelli considerati sin qui ed estendono i concetti base sia dal punto di vista della rappresentazione del linguaggio che nel modo in cui interagiscono con l'ambiente.

Il Paragrafo 12.1 descrive la pianificazione e lo scheduling in presenza di vincoli temporali e di risorse; il Paragrafo 12.2 introduce l'uso di sottopiani predefiniti. I paragrafi dal 12.3 al 12.6 presentano una serie di architetture agente progettate per funzionare in ambienti incerti. Il Paragrafo 12.7 mostra come cambia la pianificazione quando l'ambiente contiene altri agenti.

## 12.1 Tempo, scheduling e risorse

La rappresentazione STRIPS dice *quali* azioni eseguire ma, dato che è basata sul calcolo delle situazioni, non può dire *quanto tempo* ci vuole per eseguirle e neppure *quando* avvengono, tranne la possibilità di specificare che un'azione avviene prima o dopo un'altra. In alcuni domini, però, vorremmo essere in grado di dire quando cominciano e finiscono le azioni: ad esempio, nel trasporto delle merci ci piacerebbe sapere quando arriverà un aereo che trasporta un determinato pacco, e non solo che sarà a destinazione quando avrà terminato il volo.

*Init(Chassis( $C_1$ )  $\wedge$  Chassis( $C_2$ ))*

$\wedge$  Motore( $E_1$ ,  $C_1$ , 30)  $\wedge$  Motore( $E_2$ ,  $C_2$ , 60)

$\wedge$  Ruote( $W_1$ ,  $C_1$ , 30)  $\wedge$  Ruote( $W_2$ ,  $C_2$ , 15))

*Obiettivo(Fatto( $C_1$ )  $\wedge$  Fatto( $C_2$ ))*

*Azione(MontaMotore( $e$ ,  $c$ ),*

**PRECOND:** Motore( $e$ ,  $c$ ,  $d$ )  $\wedge$  Chassis( $c$ )  $\wedge$   $\neg$ MotoreIn( $c$ ),

**EFFETTO:** MotoreIn( $c$ )  $\wedge$  Durata( $d$ ))

*Azione(MontaRuote( $w$ ,  $c$ ),*

**PRECOND:** Ruote( $w$ ,  $c$ ,  $d$ )  $\wedge$  Chassis( $c$ )  $\wedge$  MotoreIn( $c$ ),

**EFFETTO:** RuoteSu( $c$ )  $\wedge$  Durata( $d$ ))

*Azione(Ispeziona( $c$ ), PRECOND: MotoreIn( $c$ )  $\wedge$  RuoteSu( $c$ )  $\wedge$  Chassis( $c$ ),*

**EFFETTO:** Fatto( $c$ )  $\wedge$  Durata(10))

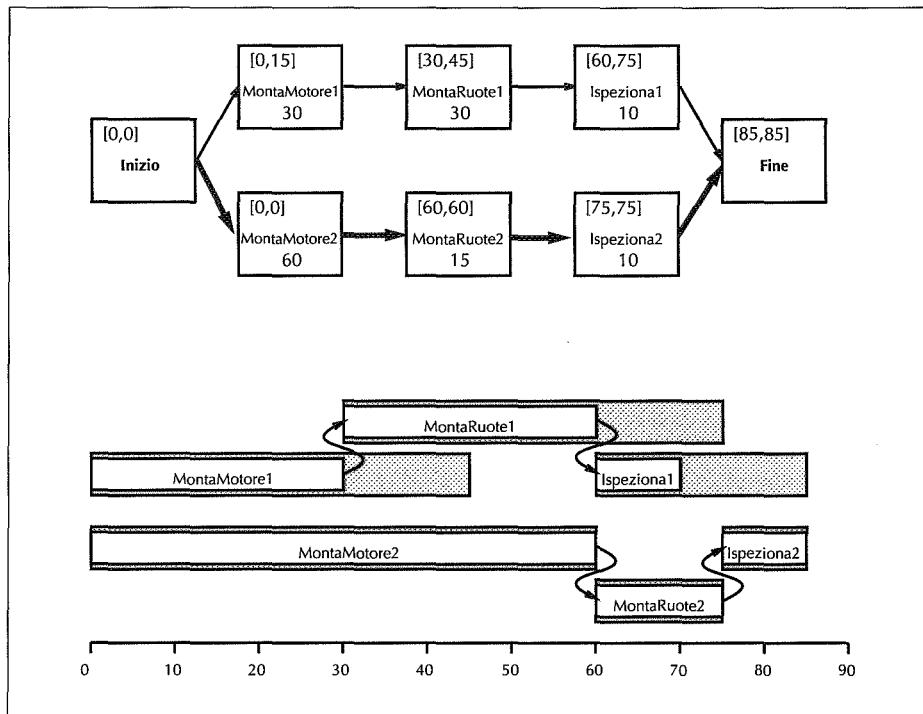
**Figura 12.1** Un problema di job shop scheduling per l’assemblaggio di due macchine. La notazione *Durata( $d$ )* significa che un’azione richiede  $d$  minuti per essere eseguita. *Motore( $E_1$ ,  $C_1$ , 60)* indica che  $E_1$  è un motore adatto allo chassis  $C_1$  e richiede 60 minuti per il montaggio.

#### job shop scheduling

Il tempo è fondamentale in una famiglia generale di applicazioni chiamate **job shop scheduling**. Questi problemi richiedono il completamento di una serie di attività o “job”, ognuna costituita da un sequenza di azioni con una durata precisa; ogni azione potrebbe anche necessitare di particolari risorse. Il problema è determinare una schedule, ovvero un “orario dettagliato” delle attività, che minimizzi il tempo totale richiesto per completare tutti i job rispettando i vincoli sulle risorse.

Un esempio di problema di job shop scheduling è illustrato nella Figura 12.1. Si tratta di un problema di assemblaggio di parti di automobile molto semplificato. I job sono due: assemblare le macchine  $C_1$  e  $C_2$ . Ognuno consiste in tre azioni: aggiungere il motore, aggiungere le ruote e ispezionare il risultato. Il motore dev’essere aggiunto per primo, perché le ruote anteriori impediscono l’accesso libero al vano, e naturalmente l’ispezione dev’essere svolta per ultima.

Il problema della Figura 12.1 può essere risolto da tutti i pianificatori che abbiamo già visto. La Figura 12.2 (ignorando per ora i numeri) riporta una soluzione che potrebbe benissimo essere stata trovata da un pianificatore POP con ordinamento parziale. Per rendere questo un problema di *scheduling*, piuttosto che di *pianificazione*, occorre determinare gli istanti di inizio e fine di ogni azione. Questo significa che si deve considerare la durata delle azioni e non solo il loro ordinamento. La notazione *Durata( $d$ )* nell’effetto di un’azione, in cui  $d$  dev’essere obbligatoriamente legata a un numero, significa che l’azione richiede  $d$  minuti per essere completata.



**Figura 12.2** Una soluzione del problema di job shop scheduling della Figura 12.1. In alto la soluzione è rappresentata come un piano con ordinamento parziale. La durata di ogni azione è specificata in basso in ogni rettangolo, con il primo e ultimo istante accettabile di inizio dell'azione indicati come  $[ES, LS]$  in alto a sinistra. La differenza tra questi due numeri è il margine dell'azione; le azioni con margine zero si trovano sul cammino critico, evidenziato da linee in grassetto. Nella parte bassa della figura la stessa soluzione è rappresentata per mezzo di un diagramma dei tempi. I rettangoli grigi rappresentano gli intervalli temporali durante i quali un'azione può essere eseguita, a patto che siano rispettati i vincoli di ordinamento. La parte libera di un rettangolo grigio indica il margine.

Dato un ordinamento parziale delle azioni con associate le durate, come quello della Figura 12.2, possiamo applicare il **metodo del cammino critico** (CPM) per determinare i possibili istanti di inizio e fine di ogni azione. Un **cammino** attraverso un piano con ordinamento parziale è una sequenza linearmente ordinata di azioni che comincia con *Inizio* e termina con *Fine*: ad esempio, nel piano della Figura 12.2 ci sono due cammini.

Il **cammino critico** è quello con la durata totale più lunga. Si chiama “critico” perché determina la durata dell’intero piano: accorciare altri cammini non accorcia il piano globale, mentre rimandare l’inizio di una qualsiasi azione sul piano critico lo rallenta. Il piano critico è indicato mediante linee in grassetto. Per completare l’intero piano nel minimo tempo totale, le azioni sul piano critico devono essere

metodo del cammino critico

cammino critico

margini

schedule

risorse

risorsa riutilizzabile

eseguite una dopo l'altra. Le azioni poste al di fuori del cammino critico hanno un po' di margine, in altre parole possono essere eseguite in una finestra temporale. Per specificare tale finestra si deve indicare il primo istante in cui è possibile iniziare l'azione (*ES*, da *Earliest Start*) e l'ultimo (*LS*, da *Latest Start*). La quantità  $LS - ES$  è denominata **margine** (*slack*) dell'azione. Possiamo vedere nella Figura 12.2 che l'intero piano richiederà 80 minuti, che ogni azione sul cammino critico ha margine 0 (sarà sempre così) e che ognuna delle azioni nell'assemblaggio di  $C_1$  ha un margine di 10 minuti. Insieme, gli *ES* e *LS* di tutte le azioni costituiscono una **schedule** del problema.

Le seguenti formule definiscono *ES* e *LS* e servono anche da guida per un algoritmo di programmazione dinamica capace di calcolarle:

$$ES(\text{Inizio}) = 0$$

$$ES(B) = \max_{A \prec B} ES(A) + \text{Durata}(A)$$

$$LS(\text{Fine}) = ES(\text{Fine})$$

$$LS(A) = \min_{A \prec B} LS(B) - \text{Durata}(A).$$

L'idea è di cominciare assegnando 0 a *ES*(*Inizio*). Non appena abbiamo un'azione *B* tale che tutte le azioni che vengono immediatamente prima hanno dei valori di *ES* assegnati, imponiamo che *ES*(*B*) sia il valore massimo tra i tempi finali più bassi di tutte le azioni immediatamente precedenti a *B*, dove il "tempo finale più basso" di un'azione è definito come il suo *ES* più la sua durata. Questo processo si ripete finché è stato assegnato un valore *ES* a ogni azione. I valori *LS* sono calcolati in modo simile, muovendosi all'indietro dall'azione *Fine*. I dettagli sono lasciati come esercizio ai lettori.

La complessità dell'algoritmo del cammino critico è solo  $O(Nb)$ , dove *N* è il numero di azioni e *b* il fattore di ramificazione massimo in ingresso o uscita da un'azione (per constatarlo, considerate che il calcolo di *LS* e *ES* viene effettuato una volta per ogni azione, e ogni volta itera al più su *b* altre azioni). Di conseguenza il problema di trovare una schedule di minima durata, *dato un ordinamento parziale delle azioni*, è abbastanza facile da risolvere.

## Scheduling con vincoli sulle risorse

I problemi di scheduling reali sono complicati dalla presenza di vincoli sulle risorse. Ad esempio, aggiungere un motore a una macchina richiede un paranco specializzato. Se ce n'è solo uno, diventa impossibile aggiungere simultaneamente il motore  $E_1$  alla macchina  $C_1$  e quello  $E_2$  alla  $C_2$ ; di conseguenza la schedule della Figura 12.2 diventa inutilizzabile. Il paranco è un esempio di **risorsa riutilizzabile**, che risulta "occupata" durante l'azione ma diventa nuovamente disponibile immediatamente dopo la sua fine. Notate che le risorse riutilizzabili non possono essere gestite dalla nostra descrizione standard delle azioni in termini di precondizioni ed effetti, perché la quantità di risorse disponibili non cambia dopo che l'azione è completata.<sup>1</sup>

---

*Init*(Chassis( $C_1$ )  $\wedge$  Chassis( $C_2$ )  
 $\wedge$  Motore( $E_1$ ,  $C_1$ , 30)  $\wedge$  Motore( $E_2$ ,  $C_2$ , 60)  
 $\wedge$  Ruote( $W_1$ ,  $C_1$ , 30)  $\wedge$  Ruote( $W_2$ ,  $C_2$ , 15)  
 $\wedge$  ParanchiPerMotore(1)  $\wedge$  StazioniMontaggioRuote(1)  $\wedge$  Ispettori(2))  
*Obiettivo*(Fatto( $C_1$ )  $\wedge$  Fatto( $C_2$ ))

*Azione*(MontaMotore( $e$ ,  $c$ ),

PRECOND: Motore( $e$ ,  $c$ ,  $d$ )  $\wedge$  Chassis( $c$ )  $\wedge$   $\neg$ MotoreIn( $c$ ),

EFFETTO: MotoreIn( $c$ )  $\wedge$  Durata( $d$ ),

RISORSA: ParanchiPerMotore(1))

*Azione*(MontaRuote( $w$ ,  $c$ ),

PRECOND: Ruote( $w$ ,  $c$ ,  $d$ )  $\wedge$  Chassis( $c$ )  $\wedge$  MotoreIn( $c$ ),

EFFETTO: RuoteSu( $c$ )  $\wedge$  Durata( $d$ ),

RISORSA: StazioniMontaggioRuote(1))

*Azione*(Ispeziona( $c$ ),

PRECOND: MotoreIn( $c$ )  $\wedge$  RuoteSu( $c$ ),

EFFETTO: Fatto( $c$ )  $\wedge$  Durata(10),

RISORSA: Ispettori(1))

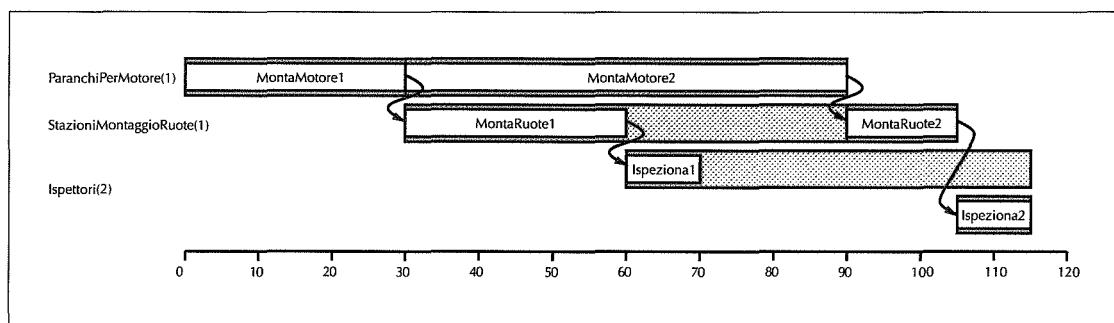
---

**Figura 12.3** Problema di job shop scheduling per l'assemblaggio di due macchine, con risorse. Le risorse disponibile sono un paranco per motori, una stazione di montaggio per le ruote e due ispettori. La notazione RISORSA:  $r$  significa che la risorsa  $r$  è utilizzata durante l'esecuzione dell'azione, ma ridiventa libera non appena questa è completata.

Per questa ragione arricchiremo la nostra rappresentazione aggiungendo un campo di forma RISORSA:  $R(k)$ , che significa che l'azione richiede  $k$  unità della risorsa  $R$ . Il requisito delle risorse rappresenta sia una precondizione (l'azione non può essere eseguita se la risorsa non è disponibile) sia un effetto *temporaneo*, nel senso che la disponibilità della risorsa  $r$  è ridotta di  $k$  per tutta la durata dell'azione. La Figura 12.3 mostra come si può estendere il problema di assemblaggio in modo da includere tre risorse: un paranco speciale per il montaggio dei motori, una stazione di installazione per le ruote e due ispettori. La Figura 12.4 mostra la soluzione con il minor tempo cumulativo, 115 minuti. Questo tempo è maggiore degli 80 minuti richiesti dalla schedule priva di vincoli sulle risorse. Notate che non c'è alcun momento in cui

---

<sup>1</sup> Al contrario le risorse consumabili, come le viti usate nel montaggio del motore, possono essere gestite all'interno dell'infrastruttura standard: cfr. l'Esercizio 12.2.



**Figura 12.4** Una soluzione per il problema di job shop scheduling con risorse della Figura 12.3. A sinistra sono indicate le tre risorse, e le azioni sono allineate orizzontalmente con le risorse consumate. Ci sono due possibili schedule, a seconda di quale linea di assemblaggio usa il paranco per prima; noi abbiamo mostrato la soluzione ottima, che richiede 115 minuti.

entrambi gli ispettori sono necessari, per cui è possibile rimuoverne uno immediatamente e assegnarlo a una mansione più produttiva.

La rappresentazione delle risorse come quantità numeriche, come *Ispettori(2)*, anziché entità dotate di nome, come *Ispettore( $I_1$ )* e *Ispettore( $I_2$ )*, è un esempio di una tecnica molto generale chiamata **aggregazione**. L'idea centrale è raggruppare oggetti distinti in quantità collettive quando sono tutti indistinguibili ai fini della nostra attività. Nel nostro problema di assemblaggio non importa *quale* ispettore esamina le macchine, per cui non c'è alcuna ragione di distinguere tra i due (la stessa idea si applica al problema dei missionari e dei cannibali dell'Esercizio 3.9). L'aggregazione è fondamentale per ridurre la complessità. Considerate che cosa succede quando si propone una schedule che ha 10 azioni *Ispeziona* concorrenti, ma sono disponibili solo 9 ispettori. Quando gli ispettori sono rappresentati come quantità il rilevamento del fallimento è istantaneo, e l'algoritmo può tornare indietro con il backtracking e provare una schedule diversa. Se gli ispettori fossero rappresentati con oggetti singoli, l'algoritmo proverebbe inutilmente tutti i  $10!$  modi di assegnarli alle azioni *Ispeziona*.

Nonostante i loro vantaggi, i vincoli sulle risorse rendono più complicati i problemi di scheduling introducendo interazioni aggiuntive tra le azioni. Laddove eseguire uno scheduling senza vincoli è facile usando il metodo del cammino critico, il problema diventa NP-difficile se si introducono le risorse e si desidera trovare la soluzione che richiede il tempo globale minimo. Questa complessità si manifesta nella pratica oltre che in teoria: un problema dimostrativo posto nel 1963 – trovare la schedule ottima per un problema con 10 macchine e 10 job, ognuno di 100 azioni – non fu risolto prima di 23 anni (Lawler et al., 1993). Per ridurre la complessità sono stati provati molto approcci, tra cui branch-and-bound, simulated annealing, ricerca con tabù, soddisfacimento di vincoli e altre tecniche viste

aggregazione

nella Parte II. Un'euristica semplice ma diffusa è quella del **margine minimo**, che organizza le azioni in modo greedy. A ogni iterazione l'algoritmo considera le azioni fuori dalla schedule i cui predecessori sono stati già inseriti tutti e seleziona quella con il margine minimo in modo che si attivi il più in fretta possibile. A questo punto non deve far altro che aggiornare i tempi *ES* e *LS* di ogni azione coinvolta e ripetere il processo. L'euristica è basata sullo stesso principio della variabile più vincolata nel soddisfacimento di vincoli: in pratica funziona spesso bene, ma nel caso del nostro problema di assemblaggio restituisce una soluzione di 130 minuti, rispetto a quella da 115 della Figura 12.4.

L'approccio adottato in questo paragrafo si potrebbe definire “pianifica prima, fai lo scheduling dopo”: in altre parole, il problema viene diviso in una fase di *pianificazione* in cui le azioni sono scelte e parzialmente ordinate per soddisfare gli obiettivi del problema, e in una fase successiva di *scheduling* in cui il problema viene arricchito con informazione temporale per assicurarci che rispetti i vincoli di risorse e di tempo. Questo è un approccio comune negli ambienti di produzione e logistica reali, in cui la fase di pianificazione è spesso eseguita da esperti umani. Quando i vincoli sulle risorse sono severi, tuttavia, potrebbe darsi che alcuni piani legali portino a schedule molto peggiori di quelle ottenibili da altri piani. In tal caso ha senso *integrare* le due fasi prendendo in considerazione la durata e le sovrapposizioni delle azioni già durante la costruzione del piano parzialmente ordinato. Molti degli algoritmi trattati nel Capitolo 11 possono essere espansi per gestire questo tipo di informazione: ad esempio, i pianificatori con ordinamento parziale possono rilevare le violazioni dei vincoli sulle risorse in modo molto simile ai conflitti con i collegamenti causali. Le euristiche possono essere modificate per stimare il tempo totale di completamento di un piano invece del semplice costo totale delle azioni. Questa è un'area di ricerca molto attiva.

margine minimo

## 12.2 Pianificazione con reti gerarchiche

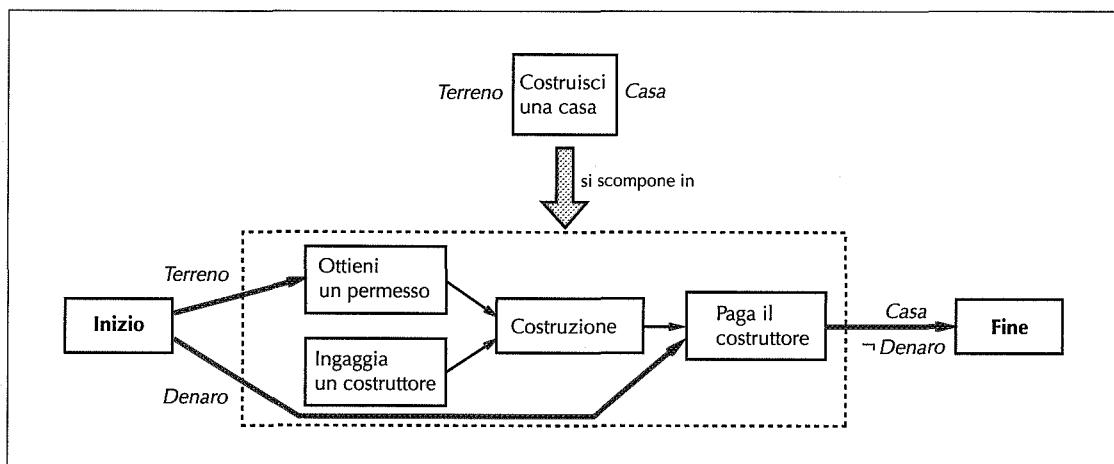
Una delle idee più diffuse per gestire la complessità è la **scomposizione gerarchica**. I software più complessi sono costruiti partendo da una gerarchia di procedure o classi di oggetti, gli eserciti operano come una gerarchia di unità, i governi e le aziende hanno gerarchie di dipartimenti, uffici e sedi distaccate. Il maggior beneficio di una struttura gerarchica è costituito dal fatto che, a ogni livello della gerarchia, un'attività computazionale, una missione militare o una funzione amministrativa si riducono a un *piccolo* numero di attività al livello immediatamente inferiore, cosicché il costo di trovare il modo migliore di organizzare tali attività risulta parimenti piccolo. I metodi non gerarchici, al contrario, riducono un'attività a un *grande* numero di azioni singole; se il problema è di grande scala quest'approccio è del tutto impraticabile. Nel caso migliore, in cui alle soluzioni di alto livello corrispondono sempre implementazioni soddisfacenti a basso livello, i metodi gerarchici possono ridurre il tempo di esecuzione degli algoritmi di pianificazione da esponenziale a lineare.

scomposizione gerarchica

reti gerarchiche

Questo paragrafo descrive un metodo di pianificazione basato sulle reti gerarchiche (o HTN, da *Hierarchical Task Networks*). L'approccio che presentiamo prende ispirazione sia dalla pianificazione con ordinamento parziale (v. Paragrafo 11.3) che dall'area nota specificatamente come "HTN planning". In quest'ultima il piano iniziale viene considerato una descrizione ad alto livello di quello che dev'essere va fatto: ad esempio, costruire una casa. I piani sono quindi raffinati applicando **scomposizioni di azioni**. Ogni scomposizione riduce un'azione di alto livello in un insieme parzialmente ordinato di azioni di livello più basso. Le scomposizioni di azioni, quindi, contengono informazioni su come implementare le azioni stesse. Ad esempio, costruire una casa potrebbe ridursi alla scomposizione mostrata nella Figura 12.5: ottenere un permesso, stipulare un contratto con un costruttore, eseguire la costruzione e pagare il contratto. Il processo di scomposizione continua finché nel piano non restano solo **azioni primitive**. Tipicamente, le azioni primitive sono quelle che l'agente può eseguire automaticamente. Per un agente che stipula contratti, "realizzare un giardino" potrebbe essere un'azione primitiva, perché si risolve chiamando un giardiniere e stipulando un contratto con lui. Per il giardiniere, un'azione primitiva potrebbe essere "pianta dei rododendri in questa posizione".

Nella pianificazione HTN "pura", i piani sono generati *soltanto* attraverso le scomposizioni successive delle azioni. HTN quindi considera la pianificazione un processo che rende le descrizioni delle attività più *concrete*, invece di *costruire* le descrizioni stesse cominciando da un'attività vuota, com'è il caso della pianificazione nello spazio degli stati e quella con ordinamento parziale. In realtà ogni descrizione di azione STRIPS può essere trasformata in una scomposizione (v. Esercizio



**Figura 12.5** Una possibile scomposizione dell'azione *CostruisciCasa*.

12.6), e la pianificazione con ordinamento parziale può essere considerato un caso speciale di HTN puro. Per certe attività, comunque, e in particolar modo quando gli obiettivi sono espressi in forma congiuntiva, il punto di vista di HTN è alquanto innaturale: per questo motivo preferiremo adottare un approccio *ibrido* in cui la scomposizione di azioni è usata come passo di raffinamento in un piano con ordinamento parziale, in aggiunta alle operazioni standard di scelta di una condizione aperta e di risoluzione di conflitti con l'aggiunta di vincoli di ordinamento. Considerare la pianificazione HTN come un'estensione di quella con ordinamento parziale ha l'ulteriore vantaggio di permetterci di continuare a usare la notazione che già conosciamo, invece di introdurne una completamente nuova. Cominceremo col descrivere la scomposizione di azioni in maggior dettaglio, quindi spiegheremo come l'algoritmo di pianificazione con ordinamento parziale dev'essere modificato per gestire le scomposizioni, e infine prenderemo in esame completezza, complessità e uso pratico.

## Rappresentare le scomposizioni di azioni

Le descrizioni generali delle azioni e i metodi di scomposizione sono memorizzati in una **libreria di piani**, da cui sono estratte e istanziate per soddisfare le necessità del piano in via di costruzione. Ogni metodo è un'espressione della forma *Scomponi(a, d)*. Questo significa che un'azione *a* può essere scomposta in *d*, un piano con ordinamento parziale come quelli descritti nel Paragrafo 11.3.

La costruzione di una casa è un esempio chiaro e concreto, per cui lo adotteremo per presentare il concetto di scomposizione. La Figura 12.5 mostra una possibile scomposizione dell'azione *CostruisciCasa* in quattro azioni di livello inferiore. La Figura 12.6 riporta la descrizione di alcune azioni del dominio, oltre alla scomposizione di *CostruisciCasa* così come apparirebbe nella libreria (che potrebbe contenere anche altre scomposizioni per la stessa azione).

L'azione *Inizio* della scomposizione aggiunge tutte le precondizioni delle azioni nel piano che non sono già richieste da altre azioni: queste prendono il nome di **precondizioni esterne**. Nel nostro esempio, le precondizioni esterne della scomposizione sono *Terreno* e *Denaro*. In modo analogo gli **effetti esterni**, che sono le precondizioni di *Fine*, sono tutti gli effetti delle azioni nel piano che non sono negati da altre azioni. Nel nostro esempio, gli effetti esterni di *CostruisciCasa* sono *Casa* e *-Denaro*. Alcuni pianificatori HTN distinguono anche tra **effetti primari** come *Casa* ed **effetti secondari** come *-Denaro*. Solo gli effetti primari possono essere usati per raggiungere obiettivi, ma entrambi i tipi possono causare conflitti con altre azioni; questo può ridurre drasticamente lo spazio di ricerca.<sup>2</sup>

libreria di piani

precondizioni esterne  
effetti esterni

effetti primari  
effetti secondari

<sup>2</sup> Può anche impedire la scoperta di piani imprevisti. Ad esempio, una persona minacciata dalla bancarotta può eliminare tutta la disponibilità liquida (ovvero, raggiungere *-Denaro*) comprando o costruendo una casa. Questo piano è utile, perché la legge impedisce la confisca della casa di abitazione da parte dei creditori.

*Azione(AcquistaTerreno, PRECOND: Denaro, EFFETTO: Terreno  $\wedge$   $\neg$ Denaro)*

*Azione(OttieniPrestito, PRECOND: CreditoConcesso, EFFETTO: Denaro  $\wedge$  Mutuo)*

*Azione(CostruisciCasa, PRECOND: Terreno, EFFETTO: Casa)*

*Azione(OttieniPermesso, PRECOND: Terreno, EFFETTO: Permesso)*

*Azione(IngaggiaCostruttore, EFFETTO: Contratto)*

*Azione(Costruzione, PRECOND: Permesso  $\wedge$  Contratto,*

*EFFETTO: CasaCostruita  $\wedge$   $\neg$ Permesso)*

*Azione(PagaCostruttore, PRECOND: Denaro  $\wedge$  CasaCostruita,*

*EFFETTO:  $\neg$ Denaro  $\wedge$  Casa  $\wedge$   $\neg$ Contratto)*

*Scomponi(CostruisciCasa,*

*Piano(PASSI: { $S_1$ : OttieniPermesso,  $S_2$ : IngaggiaCostruttore,  
 $S_3$ : Costruzione,  $S_4$ : PagaCostruttore})*

*ORDINAMENTO: {Inizio  $\prec$   $S_1 \prec S_3 \prec S_4 \prec$  Fine, Inizio  $\prec S_2 \prec S_3\}$ ,*

*COLLEGAMENTI: {Inizio  $\xrightarrow{\text{Terreno}}$   $S_1$ , Inizio  $\xrightarrow{\text{Denaro}}$   $S_4$ ,*

$S_1 \xrightarrow{\text{Permesso}} S_3$ ,  $S_2 \xrightarrow{\text{Contratto}} S_3$ ,  $S_3 \xrightarrow{\text{CasaCostruita}} S_4$ ,  
 $S_4 \xrightarrow{\text{Casa}} \text{Fine}$ ,  $S_4 \xrightarrow{\neg\text{Denaro}} \text{Fine}\})$

**Figura 12.6** Descrizioni di azioni per il problema di costruzione di una casa e scomposizione dettagliata dell'azione *CostruisciCasa*. Le descrizioni adottano un punto di vista semplificato riguardo al denaro e uno ottimistico sui costruttori.

Una scomposizione dovrebbe essere un'implementazione *corretta* dell'azione. Un piano  $d$  implementa correttamente un'azione  $a$  se  $d$  è un piano con ordinamento parziale completo e consistente che raggiunge gli effetti di  $a$  date le sue precondizioni. Naturalmente, una scomposizione ottenuta dall'esecuzione di un pianificatore risulterà corretta.

Una libreria di piani potrebbe contenere diverse scomposizioni per ogni azione di alto livello; ad esempio, per *CostruisciCasa* potrebbe esistere un'altra scomposizione che descrive un processo in cui l'agente costruisce una casa di pietra e zolle di torba con le mani nude. Ogni scomposizione dovrebbe essere un piano corretto, ma potrebbe avere precondizioni ed effetti aggiuntivi oltre a quelli specificati nella descrizione di alto livello dell'azione. Ad esempio, la scomposizione di *CostruisciCasa* nella Figura 12.5 richiede *Denaro* in aggiunta al *Terreno* e ha come effetto  $\neg$ *Denaro*. L'opzione di autocostruzione d'altra parte non richiede denaro, ma necessita di una grande scorta di *Pietra* e *Torba*, e potrebbe avere come risultato un *MalDiSchiena*.

Visto che un'azione di alto livello come *CostruisciCasa* potrebbe avere diverse possibili scomposizioni, è inevitabile che la sua descrizione STRIPS nasconde alcune delle precondizioni e degli effetti di tali scomposizioni. Le precondizioni di un'azione di alto livello dovrebbero essere l'*intersezione* delle precondizioni esterne delle sue scomposizioni, e la stessa cosa vale per gli effetti. Per dirla in un altro modo, si garantisce che le precondizioni e gli effetti di alto livello siano un sottoinsieme delle effettive precondizioni ed effetti di ogni implementazione primitiva.

Occorre notare che risultano occultati altri due tipi di informazioni: prima di tutto, la descrizione ad alto livello ignora completamente gli effetti **interni** delle scomposizioni. Ad esempio, la nostra scomposizione di *CostruisciCasa* ha gli effetti interni temporanei *Permesso* e *Contratto*.<sup>3</sup> In secondo luogo, la descrizione non specifica gli intervalli “dentro” l’attività in cui devono valere le precondizioni e gli effetti di alto livello. Ad esempio, la precondizione *Terreno* dev’essere vera (nel nostro modello molto approssimato) solo finché non si esegue *OttieniPermesso*, e *Casa* vale true solo dopo l’esecuzione di *PagaCostruttore*.

effetti interni

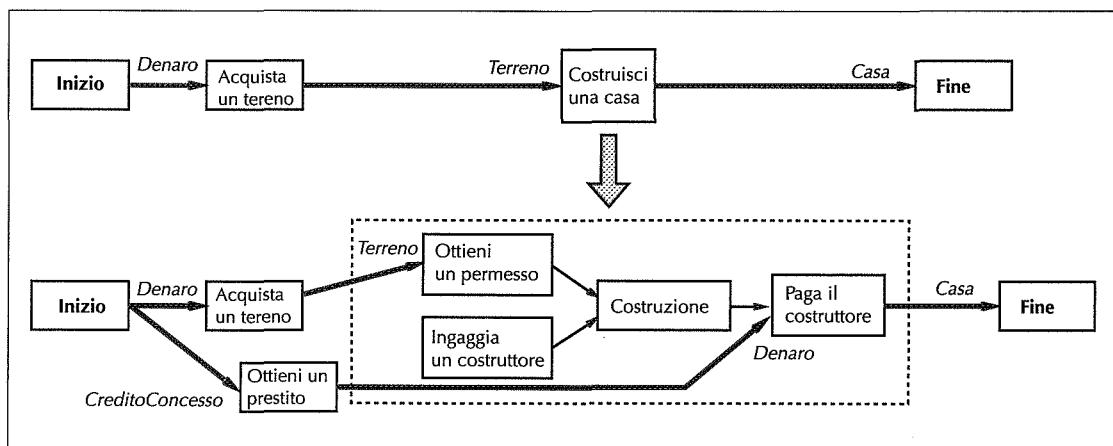
Nascondere informazione in questo modo è fondamentale per consentire alla pianificazione gerarchica di ridurre la complessità; dobbiamo essere capaci di ragionare sulle azioni ad alto livello senza preoccuparci della miriade di dettagli che riguardano le implementazioni. Per ottenere tutto questo, comunque, c’è un prezzo da pagare: ad esempio, potrebbero esserci dei conflitti tra le condizioni interne di due diverse azioni di alto livello, ma non c’è alcun modo di determinarlo dalle loro descrizioni. Questo problema ha importanti ripercussioni sugli algoritmi di pianificazione HTN; laddove le azioni primitive possono essere trattate come eventi istantanei dall’algoritmo di pianificazione, quelle di alto livello hanno un’espansione temporale al cui interno può accadere di tutto.

## Modificare il pianificatore per gestire le scomposizioni

Ora passiamo a considerare le estensioni di POP necessarie per incorporare la pianificazione HTN. Si deve modificare la funzione successore (v. pag. 496) in modo che permetta l’applicazione di metodi di scomposizione al piano parziale corrente  $P$ . I nuovi piani successore sono formati scegliendo per prima cosa alcune azioni non primitive  $a'$  in  $P$  e quindi, per ogni metodo  $Scomponi(a, d)$  della libreria tale che  $a$  e  $a'$  unificano con la sostituzione  $\theta$ , rimpiazzando  $a'$  con  $d' = \text{SUBST}(\theta, d)$ .

La Figura 12.7 mostra un esempio. In alto è rappresentato il piano  $P$  che ha lo scopo di ottenere una casa. L’azione di alto livello  $a' = \text{CostruisciCasa}$  è quella prescelta per la scomposizione. Dalla Figura 12.5 viene presa la scomposizione  $d$ , che sostituisce *CostruisciCasa*. A questo punto è introdotto un passo aggiuntivo,

<sup>3</sup> *Costruzione* nega *Permesso*, altrimenti lo stesso permesso potrebbe essere riutilizzabile per la costruzione di molte case. Sfortunatamente *Costruzione* non termina il *Contratto* perché, per far ciò, dobbiamo prima eseguire *PagaCostruttore*.



**Figura 12.7** Scomposizione di un’azione di alto livello all’interno di un piano esistente. L’azione *CostruisciCasa* è rimpiazzata dalla scomposizione della Figura 12.5. La precondizione esterna *Terreno* è fornita dal collegamento causale già esistente proveniente da *AcquistaTerreno*. La precondizione esterna *Denaro* rimane aperta dopo il passo di scomposizione, ragion per cui abbiamo aggiunto un’altra azione, *OttieniPrestito*.

*OttieniPrestito*, che risolve la nuova condizione aperta *Denaro* creata dal passo di scomposizione. Rimpiazzare un’azione con la sua scomposizione è un po’ come la chirurgia dei trapianti: dobbiamo estrarre il nuovo sottopiano dal suo pacchetto (i passi *Inizio* e *Fine*), inserirlo e cucire tutto nel modo corretto. Per far questo potrebbero esistere più modi. Precisamente, per ogni possibile scomposizione  $d'$ , si devono eseguire i seguenti passi.

1. Per prima cosa, rimuoviamo l’azione  $a'$  da  $P$ . Quindi, per ogni passo  $s$  della scomposizione  $d'$ , dobbiamo scegliere un’azione che ricopra il ruolo di  $s$  e aggiungerla al piano. Questa può essere una nuova istanza di  $s$  o un passo *preesistente*  $s'$  di  $P$  che unifica con  $s$ . Ad esempio, la scomposizione di un’azione *ProducVino* potrebbe suggerire di acquistare del terreno; è probabile che possiamo sfruttare l’azione *AcquistaTerreno* già presente nel piano. Questa prende il nome di *condivisione delle sottoattività*.

Nella Figura 12.7 non ci sono opportunità di condivisione, per cui abbiamo creato nuove istanze delle azioni. Una volta che le azioni sono state scelte, tutti i vincoli interni sono copiati da  $d'$ : ad esempio, nell’ordinamento *OttieniPermesso* viene prima di *Costruzione* e c’è un collegamento causale tra i due passi che fornisce la precondizione *Permesso* a *Costruzione*. Questo completa il compito di rimpiazzare  $a'$  con l’istanziamento di  $d\theta$ .

2. Il passo successivo è collegare i vincoli di ordinamento di  $a'$  nel piano originale ai passi in  $d'$ . Considerate un vincolo di forma  $B \prec a'$ . Come si devono ordinare  $B$  rispetto ai passi in  $d'$ ? La soluzione più ovvia è imporre che venga

prima di tutti i passi in  $d'$ , e per far questo è sufficiente sostituire ogni vincolo di forma  $Inizio \prec s$  in  $d'$  con un vincolo  $B \prec s$ . D'altra parte, quest'approccio potrebbe essere troppo stringente! Ad esempio, *AcquistaTerreno* deve venire prima di *CostruisciCasa*, ma nel piano espanso non c'è nessuna ragione perché venga prima di *IngaggiaCostruttore*. Imporre un ordinamento troppo stretto potrebbe impedire la scoperta di qualche soluzione. Di conseguenza, la soluzione migliore è associare a ogni vincolo di ordinamento la sua *ragione*; così facendo, al momento dell'espansione di un'azione di alto livello, i nuovi vincoli di ordinamento potranno essere rilassati il più possibile compatibilmente con la ragione del vincolo originale. Le stesse identiche considerazioni si applicano quando dobbiamo sostituire vincoli della forma  $a' \prec C$ .

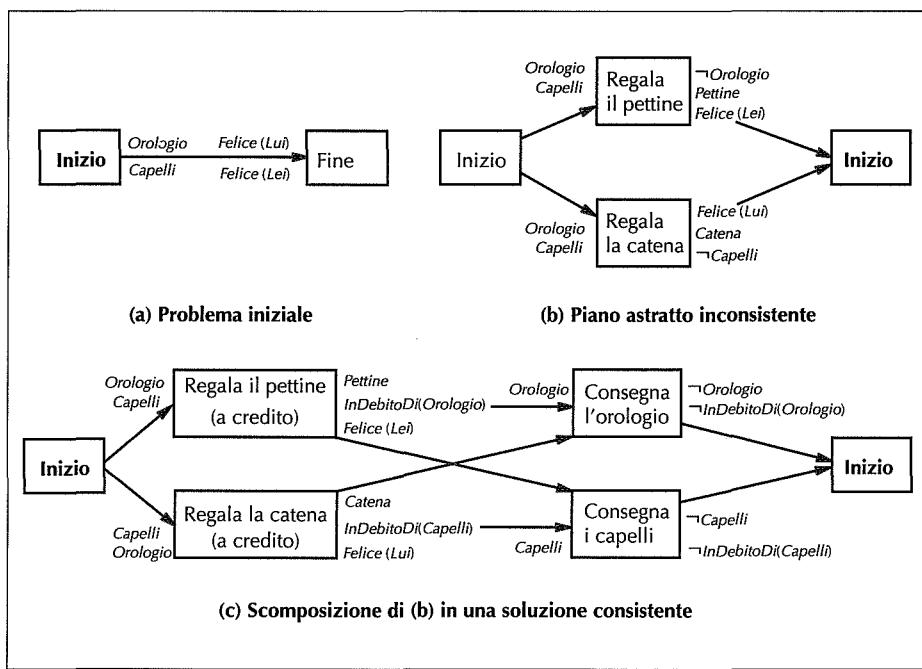
- Il passo finale è “cucire” i collegamenti causali.  $B \xrightarrow{p} a'$  nel piano originale va sostituito con un insieme di collegamenti causali da  $B$  a tutti i passi in  $d'$  con precondizioni  $p$  fornite dal passo *Inizio* della scomposizione  $d$ : in altre parole, tutti i passi in  $d'$  per cui  $p$  è una precondizione *esterna*. Nell'esempio, il collegamento causale *AcquistaTerreno*  $\xrightarrow{\text{Terreno}}$  *CostruisciCasa* è rimpiazzato dal collegamento *AcquistaTerreno*  $\xrightarrow{\text{Terreno}}$  *Permesso* (la precondizione *Denaro* di *PagaCostruttore* diventa una condizione aperta nella scomposizione, perché nessuna azione del piano originale fornisce *Denaro* a *CostruisciCasa*). Analogamente, ogni collegamento causale  $a' \xrightarrow{p} C$  nel piano va sostituito con un insieme di collegamenti a  $C$  da ogni passo in  $d'$  che fornisce  $p$  al passo *Fine* della scomposizione  $d$  (ovvero, dai passi in  $d'$  che hanno  $p$  come effetto *esterno*). In quest'esempio rimpiazziamo il collegamento *CostruisciCasa*  $\xrightarrow{\text{Casa}}$  *Fine* con *PagaCostruttore*  $\xrightarrow{\text{Casa}}$  *Fine*.

Questo completa le estensioni richieste per la generazione di scomposizioni nel contesto del pianificatore POP.<sup>4</sup>

È necessario apportare ulteriori modifiche all'algoritmo POP perché le azioni di alto livello *nascondono informazione* circa le loro implementazioni primitive finali. In particolare, l'algoritmo POP originale ritorna indietro con il backtracking se il piano corrente contiene un conflitto irrisolvibile, ovvero se un'azione confligge con un collegamento causale ma non può essere spostata prima o dopo di esso (v. Figura 11.9). Con le azioni di alto livello, al contrario, conflitti apparentemente irrisolvibili possono essere risolti *scomponendo* le azioni in conflitto e mescolando (attraverso il processo chiamato *interleaving*) i loro passi. La Figura

---

<sup>4</sup> In realtà ci sono ancora delle piccole modifiche da fare, necessarie per la gestione della risoluzione dei conflitti con azioni di alto livello; il lettore interessato può consultare i lavori citati alla fine del capitolo.



**Figura 12.8** Il problema del *Dono dei Magi*, ispirato alla storia di O. Henry, è un esempio di piano astratto inconsistente che nonostante tutto può essere scomposto in una soluzione consistente. La parte (a) illustra il problema: una coppia di poveri non possiede nulla tranne due cose preziose; lui ha un orologio d'oro, lei dei bellissimi capelli lunghi. Ognuno pianifica di acquistare un regalo per rendere felice l'altro: l'uomo decide di scambiare il suo orologio per comprare un pettine d'argento per i capelli di lei, la donna di vendere i capelli per procurare all'uomo una catena per il suo orologio. In (b) il piano parziale è inconsistente, perché non c'è modo di ordinare senza conflitto i passi astratti "Regala il pettine" e "Regala la catena". Naturalmente presumiamo che "Regala il pettine" abbia come precondizione *Capelli*, perché se la donna non ha più i capelli lunghi l'azione non avrà l'effetto inteso di renderla felice: analogo ragionamento vale per l'azione "Regala la catena" con l'orologio. In (c) scomponiamo il passo "Regala il pettine" con un metodo che potremmo definire "piano scagliato". Nel primo passo della scomposizione, l'uomo si impossessa del pettine e lo dà alla sua amata, essendosi accordato di consegnare l'orologio in un secondo tempo. Nel secondo passo, l'orologio viene effettivamente consegnato e l'obbligo è soddisfatto. Il passo "Regala la catena" è scomposto in modo simile. Finché i due passi di scambio dei regali sono ordinati prima dei passi di consegna degli oggetti impegnati, questa scomposizione risolve il problema. Notate che la soluzione si basa sul fatto che, in base alla definizione del problema, la felicità derivante dall'uso della catena con l'orologio e del pettine con i capelli perdura anche dopo che tali oggetti non sono più in possesso della coppia.

12.8 fornisce un esempio. Quindi potrebbe darsi che attraverso la scomposizione sia possibile ottenere un piano primitivo completo e consistente *anche quando non ne esiste uno di alto livello*. Questo significa che un pianificatore HTN completo deve rinunciare a molte opportunità di potatura che sarebbero disponibili a un pianificatore POP standard; alternativamente si può decidere di potare comunque, correndo il rischio di perdere qualche soluzione.

## Discussione

Cominciamo con le cattive notizie: la pianificazione HTN pura (in cui l'unico raffinamento possibile del piano è una scomposizione) è indecidibile, *anche se lo spazio degli stati sottostante è finito!* Questo potrebbe sembrare deprimente, dato che lo scopo stesso della pianificazione HTN è guadagnare efficienza. Le difficoltà sorgono perché le scomposizioni di azioni possono essere **ricorsive**: ad esempio, fare una passeggiata può essere implementato da un singolo passo seguito da una passeggiata. Per questo motivo i piani HTN possono essere arbitrariamente lunghi: in particolare potrebbe esserlo la soluzione HTN più breve, nel qual caso non ci sarebbe alcun modo di terminare la ricerca in un tempo finito. Ci sono comunque almeno tre motivi per rallegrarsi.

1. Possiamo abolire la ricorsione, che è effettivamente necessaria solo in pochi domini. In questo caso tutti i piani HTN risultano di lunghezza finita e possono essere enumerati.
2. Possiamo limitare la lunghezza dei piani a cui siamo interessati. Dato che lo spazio degli stati è finito, un piano che ha più passi di quanti sono gli stati *dove* includere un ciclo che visita due volte lo stesso stato. Escludere le soluzioni HTN di questo tipo è una piccola rinuncia e ci consente di controllare la ricerca.
3. Possiamo adottare un approccio ibrido che combina aspetti di POP e HTN. La pianificazione con ordinamento parziale da sola è sufficiente a determinare se un piano esiste, ragion per cui il problema ibrido sarebbe certamente decidibile.

Su questo terzo punto occorre fare un po' d'attenzione. POP può concatenare azioni primitive in modo arbitrario, per cui potremmo ritrovarci con soluzioni molto difficili da comprendere e prive della comoda organizzazione gerarchica dei piani HTN. Un buon compromesso potrebbe essere controllare la ricerca ibrida in modo che le scomposizioni di azioni siano preferite all'aggiunta di azioni nuove, ma non in misura tale che siano generati piani HTN arbitrariamente lunghi prima dell'aggiunta di alcuna azione primitiva. Per far questo si può usare una funzione di costo che offre uno "sconto" alle azioni introdotte mediante scomposizione; al crescere dello sconto la ricerca assomiglierà sempre più a una pianificazione HTN pura e la soluzione sarà maggiormente gerarchica. Solitamente i piani con struttura gerarchica sono molto più semplici da eseguire in contesti reali, e più facili da correggere quando qualcosa va storto.

Un'altra caratteristica importante dei piani HTN è la possibilità di condividere le sottoattività. Ricordate che con questo si intende che la stessa azione viene sfruttata per implementare due passi diversi nelle scomposizioni di un piano. Se disattiviamo la condivisione, ogni istanziazione di una scomposizione *d'* può essere fatta in un solo modo anziché molti, cosa che risulta in una notevole potatura

dello spazio di ricerca. Normalmente tale potatura consente di risparmiare tempo e, nel caso peggiore, porta a una soluzione leggermente più lunga di quella ottima. Alcuni casi, tuttavia, possono risultare problematici. Considerate ad esempio l'obiettivo "godersi la luna di miele e tirar su famiglia". La libreria di piani potrebbe scegliere "sposarsi e andare alla Hawaii" per il primo sotto-obiettivo e "sposarsi e fare due figli" per il secondo. Senza la possibilità di condividere le sottoattività il piano risultante includerà due distinti matrimoni, una soluzione spesso considerata indesiderabile.

Un esempio interessante dei costi e dei benefici della condivisione delle sottoattività si verifica nei compilatori ottimizzanti. Considerate il problema di compilare l'espressione  $\tan(x) - \sin(x)$ . La maggior parte dei compilatori unisce le due chiamate in modo banale: tutti i passi di  $\tan$  precedono tutti i passi di  $\sin$ . Ma considerate le seguenti serie di Taylor che approssimano  $\sin$  e  $\tan$ :

$$\tan x \approx x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315}; \quad \sin x \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}.$$

Un pianificatore HTN con divisione delle sottoattività potrebbe generare una soluzione più efficiente, implementando diversi passi del calcolo del seno mediante passi già esistenti del calcolo della tangente. La maggior parte dei compilatori non esegue questo tipo di condivisione tra procedure perché ci vuole troppo tempo per considerare tutti i piani risultanti, generando invece ogni sottopiano in modo indipendente ed eventualmente modificando il risultato con un ottimizzatore "peephole".

Viste le complicazioni aggiuntive causate dall'introduzione della scomposizione delle azioni, perché riteniamo che la pianificazione HTN possa essere efficiente? Le fonti reali di complessità sono difficili da analizzare in pratica, per cui considereremo un caso ideale. Supponiamo di voler costruire un piano con  $n$  azioni. Per un pianificatore in avanti nello spazio degli stati non gerarchico, con  $b$  azioni consentite in ogni stato, il costo sarà  $O(b^n)$ . Per il pianificatore HTN, supponiamo che la struttura della scomposizione sia molto regolare: ogni azione non primitiva avrà  $d$  possibili scomposizioni al livello inferiore, ognuna di  $k$  azioni. Vogliamo sapere il numero di alberi di scomposizione distinti con questa struttura. Ora, se al livello primitivo ci sono  $n$  azioni, allora il numero di livelli sotto la radice è  $\log_k n$ , per cui il numero di nodi interni di scomposizione è  $1 + k + k^2 + \dots + k^{\log_k n-1} = (n-1)/(k-1)$ . Ogni nodo interno ha  $d$  possibili scomposizioni, quindi il numero di alberi di scomposizione regolari è  $d^{(n-1)/(k-1)}$ . Esaminando questa formula, vediamo che mantenere  $d$  piccolo e  $k$  grande può farci risparmiare moltissimo: essenzialmente stiamo prendendo la radice  $k$ -esima del costo non gerarchico, se  $b$  e  $d$  sono confrontabili. D'altra parte, non sempre è possibile costruire una libreria di piani che ha un piccolo numero di lunghe scomposizioni e che ci permette comunque di risolvere tutti i problemi. Un altro modo di esprimere la stessa cosa è dire che le macro lunghe utilizzabili in una vasta gamma di problemi sono estremamente utili.

Un'altra ragione – forse più importante – per credere che la pianificazione HTN è efficiente è che funziona nella pratica. Quasi tutti i pianificatori per applicazioni su grande scala sono HTN perché questo approccio permette a un esperto umano di fornire conoscenza fondamentale sull'esecuzione di attività complesse, in modo tale che grandi piani possono essere costruiti con un piccolo sforzo computazionale. Per esempio, O-PLAN (Bell e Tate, 1985), che combina pianificazione HTN e scheduling, è stato utilizzato per sviluppare piani di produzione per la Hitachi. Un problema tipico comprende una linea di circa 350 prodotti diversi, 35 macchine di assemblaggio e più di 2000 operazioni. Il pianificatore ha generato schedule di 30 giorni divise in turni di 8 ore e contenenti milioni di passi.

La chiave della pianificazione HTN, quindi, è la costruzione di una libreria di piani che comprende metodi conosciuti per implementare azioni complesse di alto livello. Un metodo per costruire tale libreria è *apprendere* i metodi attraverso l'esperienza nella risoluzione di problemi: dopo lo sforzo di costruire un piano da zero, l'agente può salvarlo nella libreria come metodo per implementare l'azione di alto livello definita dall'attività. In questo modo l'agente può diventare col tempo sempre più competente, man mano che nuovi metodi sono costruiti sulla base di quelli vecchi. Un aspetto importante di questo processo di apprendimento è la capacità di *generalizzare* i metodi costruiti, eliminando i dettagli specifici di una particolare istanza dal problema (come il nome del costruttore o l'indirizzo del terreno) e tenendo solo gli elementi chiave del piano. Le tecniche per effettuare questo tipo di generalizzazione sono descritte nel Capitolo 19, nel 2° volume. Ci sembra difficile credere che gli esseri umani possano essere così competenti senza utilizzare un meccanismo analogo.

## 12.3 Pianificazione e azione in ambienti non deterministici

Fin qui abbiamo considerato esclusivamente i domini tipici della **pianificazione classica**: completamente osservabili, statici e deterministici. Inoltre, abbiamo sempre dato per scontato che le descrizioni delle azioni fossero corrette e complete. In queste circostanze, un agente può determinare il piano e poi eseguirlo “a occhi chiusi”. In un ambiente incerto, invece, deve sfruttare le proprie percezioni per scoprire cosa succede durante l'esecuzione del piano ed eventualmente, se si verificano situazioni inaspettate, modificarlo o sostituirlo completamente.

Gli agenti devono essere capaci di gestire sia informazione *incompleta* che *inesatta*. L'incompletezza scaturisce dal fatto che il mondo è parzialmente osservabile, non deterministico o entrambe le cose insieme. Ad esempio, lo sportello dell'armadietto potrebbe essere chiuso a chiave o no; nel caso sia chiuso una delle mie chiavi potrebbe aprirlo o no; e io potrei essere consci o meno di queste fonti di incompletezza nella mia conoscenza. Quindi, il mio modello del mondo è debole,

indeterminatezza limitata

indeterminatezza illimitata

pianificazione senza sensori

pianificazione condizionale

azioni di percezione

ma corretto. D'altra parte, l'inesattezza sorge nel momento in cui il mondo non corrisponde al modello che mi sono fatto; ad esempio potrei *credere* che le mie chiavi aprano l'armadietto, ma mi sbaglierei se hanno appena cambiato tutte le serrature. Senza la capacità di gestire informazione inesatta, un agente potrebbe ritrovarsi intelligente quanto lo scarabeo stercorario (v. pag. 52), che cerca di chiudere la sua tana con una pallina che gli è stata appena sottratta dalle zampe.

La possibilità di avere conoscenza completa o corretta dipende dalla *quantità* di indeterminatezza nel mondo. In condizioni di **indeterminatezza limitata** le azioni possono avere effetti impredicibili, ma quelli possibili possono essere tutti elencati negli assiomi di descrizione delle azioni. Ad esempio, quando tiriamo una moneta, è ragionevole dire che il risultato sarà *Testa* o *Croce*. Un agente può adattarsi all'indeterminatezza limitata formulando piani che funzionano in tutte le possibili circostanze. In condizioni di **indeterminatezza illimitata**, al contrario, l'insieme di possibili precondizioni o effetti è sconosciuto oppure troppo grande per essere enumerato completamente. Questo è il caso di ambienti molto complessi o dinamici come la guida, la pianificazione economica e la strategia militare. Un agente può gestire un'indeterminatezza illimitata solamente se è pronto a rivedere i propri piani e/o la base di conoscenza. L'indeterminatezza illimitata è strettamente imparentata al problema di **qualificazione** discusso nel Capitolo 10: l'impossibilità, nel mondo reale, di elencare *tutte* le precondizioni richieste affinché un'azione abbia l'effetto previsto.

Ci sono quattro metodi di pianificazione per gestire l'indeterminatezza: i primi due sono adatti a quella limitata, gli altri due a quella illimitata.

- ◆ **Pianificazione senza sensori:** chiamata anche **pianificazione conformante**, costruisce piani sequenziali standard che dovranno essere eseguiti senza ricorrere alla percezione. L'algoritmo di pianificazione senza sensori dovrà assicurarsi che il piano raggiunga l'obiettivo *in tutte le possibili circostanze*, indipendentemente dal vero stato iniziale e dall'effettivo risultato delle azioni. Questa tecnica si basa sulla **coercizione**, ovvero l'idea che il mondo possa essere forzato in un determinato stato anche se l'agente possiede un'informazione solo parziale sullo stato corrente. La coercizione non è sempre possibile, ragion per cui la pianificazione senza sensori è spesso inapplicabile. Abbiamo descritto nel Capitolo 3 la risoluzione di problemi senza sensori, basata sulla ricerca nello spazio degli stati-credenza.
- ◆ **Pianificazione condizionale:** noto anche come **pianificazione di contingenza**, quest'approccio gestisce l'indeterminatezza limitata costruendo un piano condizionale con più rami per le diverse contingenze che potrebbero verificarsi. Proprio come nella pianificazione classica, l'agente prima calcola il piano e poi lo esegue: determina quale parte del piano eseguire includendovi **azioni di percezione** per verificare le condizioni appropriate. Nel dominio del trasporto aereo, per esempio, potremmo avere piani che includono "controlla se l'aeroporto SFO è aperto. Se è così, vola lì; altrimenti vola a Oakland". La pianificazione condizionale è trattata nel Paragrafo 12.4.

- ◆ **Monitoraggio di esecuzione e ripianificazione:** in quest'approccio l'agente può usare una qualsiasi delle tecniche di pianificazione sopracitate per costruire un piano, ma utilizza anche il **monitoraggio di esecuzione** per giudicare se esso può governare la situazione corrente o se necessita di una revisione. Quando qualcosa va storto, si può ricorrere a una **riplanificazione**. In questo modo l'agente può gestire l'indeterminatezza illimitata. Anche nel caso in cui un agente con ripianificazione non avesse previsto la possibilità che l'aeroporto SFO potesse essere chiuso, riconoscerà tale situazione non appena si verifica e invocherà nuovamente il pianificatore per trovare un nuovo cammino verso l'obiettivo. Gli agenti con ripianificazione sono trattati nel Paragrafo 12.5.
- ◆ **Pianificazione continua:** tutti i pianificatori considerati fin qui raggiungono un obiettivo e poi si fermano; un pianificatore continuo è progettato per durare un intero ciclo di vita. L'agente è in grado di gestire circostanze inaspettate nell'ambiente, anche se queste si verificano nel bel mezzo della costruzione di un piano; inoltre può abbandonare alcuni obiettivi e creare nuovi attraverso la **formulazione di obiettivi**. La pianificazione continua è trattata nel Paragrafo 12.6.

monitoraggio di esecuzione e ripianificazione

pianificazione continua

Consideriamo un esempio per chiarire le differenze tra i vari tipi di agente. Il problema è il seguente: dato uno stato iniziale con una sedia, un tavolo e alcune latte di vernice, in cui tutti gli oggetti sono di un colore sconosciuto, raggiungere uno stato in cui la sedia e il tavolo sono dello stesso colore.

Un agente di **pianificazione classica** non potrebbe gestire questo problema, perché lo stato iniziale non è specificato completamente: non sappiamo neppure di che colore sono i mobili.

Un agente di **pianificazione senza sensori** deve trovare un piano che funziona senza richiedere alcuna percezione durante l'esecuzione. La soluzione è aprire una latta di vernice qualsiasi e dipingere sia la sedia che il tavolo, obbligandoli attraverso la **coercizione** ad avere lo stesso colore (anche se l'agente non sa quale sia). La coercione è appropriata nei casi in cui le proposizioni sono costose o impossibili da percepire. Ad esempio, i dottori spesso prescrivono antibiotici ad ampio spettro invece di usare il piano condizionale di effettuare un esame del sangue, aspettare i risultati e prescrivere un antibiotico specifico. La ragione sta nel fatto che solitamente i ritardi e i costi di un esame del sangue sono troppo alti.

Un agente di **pianificazione condizionale** può generare un piano migliore: per prima cosa percepire il colore di tavolo e sedia; se sono già identici il piano è fatto. In caso contrario, si possono percepire le etichette sulle latte di vernice; se c'è una latta dello stesso colore di uno dei mobili, si può verniciare l'altro. Altrimenti si dipingeranno entrambi i mobili di un colore qualsiasi.

Un agente di **riplanificazione** potrebbe generare lo stesso piano dell'agente di pianificazione condizionale, oppure potrebbe generare inizialmente meno rami riempiendo le lacune durante l'esecuzione man mano che se ne presenta la neces-

sità. In aggiunta, sarebbe capace di gestire eventuali inesattezze nella descrizione delle azioni. Ad esempio, supponiamo che si ritenga che l'azione *Dipingi(oggetto, colore)* abbia l'effetto deterministico *Colore(oggetto, colore)*. Un pianificatore condizionale darebbe per scontato che dopo l'esecuzione dell'azione l'effetto sia in atto, ma l'agente di ripianificazione potrebbe verificarlo, e in caso di necessità (forse perché il mobile non è stato pitturato completamente) ripianificare un'azione di pittura aggiuntiva. Torneremo su quest'esempio a pag. 559.

Un agente di pianificazione continua, oltre a gestire eventi inaspettati, è in grado di rivedere i piani in modo appropriato: se aggiungiamo l'obiettivo di cenare sul tavolo, l'agente potrà rimandare l'azione di pittura.

Nel mondo reale gli agenti utilizzano una combinazione di approcci. Le macchine hanno l'airbag e contengono ruote di scorta, che sono manifestazioni fisiche dei rami condizionali di un piano progettato per gestire schianti o forature; d'altra parte, la maggior parte dei guidatori non considera mai queste possibilità, così si può dire che rispondono agli incidenti come agenti di ripianificazione. In generale, gli agenti creano piani condizionali solo per le contingenze che hanno conseguenze importanti e una probabilità non trascurabile di andare nel verso sbagliato. Così, un guidatore che progetti l'attraversamento del Sahara farà bene a considerare esplicitamente la possibilità di guasti, laddove un salto al supermercato richiede meno pianificazione anticipata.

Gli agenti che descriviamo in questo capitolo sono progettati per gestire l'indeterminatezza, ma non sono capaci di fare compromessi tra la probabilità di successo e il costo della costruzione del piano. Il Capitolo 16, nel 2° volume, fornisce ulteriori strumenti per trattare questi problemi.

## 12.4 Pianificazione condizionale

La pianificazione condizionale è un modo di gestire l'incertezza verificando ciò che sta effettivamente succedendo nell'ambiente in corrispondenza di punti predeterminati del piano. È più facile spiegare la pianificazione condizionale nel caso di ambienti completamente osservabili, ragion per cui cominceremo con quelli. Il caso parzialmente osservabile è più difficile, ma più interessante.

### Pianificazione condizionale in ambienti completamente osservabili

La completa osservabilità significa che l'agente conosce sempre lo stato corrente. Se l'ambiente non è deterministico, comunque, l'agente non potrà predire il *risultato* delle sue azioni. Un agente di pianificazione condizionale gestisce il non determinismo includendo nel piano (al momento della sua costruzione) dei passi condizionali in cui verificherà lo stato dell'ambiente (durante l'esecuzione) per decidere cosa fare. Il problema si riduce quindi alla costruzione dei piani condizionali.

Come esempio useremo il venerabile **mondo dell'aspirapolvere**, il cui spazio degli stati, nel caso deterministico, è raffigurato a pag. 89. Ricorderete che le azioni disponibili sono *Sinistra*, *Destra* e *Aspira*. Per definire gli stati occorreranno delle proposizioni: sia *ASin(ADx)* vero se l'agente si trova nel riquadro sinistro(destro),<sup>5</sup> e sia *PulitoSin(PulitoDx)* vero se il riquadro sinistro(destro) è pulito. La prima cosa da fare è arricchire il linguaggio STRIPS per consentire l'espressione del nondeterminismo. Per far questo permetteremo alle azioni di avere **effetti disgiuntivi**, il che significa che un'azione potrà avere due o più esiti diversi ogni volta che viene eseguita. Ad esempio, supponiamo che il movimento a *Sinistra* possa talvolta fallire. Allora la normale descrizione di azione

*Azione(Sinistra, PRECOND: ADx , EFFETTO: ASin  $\wedge$   $\neg$  ADx)*

dev'essere modificata per includere un effetto disgiuntivo:

*Azione(Sinistra, PRECOND: ADx , EFFETTO: ASin  $\vee$  ADx) .* (12.1)

Sarà anche utile permettere alle azioni di avere **effetti condizionali**, laddove l'effetto dell'azione dipende dallo stato in cui è eseguita. Gli effetti condizionali appaiono nella sezione **EFFETTO** di un'azione e hanno la sintassi “when *<condizione>* : *<effetto>*”. Ad esempio, per modellare l'azione *Aspira* scriveremo

*Azione(Aspira, PRECOND: ,  
EFFETTO: (when ASin : PulitoSin)  $\wedge$  (when ADx : PulitoDx)) .*

Gli effetti condizionali non introducono indeterminatezza, ma possono aiutare a modellarla. Ad esempio, supponiamo di possedere un aspirapolvere infido che sporadicamente scarica dello sporco nel riquadro in cui si è appena spostato, ma solo se il riquadro è pulito. Questo può essere modellato con una descrizione come

*Azione(Sinistra, PRECOND: ADx,  
EFFETTO: ASin  $\vee$  (ASin  $\wedge$  when PulitoSin:  $\neg$  PulitoSin)) ,*

che è sia disgiuntiva che condizionale.<sup>6</sup> Per creare piani condizionali sono necessari **passi condizionali**. Per scriverli useremo la sintassi “if *<test>* then *piano\_A* else *piano\_B*”, dove *<test>* è una funzione booleana delle variabili di stato. Ad esempio, un passo condizionale nel mondo dell'aspirapolvere potrebbe essere “if *ASin  $\wedge$  PulitoSin* then *Destra* else *Aspira*”. L'esecuzione di un passo simile procede nel modo intuibile. Nidificando passi condizionali, i piani diventano alberi.

effetti disgiuntivi

effetti condizionali

passi condizionali

<sup>5</sup> Naturalmente, *ADx* è vero se e solo se è vero  $\neg$  *ASin* e viceversa. Useremo due proposizioni distinte per facilitare la leggibilità.

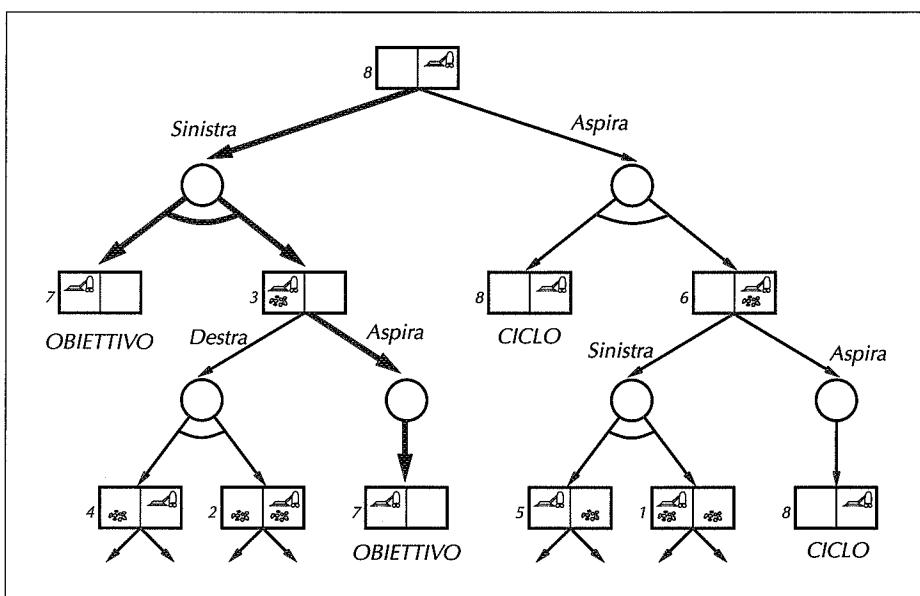
<sup>6</sup> L'effetto condizionale *when PulitoSin:  $\neg$  PulitoSin* potrebbe sembrare un po' strano. Ricordate comunque che qui *PulitoSin* si riferisce alla situazione *prima* dell'azione e  $\neg$  *PulitoSin* alla situazione *dopo* l'azione.

giochi contro la natura

Noi desideriamo che i piani condizionali funzionino *indipendentemente dal risultato effettivo delle azioni*. Abbiamo già incontrato questo problema in forma differente. Nei giochi a due giocatori (v. Capitolo 6) volevamo trovare mosse che fossero vincenti *indipendentemente dalle mosse dell'avversario*. Per questa ragione, i problemi di pianificazione non deterministica sono spesso chiamati **giochi contro la natura**.

Consideriamo un esempio specifico nel mondo dell'aspirapolvere. Lo stato iniziale ha il robot nel riquadro destro di un mondo pulito; dato che l'ambiente è completamente osservabile l'agente conosce la descrizione completa dello stato,  $ADx \wedge PulitoSin \wedge PulitoDx$ . Lo stato obiettivo ha il robot nel riquadro sinistro di un mondo pulito. Questo problema sarebbe decisamente banale se non fosse per l'aspirapolvere "doppio Murphy" che talvolta deposita sporco quando entra in un riquadro pulito o quando si applica l'azione *Aspira* in un riquadro pulito.

Un "albero di gioco" per quest'ambiente è mostrato nella Figura 12.9. Il robot esegue azioni nei nodi "stato" dell'albero, la natura decide quali saranno le conseguenze nei nodi "di possibilità", rappresentati con dei cerchietti. Una soluzione è costituita da un sottoalbero che (1) ha un nodo obiettivo in ogni foglia, (2) specifica un'azione in ognuno dei nodi "stato", (3) include ogni possibile esito delle azioni in ognuno dei nodi "di possibilità". La soluzione è evidenziata in grassetto.



**Figura 12.9** I primi due livelli di un albero di ricerca per il mondo dell'aspirapolvere "doppio Murphy". I nodi stato sono nodi OR in cui dev'essere scelta un'azione da eseguire. I nodi di possibilità, rappresentati con dei cerchietti, sono nodi AND in cui dev'essere determinato ogni esito possibile, come indicato dall'arco che collega i rami uscenti. La soluzione è evidenziata in grassetto.

nella figura e corrisponde al piano [*Sinistra, if ASin  $\wedge$  PulitoSin  $\wedge$  PulitoDx then [ ] else Aspira*]. Per adesso, dato che stiamo usando un pianificatore nello spazio degli stati, i test nei passi condizionali saranno descrizioni complete di stato.

Per risolvere i giochi abbiamo usato l'**algoritmo minimax** (v. Figura 6.3). Nella pianificazione condizionale si apportano tipicamente due modifiche. Per prima cosa, i nodi MAX e MIN possono diventare nodi OR e AND. Intuitivamente, il piano deve eseguire *una qualche* azione in ogni stato raggiunto, ma deve gestire *ogni* possibile esito dell'azione eseguita. In secondo luogo, l'algoritmo deve restituire un piano condizionale anziché una singola mossa. In un nodo OR, il piano consiste semplicemente nell'azione selezionata, seguita da qualsiasi cosa venga dopo. In un nodo AND, il piano è una serie di passi if-then-else nidificati che specificano un sottopiano per ogni possibile esito dell'azione; i test in quei passi sono costituiti da descrizioni complete di stato.<sup>7</sup>

Formalmente, lo spazio di ricerca che abbiamo definito è un **grafo AND-OR**. Abbiamo già incontrato questi grafi nel Capitolo 7, parlando di inferenza applicata alle clausole proposizionali di Horn. Qui i rami sono azioni e non passi di inferenza logica, ma l'algoritmo è lo stesso. La Figura 12.10 riporta un algoritmo ricorsivo in profondità per la ricerca su grafi AND-OR.

Un aspetto chiave dell'algoritmo è il modo in cui gestisce i cicli, che si verificano spesso nei problemi di pianificazione non deterministica (ad esempio quando capita che un'azione non abbia effetto, o ne abbia uno diverso da quello previsto). Se lo stato corrente è identico a uno stato sul cammino dalla radice, l'algoritmo restituisce un fallimento. Questo non vuol dire che non ci sia *alcuna* soluzione che passa dallo stato corrente; significa semplicemente che se una soluzione non ciclica *esiste*, dev'essere raggiungibile dall'occorrenza precedente dello stato corrente, quindi l'occorrenza successiva può essere scartata. Questa verifica assicura che l'algoritmo termini in ogni spazio degli stati finito, perché ogni cammino deve raggiungere un obiettivo, un vicolo cieco o uno stato ripetuto. Notate che l'algoritmo non verifica se lo stato corrente è la ripetizione di uno stato presente su qualche *altro* cammino dalla radice: l'Esercizio 12.15 approfondisce la questione.

I piani restituiti da RICERCA-GRAFO-AND-OR contengono passi condizionali che sottopongono a test l'intera descrizione dello stato per decidere quale ramo seguire. In molti casi possiamo cavarsela con test meno esaustivi. Ad esempio, la soluzione evidenziata nella Figura 12.9 potrebbe essere scritta semplicemente come [*Sinistra, if PulitoSin then [ ] else Aspira*] perché un singolo test, *PulitoSin*, è sufficiente a dividere gli stati del nodo AND in due insiemi singoletto, cosicché dopo il test l'agente conosce perfettamente lo stato in cui si trova. In effetti, una serie

<sup>7</sup> Piani siffatti potrebbero anche essere scritti per mezzo di un costrutto **case**.

---

```

function RICERCA-GRAFO-AND-OR(problema) returns un piano condizionale, o il fallimento
 RICERCA-OR(STATO-INIZIALE[problema], problema, [])

function RICERCA-OR(stato, problema, cammino) returns un piano condizionale, o il fallimento
 if TEST-OBIETTIVO[problema] (stato) then return il piano vuoto
 if stato si trova sul cammino then return fallimento
 for each azione, insieme_di_stati in SUCCESSORI[problema] (stato) do
 piano \leftarrow RICERCA-AND(insieme_di_stati, problema, [stato | cammino])
 if piano \neq fallimento then return [azione | piano]
 return fallimento

function RICERCA-AND(insieme_di_stati, problema, cammino) returns un piano condizionale,
 o il fallimento
 for each si in insieme_di_stati do
 pianoi \leftarrow RICERCA-OR(si, problema, cammino)
 if pianoi = fallimento then return fallimento
 return [if s1 then piano1 else if s2 then piano2 else . . . if sn-1 then pianon-1 else pianon]

```

---

**Figura 12.10** Un algoritmo per la ricerca su grafi AND-OR generati da ambienti non deterministici. Presumiamo che SUCCESSORI restituisca una lista di azioni, ognuna associata a un *insieme* di possibili esiti. Lo scopo è trovare un piano condizionale che raggiunge uno stato obiettivo in tutte le circostanze.

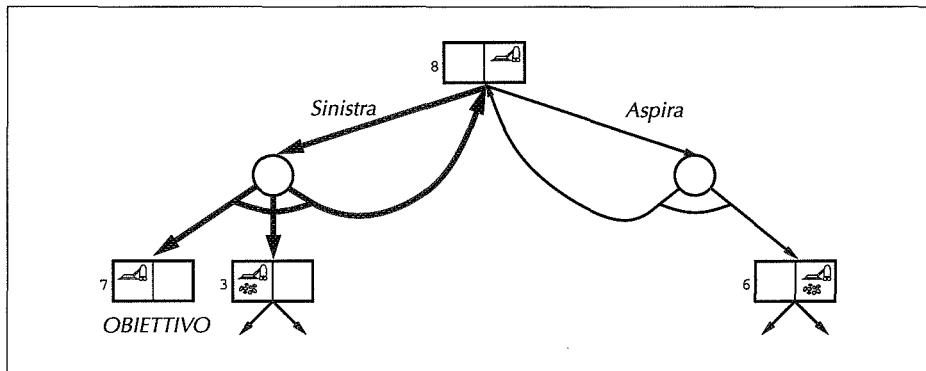
di test if-then-else su variabili singole è sempre sufficiente a dividere un insieme di stati in singoletti, *a patto che lo stato sia completamente osservabile*. Possiamo quindi restringere i test su variabili singole senza perdere generalità.

Un'ultima complicazione sorge spesso nei domini non deterministici: le cose potrebbero non funzionare la prima volta, e dovremmo riprovare l'esecuzione. Considerate ad esempio un aspirapolvere "triplo Murphy" che (oltre ai difetti che abbiamo già descritto) può talvolta rifiutarsi di muoversi quando gli viene ordinato. *Sinistra* potrà quindi avere l'effetto disgiuntivo *ASin*  $\vee$  *ADx*, come nell'Equazione (12.1). Ora non possiamo più essere sicuri che il piano [*Sinistra*, **if** *PulitoSin* **then** [] **else** *Aspira*] funzioni. La Figura 12.11 mostra una parte del grafo di ricerca; palesemente non ci sono più soluzioni acicliche, e RICERCA-GRAFO-AND-OR restituirebbe un fallimento. Esiste tuttavia una **soluzione ciclica**, che consiste nel continuare a provare *Sinistra* finché non funziona. Possiamo esprimere questa soluzione aggiungendo un'etichetta per indicare una porzione del piano a cui ritornare, invece di riscrivere il piano stesso. Così, la nostra soluzione ciclica si scriverà

[*L<sub>1</sub>* : *Sinistra*, **if** *ADx* **then** *L<sub>1</sub>* **else if** *PulitoSin* **then** [] **else** *Aspira*]

soluzione ciclica

etichetta



**Figura 12.11** Il primo livello di un grafo di ricerca per il mondo dell'aspirapolvere “triplo Murphy”, in cui abbiamo indicato i cicli esplicitamente. Tutte le soluzioni di questo problema sono piani ciclici.

(una sintassi migliore per la parte ciclica del piano sarebbe “*while ADx do Sinistra*”). Le modifiche da apportare a RICERCA-GRAFO-AND-OR sono trattate nell’Esercizio 12.16. La cosa più importante da capire è che un ciclo nello spazio degli stati che ritorna allo stato  $L$  si traduce in un ciclo nel piano che ritorna al punto in cui cominciava l’esecuzione del sottopiano per lo stato  $L$ .

Ora abbiamo la capacità di sintetizzare piani complessi che assomigliano a programmi con blocchi condizionali e cicli. Sfortunatamente, i cicli sono potenzialmente *infiniti*: non c’è nulla nella rappresentazione delle azioni del mondo di Murphy triplo che ci assicuri che l’azione *Sinistra* abbia prima o poi successo. I piani ciclici sono quindi meno desiderabili di quelli aciclici, ma possono comunque essere considerati soluzioni, a patto che ogni foglia sia uno stato obiettivo e che da ogni punto del piano sia raggiungibile una foglia.

## Pianificazione condizionale in ambienti parzialmente osservabili

Nel precedente sottoparagrafo ci siamo occupati degli ambienti completamente osservabili, che hanno il vantaggio che i test condizionali possono porre qualsiasi tipo di domanda ed essere certi di ottenere una risposta. Nel mondo reale, è molto più frequente che l’osservabilità sia solo parziale. Nello stato iniziale di un problema di pianificazione parzialmente osservabile, l’agente conosce solo una parte dello stato corrente. Il modo più semplice di modellare questa situazione è dire che lo stato iniziale appartiene a un *insieme di stati*; questo è un modo di descrivere lo *stato-credenza* iniziale dell’agente.<sup>8</sup>

<sup>8</sup> Questi concetti sono introdotti nel Paragrafo 3.6, che vi consigliamo di leggere prima di proseguire.

Supponiamo che un agente nel mondo dell'aspirapolvere sappia di trovarsi nel riquadro destro e che il riquadro è pulito, ma non possa percepire la presenza o l'assenza di sporco in altri riquadri. *Per quanto ne sa* potrebbe trovarsi in uno di due stati possibili: il riquadro sinistro potrebbe essere pulito o sporco. Questo stato-credenza è indicato con la lettera *A* nella Figura 12.12. La figura mostra una parte del grafo AND-OR per il mondo dell'aspirapolvere “doppio Murphy alternativo”, in cui l'agente può talvolta lasciare sporco dietro a sé quando abbandona un riquadro pulito.<sup>9</sup> Se il mondo fosse completamente osservabile, l'agente potrebbe costruire una soluzione ciclica del tipo “continuo a muovermi a sinistra e a destra, aspirando lo sporco ovunque ce ne sia, finché entrambi i riquadri sono puliti e mi trovo nel riquadro sinistro” (v. Esercizio 12.16.). Sfortunatamente, con una percezione solo locale dello sporco questo piano è inattuabile, perché non è possibile determinare il valore di verità del test “entrambi i riquadri sono puliti”.

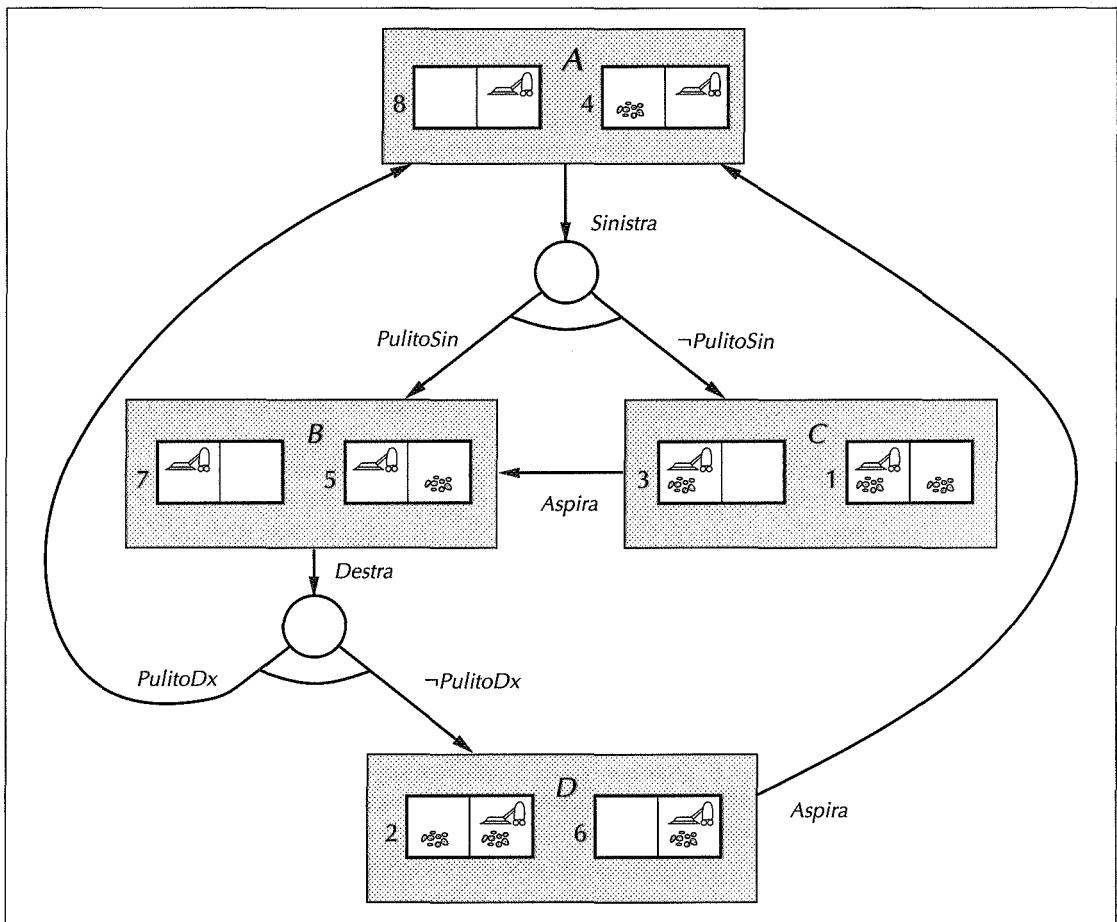
Consideriamo la costruzione del grafo AND-OR. Dallo stato credenza *A*, mostriamo il risultato del movimento a *Sinistra* (le altre azioni non hanno senso). Data che l'agente può lasciare sporco dietro di sé, i due mondi possibili iniziali diventano quattro, come illustrato da *B* e *C*. I mondi formano due stati-credenza distinti, classificati in base all'informazione disponibile ai sensori.<sup>10</sup> In *B*, l'agente sa che è *PulitoSin*; in *C* sa che  $\neg$ *PulitoSin*. Da *C*, la pulizia dello sporco porta l'agente in *B*. Da *B*, lo spostamento a *Destra* potrebbe lasciare sporco nel riquadro o no, per cui ci sono ancora quattro mondi possibili, suddivisi in base alla conoscenza dell'agente di *PulitoDx* (si ritorna in *A*) o di  $\neg$ *PulitoDx* (si va nello stato-credenza *D*).

In definitiva, gli ambienti non deterministici e parzialmente osservabili danno origine a un grafo AND-OR di stati-credenza. È quindi possibile formulare piani condizionali usando lo stesso algoritmo del caso completamente osservabile, e cioè RICERCA-GRAFO-AND-OR. Un altro modo di esprimere quello che succede è dire che gli *stati-credenza* dell'agente sono *sempre* completamente osservabili: l'agente sa ciò che conosce. La risoluzione dei problemi “standard” completamente osservabili diventa così un semplice caso speciale in cui ogni stato-credenza è un insieme singoletto che contiene esattamente uno stato fisico.

Abbiamo finito? Nient'affatto! Dobbiamo ancora decidere come devono essere rappresentati gli stati-credenza, come funziona la percezione e come debbano essere scritte le descrizioni delle azioni.

<sup>9</sup> I genitori di bambini piccoli avranno familiarità con questo fenomeno. Come al solito, ci scusiamo con gli altri.

<sup>10</sup> Notate che *non sono* classificati in base allo sporco eventualmente lasciato dall'agente durante il movimento: nello spazio degli stati-credenza le ramificazioni sono causate dalle differenze nella conoscenza, non negli effetti fisici delle azioni.



**Figura 12.12** Parte del grafo AND-OR per il mondo dell'aspirapolvere "doppio Murphy alternativo", in cui talvolta l'agente lascia dello sporco dietro a sé quando esce da un riquadro pulito. L'agente non può percepire la presenza di sporco in altri riquadri.

Per gli stati-credenza le principali possibilità sono tre.

1. Insiemi di descrizioni complete di stato. Ad esempio, lo stato-credenza iniziale della Figura 12.12 è

$$\{(ADx \wedge PulitoDx \wedge PulitoSin), (ADx \wedge PulitoDx \wedge \neg PulitoSin)\}.$$

Questa rappresentazione è facile, ma molto costosa: se uno stato è definito da  $n$  proposizioni booleane, uno stato-credenza può contenere  $O(2^n)$  descrizioni di stato fisico, ognuna di dimensione  $O(n)$ . Ogni volta che l'agente conosce solo una frazione delle proposizioni si avranno stati-credenza esponenzialmente grandi: meno l'agente conosce, più sono gli stati in cui potrebbe trovarsi.

2. Formule logiche che catturano esattamente l'insieme di mondi possibili nello stato-credenza. Ad esempio, lo stato iniziale può essere scritto come

$$ADx \wedge PulitoDx.$$

È chiaro che ogni stato-credenza può essere catturato per mezzo di esattamente una formula logica: se necessario possiamo ricorrere alla disgiunzione di tutte le descrizioni di stato congiuntive, ma il nostro esempio dimostra che possono esistere forme più compatte.

Un inconveniente delle formule logiche sta nel fatto che diverse formule logicamente equivalenti possono descrivere lo stesso stato-credenza, quindi il controllo degli stati ripetuti nell'algoritmo di ricerca su grafo può richiedere la capacità di dimostrare teoremi. Per evitare ciò serve una rappresentazione *canonica* delle formule in cui ogni stato-credenza corrisponde a esattamente una formula.<sup>11</sup> Una possibile rappresentazione è costituita da una congiunzione di letterali ordinati per nome, come  $ADx \wedge PulitoDx$ . Questa non è altro che la rappresentazione standard degli stati se si adotta l'**ipotesi del mondo aperto** (v. Capitolo 11). Non tutte le formule logiche possono essere scritte in questa forma – ad esempio, non c'è modo di rappresentare  $ASin \vee PulitoDx$  – ma è comunque possibile gestire molti domini.

3. **Proposizioni di conoscenza** che descrivono la conoscenza dell'agente (v. anche Paragrafo 7.7). per lo stato iniziale avremo

$$K(ADx) \wedge K(PulitoDx).$$

Qui  $K$  sta per “conosce” e  $K(P)$  significa che l'agente sa che  $P$  è vero.<sup>12</sup> Con le proposizioni di conoscenza si adotta l'**ipotesi del mondo chiuso**: se una proposizione non compare nella lista, si presume sia falsa. Ad esempio, la formula qui sopra dà per scontato che  $\neg K(PulitoSin)$  e  $\neg K(\neg PulitoSin)$ , catturando così il fatto che l'agente ignora il valore di verità di  $PulitoSin$ .

Le ultime due opzioni sono più o meno equivalenti, ma noi useremo la terza, ovvero le proposizioni di conoscenza, poiché rappresentano una descrizione più vivida delle percezioni. Inoltre, sappiamo già come scrivere descrizioni STRIPS che adottano l'**ipotesi del mondo chiuso**.

In entrambi i casi, ogni simbolo proposizionale può apparire in tre modi: positivo, negativo o sconosciuto. In questo modo si possono quindi descrivere esattamente 3<sup>n</sup> stati-credenza. Ora, l'insieme degli stati-credenza è l'insieme delle parti

---

<sup>11</sup> La rappresentazione canonica più diffusa per una formula proposizionale generica è il **diagramma binario di decisione** o BDD (Bryant, 1992).

<sup>12</sup> Questa è la stessa notazione usata per gli agenti basati su circuito nel Capitolo 7. Alcuni autori la usano per indicare “sa se  $P$  è vero (o no)”. Tradurre le due rappresentazioni l'una nell'altra è banale.

(quello che contiene tutti i possibili sottoinsiemi) dell'insieme degli stati fisici. Ci sono  $2^n$  stati fisici, per cui gli stati-credenza saranno  $2^{2^n}$ . Questo numero è molto superiore a  $3^n$ , ragion per cui le opzioni 2 e 3 saranno molto limitate nella rappresentazione degli stati-credenza. Si ritiene che ciò sia inevitabile, perché *qualsiasi schema capace di rappresentare ogni possibile stato-credenza nel caso pessimo richiederà  $O(\log_2(2^{2^n})) = O(2^n)$  bit per rappresentare ognuno di essi.* I nostri semplici schemi richiedono solo  $O(n)$  bit per rappresentare ogni stato-credenza, rinunciando all'espressività in favore della compattezza. In particolare, se si verifica un'azione di cui è sconosciuta una delle precondizioni, lo stato-credenza risultante non sarà rappresentabile in modo esatto e il risultato dell'azione diventerà sconosciuto.

Ora dobbiamo decidere come funzionerà la percezione. Per questo ci sono due scelte: la prima è la **percezione automatica**, che significa che a ogni passo temporale l'agente riceverà tutte le percezioni possibili. L'esempio nella Figura 12.12 presume che ci sia percezione automatica dello sporco nella posizione corrente. Alternativamente possiamo scegliere la **percezione attiva**, che significa che le percezioni saranno ottenute solo previa esecuzione di specifiche **azioni sensorie** come *ControllaSporco* o *ControllaPosizione*. Tratteremo entrambi i casi uno per volta.

Scriviamo una descrizione di azione usando le proposizioni di conoscenza. Supponiamo che un agente si muova a *Sinistra* nel mondo del “doppio Murphy alternativo” con percezione automatica dello sporco locale; secondo le regole del mondo, se il riquadro che sta lasciando è pulito l'agente potrà forse lasciare dello sporco dietro a sé. Come effetto *fisico*, questo sarebbe *disgiuntivo*; ma come effetto sulla *conoscenza*, il risultato è la semplice cancellazione della conoscenza dell'agente riguardo a *PulitoDx*. Grazie alla percezione dello sporco locale l'agente saprà anche se vale *PulitoSin*, in un senso o nell'altro, e saprà anche che si trova *ASin*:

$$\begin{aligned}
 &\text{Azione}(\text{Sinistra}, \text{PRECOND: } ADx, \\
 &\quad \text{EFFETTO: } K(ASin) \wedge \neg K(ADx) \wedge \text{when } PulitoDx: \neg K(PulitoDx) \wedge \\
 &\quad \quad \quad \text{when } PulitoSin: K(PulitoSin) \wedge \\
 &\quad \quad \quad \text{when } \neg PulitoSin: K(\neg PulitoSin)) . \tag{12.2}
 \end{aligned}$$

Notate che le precondizioni e le condizioni *when* sono proposizioni semplici, non di conoscenza. Questo è corretto, perché il risultato delle azioni dipende effettivamente dal mondo, ma come possiamo verificarne la verità se tutto quello che abbiamo è uno stato-credenza? In realtà se l'agente *conosce* una proposizione nello stato-credenza corrente, diciamo  $K(ADx)$ , allora tale proposizione dev'essere vera nello stato fisico corrente e l'azione effettivamente applicabile. Se l'agente non conosce una proposizione – ad esempio, la condizione *when PulitoSin* – allora lo stato-credenza deve includere mondi in cui il riquadro sinistro è pulito e mondi in cui non lo è. È proprio questo che dà origine a stati-credenza multipli come risultato di un'azione. Così, se lo stato iniziale è  $(K(ADx) \wedge K(PulitoDx))$ , dopo la mossa a *Sinistra* i due stati-credenza risultanti saranno  $(K(ASin) \wedge K(PulitoSin))$  e  $(K(ASin) \wedge K(\neg PulitoSin))$ . In entrambi i casi, il valore di verità di *PulitoSin* è noto, per cui il suo valore può essere usato come test all'interno del piano.



percezione automatica

percezione attiva  
azioni sensorie

Con la percezione attiva, e non più automatica, l'agente otterrà informazioni sensorie solo su richiesta. Così, dopo essersi mosso a *Sinistra*, l'agente non saprà se il riquadro è sporco, cosicché nella descrizione dell'azione dell'Equazione (12.2) non compariranno più i due effetti condizionali. Per appurare il grado di pulizia del riquadro l'agente può ricorrere all'azione *ControllaSporco*:

$$\begin{aligned}
 \text{Azione}(\text{ControllaSporco}, \text{EFFETTO}: & \text{when } ASin \wedge \text{PulitoSin}: K(\text{PulitoSin}) \wedge \\
 & \text{when } ASin \wedge \neg \text{PulitoSin}: K(\neg \text{PulitoSin}) \wedge \\
 & \text{when } ADx \wedge \text{PulitoDx}: K(\text{PulitoDx}) \wedge \\
 & \text{when } ADx \wedge \neg \text{PulitoDx}: K(\neg \text{PulitoDx})) . \quad (12.3)
 \end{aligned}$$

È facile dimostrare che *Sinistra* seguita da *ControllaSporco* nella situazione con percezione attiva ha come risultato gli stessi due stati-credenza di *Sinistra* da sola nella situazione con percezione automatica. Con la percezione attiva, le azioni fisiche fanno sempre corrispondere uno stato-credenza a un singolo stato-credenza successore. Gli stati-credenza multipli possono essere introdotti solo attraverso le azioni sensorie, che forniscono conoscenza specifica e quindi permettono di usare test condizionali nei piani.

Abbiamo descritto l'approccio generale alla pianificazione condizionale basata sulla ricerca AND-OR nello spazio degli stati. L'approccio si è dimostrato abbastanza efficace su problemi di prova, ma altri si sono rivelati intrattabili. Teoricamente, può essere dimostrato che la pianificazione condizionale appartiene a una classe di complessità più difficile della pianificazione classica. Ricordate che la definizione della classe NP è che una soluzione candidata può essere controllata in tempo polinomiale per verificare se lo è veramente. Questo è vero per i piani classici (almeno quelli di dimensione polinomiale), per cui il problema relativo è *NP*. Ma nella pianificazione condizionale una soluzione candidata dev'essere verificata in *tutti* i possibili stati per vedere se esiste *un qualche* cammino che attraversa il piano e soddisfa l'obiettivo. La verifica delle combinazioni "tutti/alcuni" non può essere svolta in tempo polinomiale, per cui la pianificazione condizionale dev'essere più difficile di *NP*. L'unica via d'uscita è ignorare alcune delle possibili contingenze durante la fase di pianificazione e gestirle quando si verificano effettivamente. Questo è l'approccio che prendiamo in considerazione nel prossimo paragrafo.

## 12.5 Monitoraggio dell'esecuzione e ripianificazione

Un agente con **monitoraggio dell'esecuzione** controlla le proprie percezioni per verificare se tutto sta andando secondo i piani. La legge di Murphy ci dice che anche i piani più accurati di topi, uomini o agenti di pianificazione condizionale sono destinati a fallire frequentemente. Il problema è l'indeterminatezza illimitata: potranno sorgere circostanze impreviste che rendono scorrette le descrizioni delle

azioni dell'agente. Ne consegue che, in ambienti realistici, il monitoraggio dell'esecuzione è una necessità. Ne considereremo due tipi: una forma semplice ma debole chiamata **monitoraggio delle azioni**, in cui un agente controlla l'ambiente per verificare che l'azione successiva possa funzionare, e una più complessa ma efficace denominata **monitoraggio dei piani**, in cui l'agente verifica l'intera parte rimanente del piano in esecuzione.

Un agente di ripianificazione sa cosa fare quando accade qualcosa di inaspettato: invoca un'altra volta il pianificatore per ottenere un nuovo piano con cui raggiungere l'obiettivo. Per evitare di perdere troppo tempo, normalmente si tenta di riparare il piano vecchio cercando un modo di tornare indietro dall'inaspettato stato corrente.

Come esempio, ritorniamo al mondo dell'aspirapolvere "doppio Murphy" della Figura 12.9. In questo mondo, spostarsi in un riquadro pulito talvolta deposita dello sporco in quella locazione; ma cosa succede se l'agente non lo sa o non se ne cura? In tal caso la soluzione sarà molto semplice: [*Sinistra*]. Se all'arrivo non c'è perdita di sporco da parte dell'agente, l'obiettivo sarà effettivamente soddisfatto. In caso contrario, dato che la precondizione *PulitoSin* del passo implicito *Fine* non è soddisfatta, l'agente genererà un nuovo piano: [*Aspira*]. L'esecuzione di quest'ultimo avrà sempre successo.

Insieme, il monitoraggio dell'esecuzione e la ripianificazione costituiscono una strategia generale applicabile agli ambienti sia completamente che parzialmente osservabili, che può sfruttare una varietà di tecniche tra cui la pianificazione nello spazio degli stati, quella con ordinamento parziale e i piani condizionali. Un semplice approccio alla pianificazione nello spazio degli stati è mostrato nella Figura 12.13. L'agente comincia con un obiettivo e crea un piano iniziale per raggiungerlo; dopodiché comincia a eseguire azioni una dopo l'altra. L'agente di ripianificazione, a differenza degli altri, tiene traccia sia del pezzo di piano che rimane ancora da eseguire (*piano*) che di quello originale completo (*piano-intero*) ed effettua il **monitoraggio delle azioni**: prima di eseguire l'azione successiva di *piano*, esamina le percezioni per controllare se per caso una delle precondizioni risulta insoddisfatta. Se è così, l'agente cerca di ritornare "in carreggiata" ripianificando una sequenza di azioni che lo riporti in qualche punto di *piano-intero*.

La Figura 12.14 fornisce una rappresentazione schematica di questo processo. Il ripianificatore nota che le precondizioni della prima azione di *piano* non sono soddisfatte nello stato corrente e chiama il pianificatore affinché costruisca un nuovo sottopiano chiamato *riparazione* che lo porterà dallo stato corrente a qualche stato *s* di *piano-intero*. In quest'esempio, lo stato *s* è un passo indietro rispetto al *piano* rimanente (questa è la ragione per cui teniamo in memoria tutto il piano, e non solo la parte che rimane ancora da eseguire). In generale, verrà scelto lo stato *s* più vicino possibile allo stato corrente. La concatenazione di *riparazione* e della porzione di *piano-intero* da *s* in avanti – che chiameremo *continuazione* – costituisce il nuovo *piano*, e l'agente è pronto a riprendere l'esecuzione.

monitoraggio delle azioni

---

```

function AGENTE-RIPIANIFICAZIONE(percezione) returns un'azione
 static: KB, una base di conoscenza (che include la descrizione delle azioni)
 piano, un piano, inizialmente[]
 piano-intero, un piano, inizialmente[]
 obiettivo, un obiettivo

 TELL(KB, COSTRUISCI-FORMULA-PERCEZIONE(percezione, t))
 stato_corrente ← DESCRIZIONE-STATO(KB, t)
 if piano = [] then
 piano-intero ← piano ← PIANIFICATORE(stato_corrente, obiettivo, KB)
 if PRECONDIZIONI(PRIMO(piano)) non sono attualmente vere in KB then
 candidati ← ORDINA(piano-intero, in base alla distanza da stato_corrente)
 trova lo stato s in candidati tale che
 fallimento ≠ riparazione ← PIANIFICATORE(stato_corrente, s, KB)
 continuazione ← la coda di piano-intero partendo da s
 piano-intero ← piano ← CONCATENA(riparazione, continuazione)
 return POP(piano)

```

---

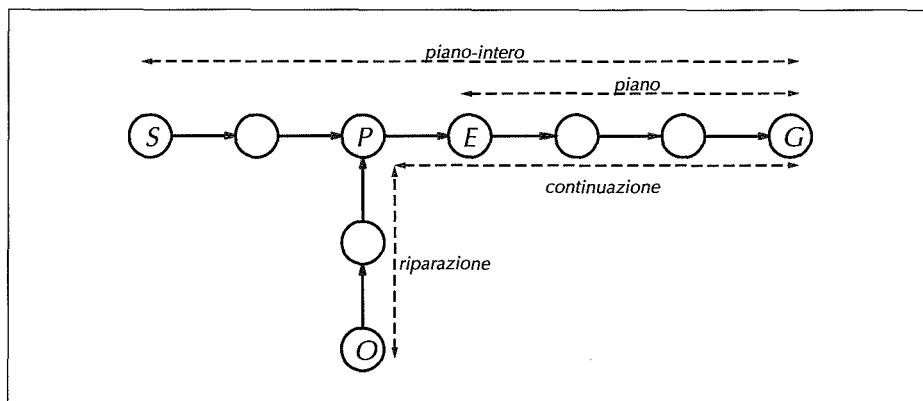
**Figura 12.13** Un agente che esegue il monitoraggio dell'esecuzione e la ripianificazione, utilizzando come sottoprocedura un algoritmo completo di pianificazione nello spazio degli stati chiamato PIANIFICATORE. Se le precondizioni dell'azione successiva non sono soddisfatte l'agente esegue un ciclo nei possibili punti *p* di piano-intero cercando di trovare uno stato raggiungibile da PIANIFICATORE. Questo cammino prende il nome di *riparazione*. Se PIANIFICATORE riesce a trovare una riparazione, l'agente concatena *riparazione* alla coda del piano dopo *p* per creare il nuovo piano e restituisce il suo primo passo.

Ora torniamo al problema esemplificativo di ottenere un tavolo e una sedia dello stesso colore, e questa volta cerchiamo di risolverlo attraverso la ripianificazione. Presumeremo che l'ambiente sia completamente osservabile. Nello stato iniziale la sedia è blu, il tavolo è verde e ci sono due latte di vernice, una blu e una rossa. Questo ci dà la seguente definizione del problema:

```

Init(Colore(Sedia, Blu) ∧ Colore(Tavolo, Verde)
 ∧ ContieneColore(LB, Blu) ∧ LattaDiVernice(LB)
 ∧ ContieneColore(LR, Rosso) ∧ LattaDiVernice(LR))
Obiettivo(Colore(Sedia, x) ∧ Colore(Tavolo, x))
Azione(Dipingi(oggetto, colore),
 PRECOND: DisponibilePittura(colore)
 EFFETTO: Colore(oggetto, colore))
Azione(Apri(latta),
 PRECOND: LattaDiVernice(latta) ∧ ContieneColore(latta, colore)
 EFFETTO: DisponibilePittura(colore))

```



**Figura 12.14** Prima dell'esecuzione il pianificatore costruisce un piano, qui chiamato *piano-intero*, per andare da *S* a *G*. L'agente esegue il piano fino al punto *E*. Prima di proseguire con il restante *piano*, controlla come di consueto le precondizioni e si accorge che in effetti si trova nello stato *O* anziché *E*. A questo punto invoca il suo algoritmo di pianificazione per costruire una *riparazione*, che è un piano che va da *O* a qualche punto *P* del *piano-intero* originale. Il nuovo *piano* ora è costituito dalla concatenazione di *riparazione* e *continuazione* (la parte finale del *piano-intero* originale, con inizio in *P*).

Il PIANIFICATORE dell'agente dovrebbe trovare il seguente piano:

[*Inizio*; *Apri(LB)*; *Dipingi(Tavolo, Blu)*; *Fine*]

Ora l'agente è pronto a eseguire il piano. Presumiamo che tutto vada per il meglio mentre l'agente apre la latta e applica la vernice blu al tavolo. Gli agenti visti sin qui a questo punto dichiarerebbero vittoria, avendo completato tutti i passi del piano. Ma l'agente con monitoraggio dell'esecuzione deve prima verificare la precondizione del passo *Fine*, che richiede che i due mobili siano dello stesso colore. Supponiamo che l'agente percepisca che non è proprio così, perché durante la pittura è rimasta scoperta un'area del tavolo di colore verde: a questo punto deve determinare quale punto di *piano-intero* scegliere come obiettivo parziale e trovare una sequenza di azioni di riparazione per raggiungerlo. L'agente nota che lo stato corrente è identico alla precondizione dell'azione *Dipingi*, così può scegliere una sequenza vuota come *riparazione* e dire che il suo *piano* è la stessa sequenza [*Dipingi*, *Fine*] appena tentata. A questo punto il monitoraggio dell'esecuzione riprende con la seconda esecuzione di *Dipingi*. Questo comportamento si ripeterà ciclicamente finché il tavolo non sarà percepito come completamente dipinto. Noteate che il ciclo è originato dal processo pianificazione-esecuzione-riplanificazione, e non per la presenza di cicli espliciti all'interno del piano.

Quello che guarda esclusivamente le azioni è un metodo molto semplice di monitoraggio dell'esecuzione, e talvolta può causare un comportamento non molto intelligente. Ad esempio, supponiamo che l'agente abbia costruito un piano per risolvere il problema dei mobili dipingendo di rosso sia la sedia che il tavolo. Una volta aperta la latta, però, si vede che c'è solo vernice sufficiente per la sedia. Il mo-

monitoraggio dei piani

nitoraggio delle azioni non rileverebbe un fallimento se non *dopo* che la sedia è stata dipinta, perché a quel punto *DisponibilePittura(Rosso)* diventerà falso. Quello che il piano dovrebbe veramente fare è rilevare il fallimento non appena lo stato diventa tale che il piano restante non può più funzionare. Il **monitoraggio dei piani** riesce a far ciò verificando le precondizioni di tutte le azioni rimanenti nel piano che non sono destinate a essere soddisfatte da qualche passo ancora da svolgere. Il monitoraggio dei piani interrompe l'esecuzione di un piano inattuabile il prima possibile anziché continuare l'esecuzione fino a raggiungere effettivamente il fallimento.<sup>13</sup> In alcuni casi questo può salvare l'agente dal disastro, nel caso in cui l'esecuzione del piano porti in un vicolo cieco dal quale l'obiettivo non sarebbe più raggiungibile.

Modificare un algoritmo di pianificazione in modo che annoti il piano in ogni punto con le precondizioni per il successo delle azioni restanti è abbastanza semplice. Se estendiamo il monitoraggio in modo che verifichi se lo stato corrente soddisfa le precondizioni del piano in un qualsiasi punto futuro, anziché solo quello in cui ci troviamo, l'agente potrà anche trarre vantaggio dalla serendipità, ovvero dal successo involontario. Se mentre l'agente sta dipingendo la sedia di rosso passa un amico e dipinge il tavolo dello stesso colore, allora le precondizioni del piano finale sono state soddisfatte (cioè, l'obiettivo è stato raggiunto), e l'agente può andarsene a casa in anticipo.

Fin qui abbiamo descritto il monitoraggio e la pianificazione in ambienti completamente osservabili. Le cose diventano molto più complicate quando l'ambiente è osservabile solo parzialmente. Prima di tutto, potrebbero andare storte cose che l'agente non è in grado di rilevare. In secondo luogo, la "verifica delle precondizioni" richiederebbe l'esecuzione di azioni sensorie che dovranno essere pianificate: questo potrà avvenire al momento della pianificazione (e ritorneremo nella situazione della pianificazione condizionale) oppure durante l'esecuzione. Nel caso peggiore, l'esecuzione di un'azione sensoria potrebbe esigere un piano complesso che necessiterebbe anch'esso di monitoraggio, ma questo richiederebbe ulteriori azioni sensorie, e così via. Se l'agente insiste nella verifica di ogni precondizione, potrebbe non arrivare mai al punto di *fare* effettivamente qualcosa. L'agente dovrebbe limitarsi a controllare le variabili che sono importanti, hanno una certa probabilità di cambiare senza preavviso e non sono troppo costose da percepire. Questo permette all'agente di rispondere in modo appropriato alle minacce gravi, senza perdere tutto il tempo controllando se per caso il cielo gli sta cadendo sulla testa.

Dopo aver descritto un metodo per il monitoraggio e la pianificazione, occorre chiedersi: "ma funziona?". Questa domanda è sorprendentemente subdola.

---

<sup>13</sup> Il monitoraggio dei piani rende il nostro agente più intelligente dello scarabeo stercorario (v. pag. 52). Il nostro agente noterebbe che la pallina di sterco non è più in suo possesso ed eseguirebbe una ripianificazione per procurarsene un'altra con cui tappare l'ingresso della tana.

Se intendiamo “possiamo garantire che l'agente raggiungerà sempre l'obiettivo, anche in presenza di indeterminatezza illimitata?” allora la risposta è no, perché potremmo sempre infilarci inavvertitamente in un vicolo cieco, come nella ricerca online nel Paragrafo 4.5. Ad esempio, l'agente aspirapolvere potrebbe non sapere che le sue batterie hanno un'autonomia limitata. Ma escludiamo pure i vicoli ciechi; diamo cioè per scontato che l'agente possa sempre costruire un piano per raggiungere l'obiettivo partendo da *qualsiasi* stato. Se presumiamo che l'ambiente sia veramente non deterministico, nel senso che un tentativo di eseguire un piano ha sempre *qualche* probabilità di successo, l'agente prima o poi raggiungerà l'obiettivo. L'agente di ripianificazione, quindi, ha una capacità analoga a quella dell'agente di pianificazione condizionale. In effetti, è possibile modificare un pianificatore condizionale in modo che costruisca soluzioni parziali che includono passi della forma “*if <test> then piano\_A else ripianifica*”. Con le ipotesi che abbiamo formulato, un piano simile può essere una soluzione corretta per il problema originale, e anche molto più economico da costruire di un piano condizionale completo.

I problemi sorgono quando i ripetuti tentativi dell'agente di raggiungere l'obiettivo sono del tutto futili, perché bloccati da qualche precondizione o effetto che l'agente non conosce. Se in albergo vi hanno dato la chiave della camera sbagliata, inserirla e rimuoverla più volte non servirà ad aprire la porta.<sup>14</sup> Una soluzione potrebbe essere scegliere casualmente in un insieme di possibili piani di riparazione, invece di tentare lo stesso ogni volta. In questo caso, il piano di riparazione di andare alla reception e chiedere un'altra chiave sarebbe un'alternativa particolarmente utile. Dato che l'agente potrebbe non essere in grado di distinguere il caso veramente non deterministico da quello futile, introdurre un certo grado di variazione nelle riparazioni è in generale una buona idea.

Un'altra soluzione al problema delle descrizioni di azioni inesatte è l'**apprendimento**. Dopo qualche tentativo, un agente capace di apprendere dovrebbe riuscire a modificare la descrizione che dice che la chiave apre la porta. A quel punto, il ripianificatore costruirà automaticamente un piano alternativo, come quello di chiedere un'altra chiave. Questo tipo di apprendimento sarà descritto nel Capitolo 21, nel 2° volume.

Anche con tutte queste potenziali migliorie, l'agente di ripianificazione ha ancora qualche limitazione. Non può operare in ambienti real-time, dato che non c'è limite alla quantità di tempo spesa nella ripianificazione e di conseguenza al tempo richiesto per decidere l'azione successiva. Inoltre, l'agente non può formulare da solo nuovi obiettivi o accettarne altri in aggiunta a quelli correnti; per questa ragione non potrà vivere a lungo in un ambiente complesso. Queste limitazioni sono affrontate nel prossimo paragrafo.

---

<sup>14</sup> La ripetizione futile della riparazione di un piano è precisamente il comportamento esibito dalla vespa sphex. (v. pag. 52.)

agente di  
pianificazione continua

## 12.6 Pianificazione continua

In questo paragrafo progetteremo un agente in grado di esistere in un ambiente per un tempo indefinito. Non si tratta quindi di un “risolutore di problemi” che riceve un singolo obiettivo e pianifica e agisce finché non l’ha raggiunto; piuttosto attraversa una serie di fasi sempre diverse di formulazione di obiettivi, pianificazione e azione. Invece di pensare che il pianificatore e il monitor dell’esecuzione siano processi separati, di cui uno passa i propri risultati all’altro, possiamo considerarli un unico processo in un agente di **pianificazione continua**.

L’agente è sempre *a metà dello svolgimento* di un piano: quello generale di vivere la sua vita. Le sue attività includono l’esecuzione dei passi pronti per essere attuati, il raffinamento del piano per soddisfare precondizioni aperte o risolvere conflitti e la sua modifica alla luce di nuova informazione ottenuta durante l’esecuzione. Naturalmente, quando formula un nuovo obiettivo non ci saranno azioni pronte, per cui l’agente potrà dedicare un po’ di tempo alla generazione di un piano parziale. È perfettamente possibile, comunque, che l’agente cominci l’esecuzione prima che il piano sia completo, specialmente quando deve raggiungere sotto-obiettivi indipendenti. L’agente di pianificazione continua verifica costantemente lo stato del mondo, aggiornando il suo modello in base alle nuove percezioni mentre sta ancora deliberando sul da farsi.

Per prima cosa esamineremo un esempio, quindi descriveremo un programma agente, che chiameremo AGENTE-POP-CONTINUO perché usa piani con ordinamento parziale per rappresentare le attività che intende svolgere. Per semplificare la presentazione presumeremo che l’ambiente sia completamente osservabile: le stesse tecniche possono essere estese al caso parzialmente osservabile.

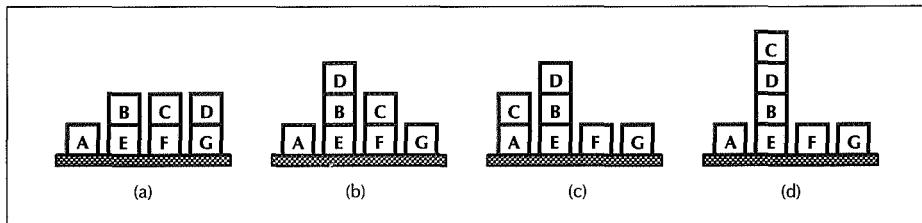
Il nostro esempio sarà un problema dal dominio del mondo dei blocchi (v. Paragrafo 11.1). Lo stato iniziale è mostrato nella Figura 12.15(a). Ci servirà l’azione *Move(x, y)*, che sposta il blocco *x* su quello *y*, a patto che entrambi siano liberi. Il suo schema è

*Azione*(*Move(x, y)*) ,

PRECOND: *Libero(x)  $\wedge$  Libero(y)  $\wedge$  On(x, z)* ,

EFFETTO: *On(x, y)  $\wedge$  Libero(z)  $\wedge$   $\neg$ On(x, z)  $\wedge$   $\neg$ Libero(y)* .

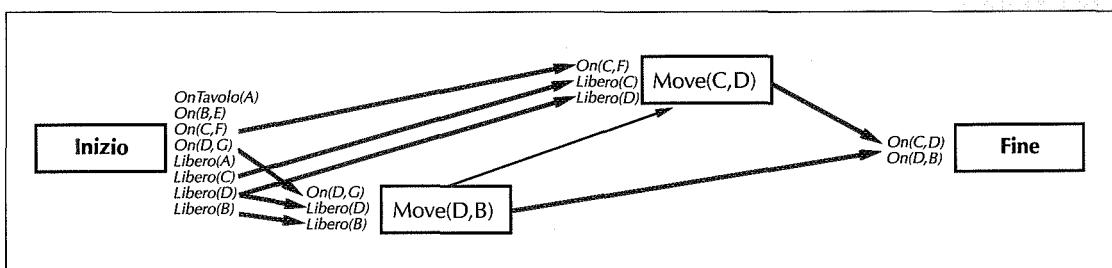
Per prima cosa l’agente deve formulare un obiettivo. Non discuteremo qui l’aspetto della formulazione; supponiamo che l’agente abbia ricevuto l’ordine (o abbia deciso da solo) di raggiungere l’obiettivo *On(C, D)  $\wedge$  On(D, B)*. L’agente comincia a pianificare: a differenza di tutti gli agenti considerati fin qui, che “chiudono la porta” alle percezioni finché non hanno costruito una soluzione completa del problema, l’agente di pianificazione continua costruisce il piano in modo incrementale, e ogni incremento richiede un tempo limitato. Dopo ogni incremento, l’agente restituisce *NoOp* (nessuna operazione) come azione prescelta e controlla ancora le percezioni.



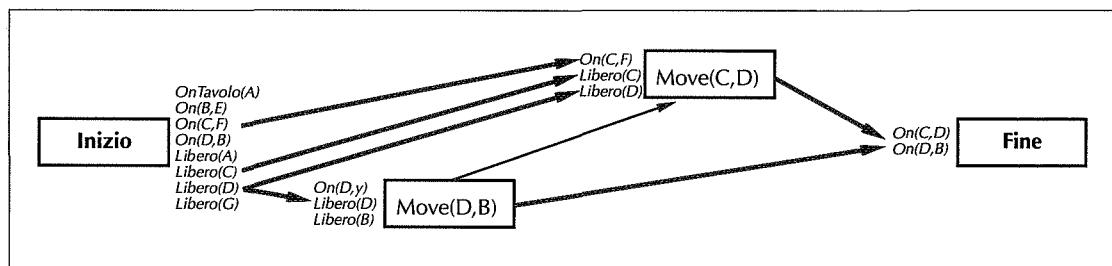
**Figura 12.15** La sequenza di stati attraversata dall'agente di pianificazione continua mentre cerca di raggiungere lo stato obiettivo  $On(C, D) \wedge On(D, B)$ , indicato con la (d). Lo stato iniziale è (a). In (b) c'è stata l'interferenza di un altro agente, che ha posto  $D$  su  $B$ . In (c) l'agente ha eseguito  $Move(C, D)$  ma ha fallito, lasciando cadere  $C$  su  $A$ . Il secondo tentativo di  $Move(C, D)$  raggiunge lo stato obiettivo (d).

Presumiamo che le percezioni non cambino e che l'agente arrivi velocemente al piano mostrato nella Figura 12.16. Notate che, sebbene le precondizioni di entrambe le azioni siano soddisfatte da *Inizio*, c'è un vincolo di ordinamento che pone  $Move(D, B)$  prima di  $Move(C, D)$ . Questo è necessario per assicurarsi che  $Libero(D)$  rimanga vero fino al completamento di  $Move(D, B)$ . Attraverso tutto il processo di pianificazione continua, *Inizio* viene usata come etichetta per lo stato corrente; l'agente aggiorna lo stato dopo ogni azione.

Ora il piano è pronto per essere eseguito, ma prima che l'agente possa agire la natura decide di intervenire. Un agente esterno (forse uno scienziato impaziente) sposta  $D$  su  $B$ , cosicché il mondo si trova ora nello stato mostrato nella Figura 12.15(b). L'agente percepisce questo fatto, riconosce che  $Libero(B)$  e  $On(D, G)$  non sono più vere nello stato corrente e aggiorna di conseguenza il suo modello. I collegamenti causali che stavano fornendo le precondizioni  $Libero(B)$  e  $On(D, G)$  per l'azione  $Move(D, B)$  non sono più validi e devono essere rimossi dal piano. Il nuovo piano è mostrato nella Figura 12.17. In tutti gli istanti, *Inizio* rappresenta lo stato corrente, per cui questo *Inizio* è diverso da quello della figura precedente.



**Figura 12.16** Il piano iniziale costruito dall'agente di pianificazione continua: a questo punto è ancora indistinguibile da quello prodotto da un normale pianificatore con ordinamento parziale.



**Figura 12.17** Dopo che qualcuno ha spostato D su B, i collegamenti non supportati che fornivano *Libero(B)* e *On(D, G)* sono eliminati, producendo questo piano.

estensione

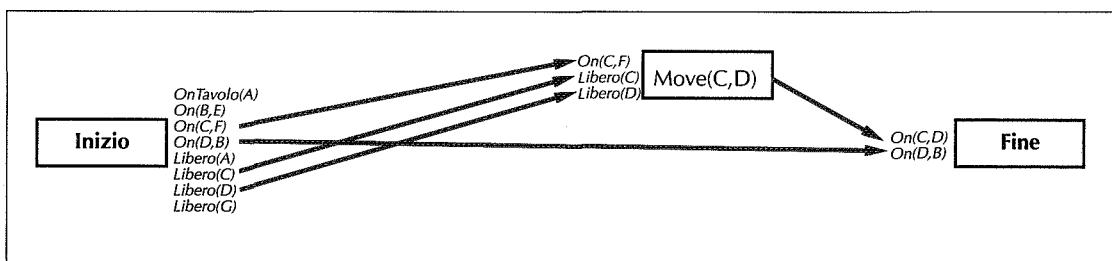
passo ridondante

Notate che ora il piano è incompleto: due precondizioni di *Move(D, B)* sono aperte e la sua precondizione *On(D, y)* non è più istanziata, perché non c'è più alcuna ragione di ritenere che il blocco sarà preso *G*.

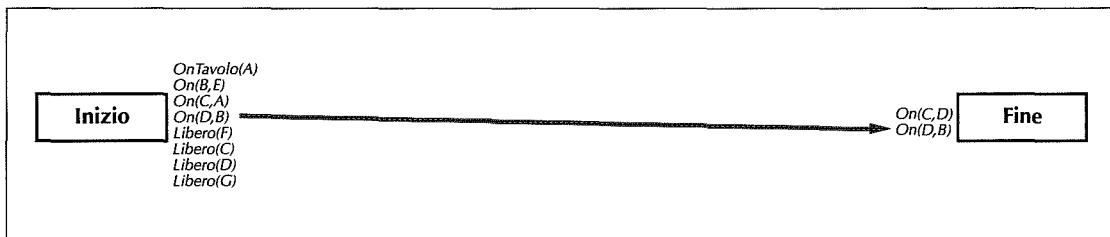
Ora l'agente può trarre vantaggio dall'interferenza “amichevole” notando che il collegamento causale *Move(D, B)*  $\xrightarrow{\text{On}(D,B)}$  *Fine* può essere rimpiazzato da un collegamento diretto da *Inizio* a *Fine*. Questa si chiama **estensione** di un collegamento causale, e viene effettuata ogni volta che una condizione può essere fornita da un passo precedente a quello previsto senza che ciò caisi un nuovo conflitto.

Una volta che il vecchio collegamento causale da *Move(D, B)* a *Fine* è rimosso, *Move(D, B)* non fornisce più alcun collegamento causale in assoluto: è diventato un **passo ridondante**. Tutti i passi ridondanti, e i collegamenti che li forniscono, possono essere eliminati dal piano. Questo ci dà la situazione nella Figura 12.18.

Ora il passo *Move(C, D)* è pronto all'esecuzione, dato che tutte le sue precondizioni sono soddisfatte dal passo *Inizio*, nessun altro passo lo precede e non confligge con alcun collegamento del piano. Il passo è quindi rimosso dal piano ed eseguito. Sfortunatamente, l'agente è piuttosto goffo e lascia cadere *C* sul blocco *A* anziché *D*, dando origine allo stato della Figura 12.15(c). Il nuovo piano è mostrato nella Figura 12.19. Notate che, sebbene non ci siano nuove azioni, c'è ancora una precondizione aperta per il passo *Fine*.



**Figura 12.18** Il collegamento fornito da *Move(D, B)* è stato rimpiazzato da uno che parte da *Inizio* e il passo *Move(D, B)*, ora ridondante, è stato eliminato.



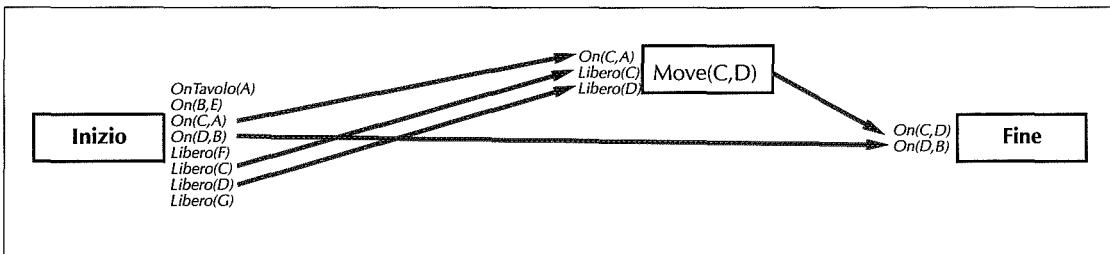
**Figura 12.19** Dopo l'esecuzione di  $Move(C, D)$  e la sua rimozione dal piano, gli effetti del passo *Inizio* riflettono il fatto che  $C$  è finito su  $A$  invece di essere posizionato, come si voleva, su  $D$ . La precondizione dell'obiettivo  $On(C, D)$  è ancora aperta.

L'agente decide di pianificare per soddisfare la condizione aperta. Ancora una volta,  $Move(C, D)$  ci permetterà di ottenere il risultato voluto: le sue precondizioni sono soddisfatte dai nuovi collegamenti causali che partono dal passo *Inizio*. Il nuovo piano è riportato nella Figura 12.20.

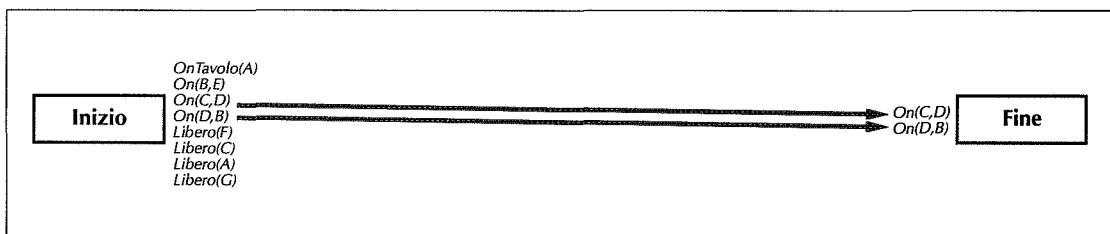
L'azione  $Move(C, D)$  è nuovamente pronta all'esecuzione: questa volta l'agente ce la fa, raggiungendo così lo stato obiettivo della Figura 12.15(d). Non appena il passo è cancellato dal piano, la condizione dell'obiettivo  $On(C, D)$  ridiventava aperta. Dato che il passo *Inizio* è aggiornato per riflettere il nuovo stato del mondo, comunque, la condizione può essere immediatamente soddisfatta da un collegamento proveniente dal passo *Inizio*. Questo è il normale corso degli eventi quando un'azione ha successo: il piano finale dello stato è mostrato nella Figura 12.21. Dato che tutte le condizioni dell'obiettivo sono soddisfatte dal passo *Inizio* e non rimangono azioni da eseguire, l'agente è ora libero di rimuovere l'obiettivo da *Fine* e formularne un altro.

Da quest'esempio possiamo vedere che la pianificazione continua è molto simile a quella con ordinamento parziale. A ogni iterazione l'algoritmo trova qualche aspetto del piano che ha bisogno di essere "aggiustato" – un cosiddetto **difetto del piano** – e lo aggiusta. POP può essere considerato un algoritmo di rimozione di difetti, dove questi ultimi possono essere precondizioni aperte o conflitti causali. L'agente di pianificazione continua, d'altra parte, ne risolve una varietà molto più ampia.

difetto del piano



**Figura 12.20** La precondizione aperta è risolta aggiungendo nuovamente al piano  $Move(C, D)$ . Notate i nuovi legami delle precondizioni.



**Figura 12.21** Dopo l'esecuzione di *Move (C, D)* e la sua cancellazione del piano, la restante condizione aperta *On(C, D)* è risolta aggiungendo un collegamento causale dal nuovo passo *Inizio*. Il piano è ora completo.

- ◆ *Obiettivo mancante*: l'agente può decidere di aggiungere uno o più nuovi obiettivi allo stato *Fine* (nell'ambito della pianificazione continua, sarebbe più sensato chiamare *Infinito* lo stato *Fine* e dire *StatoCorrente* anziché *Inizio*, ma abbiamo deciso di rispettare la tradizione).
- ◆ *Precondizione aperta*: si aggiunge un collegamento causale verso una precondizione aperta, scegliendo un'azione esistente o istanziandone una nuova (come in POP).
- ◆ *Conflitto causale*: dato un collegamento causale  $A \xrightarrow{p} B$  e un'azione  $C$  con effetto  $\neg p$ , si sceglie un vincolo di ordinamento o un vincolo sulle variabili per risolvere il conflitto (come in POP).
- ◆ *Collegamento non supportato*: se c'è un collegamento causale  $Inizio \xrightarrow{p} A$  e  $p$  non è più valido, si rimuove il collegamento (questo ci impedisce di eseguire un'azione con delle precondizioni false).
- ◆ *Azione ridondante*: se un'azione  $A$  non fornisce alcun collegamento causale, si rimuovono sia l'azione che i suoi collegamenti (questo ci permette di approfittare di circostanze impreviste ma favorevoli).
- ◆ *Azione non eseguita*: se un'azione  $A$  (diversa da *Fine*) ha le sue precondizioni soddisfatte in *Inizio*, non ha altre azioni (oltre a *Inizio*) prima di sé e non confligge con alcun collegamento causale, si rimuove  $A$  e i suoi collegamenti causali e la si restituisce come azione da eseguire.
- ◆ *Obiettivo storico non necessario*: Se nel piano non ci sono precondizioni aperte né azioni (in modo tale che tutti i collegamenti causali vadano direttamente da *Inizio* a *Fine*), allora abbiamo raggiunto l'insieme di obiettivi correnti, che possono essere rimossi con i loro collegamenti per far posto a nuovi obiettivi.

L'AGENTE-POP-CONTINUO è mostrato nella Figura 12.22. Il suo funzionamento è un ciclo di percezione, rimozione di difetti e azione. La base di conoscenza memorizza un piano persistente, da cui a ogni iterazione è rimosso un difetto. Fatto questo viene eseguita un'azione (anche se spesso sarà *NoOp*) e il ciclo si ripete.

---

```

function AGENTE-POP-CONTINUO(percezione) returns un'azione
 static: piano, un piano, inizialmente contenente solo Inizio e Fine

 azione \leftarrow NoOp (il default)
 EFFETTI[Inizio] = AGGIORNA(EFFETTI[Inizio], percezione)
 RIMUOVI-DIFETTO(piano) // possibilmente aggiornando azione
 return azione

```

---

**Figura 12.22** AGENTE-POP-CONTINUO è un agente di pianificazione continua con ordinamento parziale. Dopo aver ricevuto una percezione, l'agente rimuove un difetto dal piano in costante aggiornamento e restituisce un'azione. Spesso prima di essere pronto ad agire veramente dovrà effettuare più passi di pianificazione (e rimozione di difetti), durante i quali l'azione restituita sarà *NoOp*.

Quest'agente può gestire molti dei problemi che abbiamo citato nella discussione dell'agente di ripianificazione alla fine del Paragrafo 12.5. In particolare è capace di agire in tempo reale, di approfittare delle circostanze fortuite positive, di formulare i propri obiettivi e di gestire eventi inaspettati che influenzano i piani futuri.

## 12.7 Pianificazione multiagente

---

Fin qui ci siamo occupati di **ambienti ad agente singolo**, in cui il nostro agente operava da solo. Quando l'ambiente ne include altri, il nostro agente potrebbe semplicemente limitarsi a includerli nel suo modello dell'ambiente, senza modificare gli algoritmi base. In molti casi tuttavia questo porterebbe a prestazioni inadeguate, perché le interazioni con altri agenti sono ben diverse da quelle con la natura. In particolare, la natura è (si presume) indifferente alle intenzioni dell'agente,<sup>15</sup> cosa che non si può dire degli altri. Questo paragrafo introduce la pianificazione multiagente.

Nel Capitolo 2 abbiamo visto che gli ambienti multiagente possono essere **cooperativi** o **competitivi**. Cominceremo con un semplice esempio cooperativo: la pianificazione di squadra in un doppio di tennis. Si possono costruire piani che specificano le azioni di entrambi i giocatori della squadra; descriveremo tecniche per farlo in modo efficiente. La costruzione efficiente di piani è utile, ma non garantisce il successo: gli agenti devono accordarsi sull'uso dello stesso piano! Questo richiede un certo grado di **coordinamento**, che si può ottenere grazie alla **comunicazione**.

---

<sup>15</sup> I residenti del Regno Unito, dove il semplice fatto di pianificare un picnic garantisce la pioggia, saranno in disaccordo.

*Agenti(A, B)*

*Init(Posizione(A, [Sinistra, Fondo])  $\wedge$  Posizione(B, [Destra, Rete]))  $\wedge$*

*SiAvvicina(Pallina, [Destra, Fondo]))  $\wedge$  Partner(A, B)  $\wedge$  Partner(B, A)*

*Obiettivo(Ribattuta(Pallina)  $\wedge$  Posizione(agente, [x, Rete])))*

*Azione(Colpisci(agente, Pallina),*

*PRECOND: SiAvvicina(Pallina, [x, y])  $\wedge$  Posizione(agente, [x, y])  $\wedge$*

*Partner(agente, partner)  $\wedge$   $\neg$ Posizione(partner, [x, y])*

*EFFETTO: Ribattuta(Pallina))*

*Azione(Vai(agente, [x, y]),*

*PRECOND: Posizione(agente, [a, b]),*

*EFFETTO: Posizione(agente, [x, y])  $\wedge$   $\neg$ Posizione(agente, [a, b]))*

**Figura 12.23** Il problema del doppio tennistico. Due agenti stanno giocando insieme e possono trovarsi in una di quattro possibili posizioni [Sinistra, Fondo], [Destra, Fondo], [Sinistra, Rete] e [Destra, Rete]. La pallina può essere ribattuta se esattamente un giocatore si trova nel posto giusto.

## Cooperazione: obiettivi e piani congiunti

Due agenti che giocano a tennis in doppio hanno l'obiettivo congiunto di vincere il match, il che dà origine a vari sotto-obiettivi. Supponiamo che a un certo punto della partita l'obiettivo congiunto sia di respingere la pallina scagliata nella loro parte di campo e di assicurarsi che almeno uno di loro stia coprendo la rete. Possiamo rappresentare questa nozione come un problema di **pianificazione multiagente**, come si vede nella Figura 12.23.

Questa notazione introduce due caratteristiche nuove. Prima di tutto, *Agenti(A, B)* dichiara che ci sono due agenti, denominati *A* e *B*, che partecipano insieme al piano (in questo caso i giocatori avversari non saranno considerati agenti). In secondo luogo, ogni azione dovrà menzionare specificatamente un agente come parametro, perché occorre tener traccia di chi fa che cosa.

Una soluzione di un problema di pianificazione multiagente è un **piano congiunto** che contiene azioni per ogni agente. Un piano congiunto è una soluzione se l'obiettivo sarà raggiunto quando ogni agente esegue le azioni assegnategli. Il seguente piano è una soluzione al problema del tennis:

PIANO 1:

*A : [Vai(A, [Destra, Fondo]), Colpisci(A, Pallina)]*

*B : [NoOp(B), NoOp(B)] .*

Se entrambi gli agenti hanno la stessa base di conoscenza, e se questa è l'unica soluzione, allora tutto dovrebbe funzionare; ognuno dei due potrà determinare la soluzione ed eseguirla congiuntamente all'altro. Sfortunatamente per gli agenti (e presto vedremo dove sta la sfortuna), un altro piano soddisfa l'obiettivo altrettanto bene del primo:

PIANO 2:

$$\begin{aligned} A &: [Vai(A, [Sinistra, Rete]), NoOp(A)] \\ B &: [Vai(B, [Destra, Fondo]), Colpisci(B, Pallina)] . \end{aligned}$$

Se  $A$  sceglie il secondo piano e  $B$  il primo, nessuno ribatterà la pallina. Analogamente, se  $A$  sceglie il primo e  $B$  il secondo, probabilmente gli agenti si scontreranno; nessuno ribatterà la palla e la rete rimarrà scoperta. Di conseguenza, l'esistenza di piani congiunti corretti non significa che l'obiettivo sarà raggiunto. Gli agenti devono ricorrere a un meccanismo di **coordinamento** per seguire lo *stesso* piano congiunto; inoltre, dev'essere conoscenza comune (v. Capitolo 10) tra gli agenti che qualche particolare piano congiunto sarà eseguito.

coordinamento

## Pianificazione multibody

Questa sezione si concentra sulla costruzione di piano congiunti corretti, rimanendo per ora il problema del coordinamento. Questa viene chiamata **pianificazione multibody** (letteralmente “a più corpi”); essenzialmente è il problema che si trova ad affrontare un singolo agente centralizzato che può dettare le azioni da eseguire a diverse entità fisiche separate. Nel caso veramente multiagente, questa pianificazione permette a ogni agente di identificare i possibili piani congiunti che potrebbero funzionare se fossero eseguiti in modo coordinato.

pianificazione multibody

Il nostro approccio alla pianificazione multibody sarà basato su quella con ordinamento parziale, descritta nel Paragrafo 11.3: per semplicità supporremo la completa osservabilità dell'ambiente. C'è un altro problema che non si presenta nel caso ad agente singolo: l'ambiente non è più veramente **statico**, perché altri agenti potrebbero agire mentre uno sta decidendo cosa fare. Dovremo quindi preoccuparci anche della **sincronizzazione**. Per semplicità, presumeremo che ogni azione richieda la stessa quantità di tempo e che le azioni in ogni punto del piano congiunto siano simultanee.

sincronizzazione

In ogni istante di tempo, ogni agente sta eseguendo esattamente un'azione (che può anche essere *NoOp*). Questo insieme di azioni concorrenti prende il nome di **azione congiunta**. Ad esempio, un'azione congiunta nel dominio del tennis con due agenti  $A$  e  $B$  è  $\langle NoOp(A), Colpisci(B, Pallina) \rangle$ . Un piano congiunto consiste in un grafo parzialmente ordinato di azioni congiunte. Ad esempio, il piano numero 2 per il problema del tennis potrebbe essere rappresentato come sequenza di azioni congiunte:

$$\begin{aligned} &\langle Vai(A, [Sinistra, Rete]), Vai(B, [Destra, Fondo]) \rangle \\ &\langle NoOp(A), Colpisci(B, Pallina) \rangle . \end{aligned}$$

azione congiunta

Sarebbe possibile eseguire la pianificazione con l'algoritmo POP standard, applicandolo all'insieme di tutte le possibili azioni congiunte. L'unico problema è la dimensione di tale insieme: con 10 azioni e 5 agenti il numero di azioni congiunte sarà già  $10^5$ . Specificare correttamente le precondizioni e gli effetti di ogni azione sarebbe lungo e tedioso, e la pianificazione su un insieme così grande risulterebbe inefficiente.

Un'alternativa è definire le azioni congiunte in modo implicito, descrivendo come ogni singola azione interagisce con le altre. Questo sarà molto più semplice, perché la maggior parte delle azioni sarà indipendente da quasi tutte le altre; dovremo quindi indicare solo le poche azioni che presentano effettivamente interazioni. Per far questo possiamo arricchire le consuete descrizioni STRIPS o ADL con una nuova caratteristica: una **lista delle azioni concorrenti**. Il concetto è simile alle precondizioni di un'azione, tranne che in questo caso non vengono descritte variabili di stato ma azioni che devono o non devono essere eseguite in modo concorrente. Ad esempio, l'azione *Colpisci* potrebbe essere descritta come segue:

*Azione(Colpisci(A, Pallina),*  
**CONCORRENTI:**  $\neg \text{Colpisci}(B, \text{Pallina})$   
**PRECOND:** *SiAvvicina(Pallina, [x, y])*  $\wedge$  *Posizione(A, [x, y])*  
**EFFETTO:** *Ribattuta(Pallina)*) .

Qui abbiamo un vincolo di concorrenza proibita, che afferma che durante l'esecuzione dell'azione *Colpisci* non può esserci un'altra azione *Colpisci* da parte di un altro agente. Possiamo anche porre come *requisito* che ci siano azioni concorrenti, ad esempio quando servono due agenti per trasportare fino al campo da tennis un frigobar ricolmo di bevande. La descrizione per quest'azione dice che l'agente *A* non può eseguire un'azione *Trasporta* a meno che non ci sia un altro agente *B* che sta simultaneamente eseguendo *Trasporta* sullo stesso frigobar:

*Azione(Trasporta(A, frigobar, qua, là),*  
**CONCORRENTI:** *Trasporta(B, frigobar, qua, là)*  
**PRECOND:** *Posizione(A, qua)  $\wedge$  Posizione(frigobar, qua)  $\wedge$  Frigobar(frigobar)*  
**EFFETTO:** *Posizione(A, là)  $\wedge$  Posizione(frigobar, là)*  
 $\wedge$   $\neg \text{Posizione}(A, \text{qua}) \wedge \neg \text{Posizione}(\text{frigobar}, \text{qua})$ ) .

Con questa rappresentazione, è possibile creare un pianificatore molto vicino al POP con ordinamento parziale. Ci sono tre differenze:

- oltre alla relazione di ordinamento temporale  $A \prec B$  sono consentite anche  $A = B$  e  $A \preceq B$ , che significano rispettivamente “concorrente” e “prima di o concorrente”;
- quando una nuova azione richiede azioni concorrenti dobbiamo istanziare tali azioni, usando azioni nuove o preesistenti nel piano;
- le azioni di cui è vietata la concorrenza sono una nuova fonte di vincoli. Ogni vincolo dev'essere risolto spostando un'azione prima o dopo quella con cui confligge.

Questa rappresentazione ci dà l'equivalente di POP per i domini multibody. Potremmo estendere quest'approccio con i raffinamenti presentati negli ultimi due capitoli – HTN, osservabilità parziale, condizionali, monitoraggio dell'esecuzione e ripianificazione – ma questo andrebbe oltre gli scopi del libro.

## Meccanismi di coordinamento

Il metodo più semplice con cui un gruppo di agenti può assicurarsi di essere d'accordo sull'esecuzione di un piano congiunto è adottare una **convenzione** prima di cominciare. Una convenzione può essere espressa mediante vincoli di qualsiasi natura sulla selezione dei piani congiunti, al di là del requisito base che il piano deve funzionare se tutti gli agenti lo adottano. Ad esempio, la convenzione "stai dalla tua parte del campo" farà sì che i partner del doppio selezionino il piano 2, mentre la convenzione "un giocatore deve stare sempre a rete" li porterebbe a scegliere il piano 1. Alcune convenzioni, come quella di guidare dalla parte giusta della carreggiata, sono adottate in modo così diffuso che vengono considerate **leggi sociali**. Anche i linguaggi umani possono essere considerati convenzioni.

Le convenzioni del precedente paragrafo sono specifiche di un dominio e possono essere implementate vincolando le descrizioni delle azioni in modo che escludano le violazioni. Un approccio più generale è costituito dall'adozione di convenzioni indipendenti dal dominio. Ad esempio, se ogni agente esegue lo stesso algoritmo di pianificazione multibody con gli stessi input, potrebbe seguire la convenzione di eseguire il primo piano congiunto attuabile trovato, sicuro che gli altri agenti arriveranno alla stessa scelta. Una strategia più costosa, ma anche più robusta, sarebbe generare tutti i piani congiunti e poi scegliere quello, poniamo, la cui rappresentazione stampata precede alfabeticamente tutte le altre.

Le convenzioni possono anche scaturire da processi evolutivi: le colonie di insetti sociali eseguono piani congiunti molto elaborati, facilitati dal corredo genetico comune degli individui. La conformità può anche essere rinforzata dal fatto che la deviazione dalle convenzioni riduce l'adattamento evolutivo, cosicché ogni piano congiunto attuabile può diventare un equilibrio stabile. Considerazioni simili si applicano allo sviluppo del linguaggio umano, in cui la cosa importante non è la particolare lingua parlata da ogni individuo, ma il fatto che tutti parlino la stessa. Un esempio finale è dato dal comportamento dei branchi di uccelli: possiamo ottenere una buona simulazione se ogni agente uccello (talvolta indicato con il termine "**birdoid**" o **boid**) segue le tre regole qui sotto, combinandole in qualche modo.

1. Separazione: allontanati dai vicini quando si avvicinano troppo.
2. Coesione: muoviti verso la posizione media dei vicini.
3. Allineamento: orientati in base all'orientamento medio dei vicini.

Se tutti gli uccelli seguono lo stesso insieme di regole, il branco esibisce un **comportamento emergente** che consiste nel volare come un corpo pseudo-rigido di densità più o meno costante che non si disperde con il tempo. Come nel caso degli insetti, non c'è alcuna ragione che ogni agente possieda una copia del piano congiunto che modella le azioni degli altri agenti.

convenzione

leggi sociali

boid

comportamento emergente

Tipicamente, le convenzioni sono adottate per coprire un intero universo di problemi di pianificazione multiagente, e non vengono sviluppate da zero per ogni problema. Talvolta questo può portare a scarsa flessibilità ed errori, come si vede nei doppi di tennis quando la pallina è più o meno equidistante dai due partner. Senza una convenzione applicabile, gli agenti possono ricorrere alla **comunicazione** per raggiungere una conoscenza comune di un piano congiunto applicabile. Ad esempio, il tennista potrebbe gridare "mia!" o "tua!" per indicare il proprio piano congiunto preferito. Tratteremo i meccanismi della comunicazione più approfonditamente nel Capitolo 22 del 2º volume, in cui osserveremo che la comunicazione non richiede necessariamente uno scambio verbale. Ad esempio, un giocatore può comunicare all'altro il piano congiunto preferito semplicemente eseguendone la prima parte. Nel nostro problema tennistico, se l'agente *A* si avvicina alla rete, allora l'agente *B* è obbligato ad andare sul fondo per colpire la palla, dato che il piano 2 è l'unico che comincia con l'avvicinamento di *A* alla rete. Quest'approccio al coordinamento, chiamato talvolta **riconoscimento di piani**, funziona quando una singola azione (o una breve sequenza) è sufficiente a determinare in modo non ambiguo il piano congiunto prescelto.

La responsabilità di arrivare a un piano congiunto di successo può ricadere sulle spalle dei progettisti degli agenti o sugli agenti stessi. Nel primo caso, i progettisti dovrebbero dimostrare che le politiche e le strategie degli agenti avranno successo prima ancora che essi comincino a pianificare. Gli agenti stessi possono anche essere reattivi se questo funziona nell'ambiente in cui si trovano, e non è necessario che abbiano un modello esplicito degli altri agenti. Nel secondo caso gli agenti sono deliberativi; sono loro che devono dimostrare che il piano sarà efficace, prendendo in considerazione anche la presenza di altri agenti. Ad esempio, in un ambiente con due agenti logici *A* e *B*, entrambi potrebbero contenere la definizione

$$\forall p, s \ Realizzabile(p, s) \Leftrightarrow ConoscenzaComune(\{A, B\}, Raggiunge(p, s, Obiettivo))$$

che afferma che, in ogni situazione *s*, *p* è un piano congiunto realizzabile se tra gli agenti è conoscenza comune che *p* raggiungerà l'obiettivo. Sono necessari altri assiomi per stabilire la conoscenza comune dell'**intenzione congiunta** di eseguire un *particolare* piano congiunto; solo allora gli agenti potranno cominciare ad agire.

## Competizione

Non tutti gli ambienti multiagente prevedono la presenza di agenti cooperativi. Gli agenti con funzioni di utilità in conflitto sono in **competizione** tra loro. Un esempio sono i giochi a due giocatori a somma zero, come gli scacchi. Abbiamo visto nel Capitolo 6 che un agente giocatore di scacchi deve considerare le possibili mosse dell'avversario per molti passi nel futuro. Questo significa che un agente in un ambiente competitivo deve (a) riconoscere l'esistenza di altri agenti, (b) calcolare alcuni dei loro possibili piani, (c) calcolare le interazioni dei piani degli altri

riconoscimento di piani

intenzione congiunta

competizione

agenti con il proprio e (d) decidere qual è l'azione migliore alla luce di queste interazioni. La competizione quindi, come la cooperazione, richiede un modello dei piani degli altri agenti. D'altra parte, in un ambiente competitivo non c'è alcun bisogno di impegnarsi su piani congiunti.

Nel Paragrafo 12.4 abbiamo tracciato un'analogia tra i giochi e i problemi di pianificazione condizionale. L'algoritmo della Figura 12.10 costruisce piani che considerano sempre il caso pessimo per quanto riguarda l'ambiente, ragion per cui può essere applicato a situazioni competitive in cui l'agente è interessato solo al successo o al fallimento. Quando l'agente e i suoi avversari si preoccupano anche del *costo* di un piano, è appropriato usare minimax. Oggi come oggi non è stata svolta molta ricerca sull'integrazione di minimax con metodi, quali POP e HTN, che vanno oltre il modello di ricerca nello spazio degli stati presentato nel Capitolo 6. Torneremo sulla questione della competizione nel Paragrafo 17.6 (2° vol.), che tratta della teoria dei giochi.

## 12.8 Riepilogo

---

Questo capitolo ha affrontato alcune delle difficoltà della pianificazione e dell'azione nel mondo reale. I punti principali sono i seguenti.

- ◆ Molte azioni consumano **risorse**, come denaro, benzina o materie prime. È più comodo trattare queste risorse come misure fisiche collettive piuttosto che cercare di ragionare su, poniamo, ogni singola moneta o banconota nel mondo. Le azioni possono generare e consumare risorse, e di solito è economico ed efficace controllare i piani parziali per verificare il soddisfacimento dei vincoli sulle risorse prima di tentare raffinamenti ulteriori.
- ◆ Il tempo è una delle risorse più importanti. Per gestirlo si possono usare algoritmi specializzati di scheduling, oppure si può integrare lo stesso scheduling nella pianificazione.
- ◆ La pianificazione basata su **reti gerarchiche** (HTN) permette all'agente di accettare i consigli del progettista del dominio, espressi sotto forma di regole di scomposizione. Questo rende possibile la creazione dei piani molto grandi richiesti dalle applicazioni reali.
- ◆ Gli algoritmi di pianificazione standard presuppongono che l'informazione disponibile sia corretta e completa e che gli ambienti siano deterministici e completamente osservabili. Molti domini non rispettano queste ipotesi.
- ◆ L'incompletezza dell'informazione può essere gestita pianificando l'uso di azioni sensorie per ottenere l'informazione necessaria. I **piani condizionali** permettono all'agente di percepire il mondo durante l'esecuzione e decidere

quale ramo del piano seguire. In alcuni casi, la **pianificazione senza sensori o conformante** può consentire di costruire un piano che funziona senza necessità di ricevere percezioni durante l'esecuzione. Sia i piani senza sensori che quelli condizionali possono essere costruiti con una ricerca nello spazio degli stati-credenza.

- ◆ In caso di informazione inesatta si potrebbe avere come conseguenza la presenza di precondizioni insoddisfatte per azioni e piani. Il **monitoraggio dell'esecuzione** rileva le violazioni delle precondizioni e permette di completare il piano con successo.
- ◆ Un **agente di ripianificazione** sfrutta il monitoraggio delle esecuzioni e introduce riparazioni nel piano quando se ne manifesta la necessità.
- ◆ Un **agente di pianificazione continua** crea nuovi obiettivi durante l'esecuzione e reagisce in tempo reale.
- ◆ La **pianificazione multiagente** è necessaria quando nell'ambiente ci sono altri agenti con cui cooperare, competere o coordinarsi. La **pianificazione multibody** costruisce piani congiunti usando una scomposizione efficiente di azioni congiunte, ma dev'essere arricchita con qualche forma di coordinamento per assicurarsi che due agenti cooperativi siano d'accordo sul piano congiunto da eseguire.

## Note storiche e bibliografiche

La pianificazione a tempo continuo fu affrontata per la prima volta da DEVISER (Vere, 1983). Il problema della rappresentazione sistematica del tempo nei piani fu trattato da Dean et al. (1990) nel sistema FORBIN. NONLIN+ (Tate e Whiter, 1984) e SIPE (Wilkins, 1988, 1990) potevano ragionare sull'allocazione di risorse limitate ai vari passi di un piano. O-PLAN (Bell e Tate, 1985), un pianificatore HTN, aveva una rappresentazione generale e uniforme dei vincoli sul tempo e sulle risorse. Oltre all'applicazione della Hitachi menzionata nel testo, O-PLAN è stato applicato all'acquisizione software presso la Price Waterhouse e alla pianificazione di assemblaggio alla Jaguar. Diversi sistemi ibridi di pianificazione e scheduling sono stati usati nell'industria: ISIS (Fox et al., 1982; Fox, 1990) è stato applicato al job shop scheduling alla Westinghouse, GARI (Descotte e Latombe, 1985) ha pianificato la costruzione di parti meccaniche, FORBIN è stato usato per il controllo della produzione e NONLIN+ per la logistica navale.

Dopo un picco iniziale di lavori teorici alla fine degli anni '80 la pianificazione temporale è ritornata recentemente in auge, dopo che nuovi algoritmi e una maggiore potenza di calcolo hanno reso possibile affrontare applicazioni pratiche. I due pianificatori SAPA (Do e Kambhampati, 2001) e T4 (Haslum e Geffner, 2001) usavano entrambi la ricerca in avanti nello spazio degli stati, con euristiche

sofisticate per gestire la durata delle azioni e le risorse. Un'alternativa è usare linguaggi di descrizione delle azioni molto espressivi e guidarli con euristiche specifiche del dominio scritte da esperti umani, come fanno ASPEN (Fukunaga et al., 1997), HSTS (Jonsson et al., 2000) e IxTeT (Ghallab e Laruelle, 1994).

In ambito aerospaziale c'è una lunga tradizione di scheduling. T-SCHED (Drabble, 1990) è stato usato per lo scheduling delle sequenze di comando missione per il satellite UOSAT-II. OPTIMUM-AIV (Aarup et al., 1994) e PLAN-ERS1 (Fuchs et al., 1990), entrambi basati su O-PLAN, furono usati presso la European Space Agency rispettivamente per l'assemblaggio di veicoli spaziali e la pianificazione delle osservazioni. SPIKE (Johnston e Adorf, 1992) è stato usato alla NASA per la pianificazione delle osservazioni del telescopio spaziale Hubble, mentre lo Space Shuttle Ground Processing Scheduling System (Deale et al., 1994) svolge il job-shop scheduling dei turni di lavoro di 16.000 persone. Remote Agent (Muscettola et al., 1998) fu il primo pianificatore-scheduler autonomo a controllare un veicolo spaziale durante il volo a bordo del Deep Space One nel 1999. Vaessens et al. (1996) offre una panoramica della letteratura sul job-shop scheduling nella ricerca operativa; risultati teorici sono presentati da Martin e Shmoys (1996).

L'aggiunta ai programmi STRIPS della capacità di apprendere **macrops** – “macro-operatori” che consistono in una sequenza di passi primitivi – può essere considerato il primo meccanismo di pianificazione gerarchica (Fikes et al., 1972). Anche il sistema LAWALY (Siklossy e Dreussi, 1973) usava una gerarchia. Il sistema ABSTRIPS (Sacerdoti, 1974) introdusse l'idea di una **gerarchia di astrazioni**, in cui la pianificazione ai livelli superiori aveva il permesso di ignorare le precondizioni delle azioni di livello più basso per derivare la struttura generale di un piano funzionante. La tesi di dottorato di Austin Tate (1975b) e il lavoro di Earl Sacerdoti (1977) svilupparono le idee fondamentali della pianificazione HTN nella sua forma moderna. Molti pianificatori usati nella pratica, inclusi O-PLAN e SIPPE, sono HTN. Yang (1990) discute le caratteristiche delle azioni che rendono efficiente la pianificazione HTN. Erol, Hendler, e Nau (1994, 1996) presentano un pianificatore basato su una scomposizione gerarchica completa oltre a una serie di studi di complessità sui pianificatori HTN puri. Altri autori (Ambros-Ingerson e Steel, 1988; Young et al., 1994; Barrett e Weld, 1994; Kambhampati et al., 1998) hanno proposto l'approccio ibrido che abbiamo presentato in questo capitolo, in cui le scomposizioni sono semplicemente un'altra forma di raffinamento usata nella pianificazione con ordinamento parziale.

A partire dall'uso dei macro-operatori in STRIPS, uno degli obiettivi della pianificazione gerarchica è stato il riuso delle esperienze precedenti sotto forma di piani generalizzati. La tecnica dell'**apprendimento basato sulle spiegazioni**, che descriveremo approfonditamente nel Capitolo 19 (nel 2° vol.), è stata applicata per generalizzare piani precedentemente calcolati in diversi sistemi tra cui SOAR (Laird et al., 1986) e PRODIGY (Carbonell et al., 1989). Un approccio alternativo è memorizzare i piani precedentemente costruiti nella loro forma originale e riusarli per

macrops

gerarchia di astrazioni

risolvere per analogia problemi simili. Questo è l'approccio adottato dalla branca nota come **pianificazione basata sui casi** (Carbonell, 1983; Alterman, 1988; Hammond, 1989). Kambhampati (1994) sostiene che la pianificazione basata sui casi dovrebbe essere analizzata come una forma di pianificazione per raffinamento e presenta una base formale della pianificazione basata sui casi con ordinamento parziale.

L'impredicibilità e la parziale osservabilità degli ambienti reali fu riconosciuta molto presto nei progetti di robotica che usavano tecniche di pianificazione, tra cui Shakey (Fikes et al., 1972) e FREDDY (Michie, 1974). Il problema ricevette un'attenzione maggiore dopo la pubblicazione dell'importante articolo di McDermott (1978a) *Planning and Acting*.

I primi pianificatori, che non avevano condizioni e cicli, non riconoscevano esplicitamente il concetto di pianificazione condizionale; nonostante questo potevano ricorrere talvolta a uno stile coercitivo per rispondere all'incertezza dell'ambiente. Il sistema NOAH di Sacerdoti usava la coercizione nella sua soluzione al problema delle "chiavi e scatole", un problema-sfida in cui il pianificatore conosce molto poco dello stato iniziale. Mason (1993) sostenne che spesso le percezioni possono e devono essere evitate nella pianificazione robotica, e descrisse un piano privo di sensori capace di spostare uno strumento in una posizione specifica del tavolo per mezzo di una serie di azioni, *indipendentemente* dalla sua posizione iniziale. Descriveremo quest'idea nel contesto della robotica (v. Figura 25.17, nel 2° volume).

Goldman e Boddy (1996) introdussero il termine **pianificazione conformante** per i pianificatori privi di sensori che gestiscono l'incertezza forzando il mondo in stati conosciuti, e notando che i piani privi di percezioni sono spesso efficaci anche se l'agente possiede dei sensori. Il primo pianificatore conformante moderatamente efficiente fu Conformant Graphplan, o CGP, di Smith e Weld (1998). Ferraris e Giunchiglia (2000) e Rintanen (1999) hanno sviluppato indipendentemente pianificatori conformanti basati su SATPLAN. Bonet e Geffner (2000) descrivono un pianificatore conformante basato sulla ricerca euristica nello spazio degli stati-credenza, ispirandosi a idee sviluppate inizialmente negli anni '60 per i processi decisionali di Markov parzialmente osservabili, o POMDP (v. Capitolo 17, nel 2° volume). Oggi come oggi i pianificatori conformanti più veloci, come HSCP (Bertoli et al., 2001a), utilizzano diagrammi binari di decisione (BDD) (Bryant, 1992) per rappresentare gli stati-credenza e sono fino a cinque ordini di grandezza più veloci di CGP.

WARPLAN-C (Warren, 1976), una variante di WARPLAN, è stato uno dei primi pianificatori con azioni condizionali. Olawski e Gini (1990) illustrano i problemi più importanti della pianificazione condizionale.

L'approccio alla pianificazione condizionale descritto in questo capitolo è basato su algoritmi efficienti di ricerca su grafi AND-OR ciclici, sviluppati da Jimenez e Torras (2000) e Hansen e Zilberstein (2001). Bertoli et al. (2001b) descrive un approccio basato su BDD che costruisce piani condizionali con cicli.

C-BURIDAN (Draper et al., 1994) gestisce la pianificazione condizionale per azioni che hanno esiti probabilistici, un problema affrontato anche nel campo dei POMDP (Capitolo 17).

C'è una stretta relazione tra la pianificazione condizionale e la sintesi automatica di programmi; molti riferimenti sono stati citati nel Capitolo 9. I due campi sono stati approfonditi separatamente, in virtù dell'enorme differenza di costo tra l'esecuzione di istruzioni macchina e quella delle azioni tipiche di veicoli robotizzati o manipolatori. Linden (1991) tenta un esplicito interscambio di idee tra i due campi.

In retrospettiva, è possibile vedere come i più importanti algoritmi di pianificazione classica hanno stimolato lo sviluppo di versioni estese per i domini che presentano incertezza. Le tecniche basate su ricerca hanno portato alla ricerca nello spazio degli stati-credenza (Bonet e Geffner, 2000); SATPLAN ha dato origine a SATPLAN stocastico (Majercik e Littman, 1999) e alla pianificazione che utilizza la logica di Boole quantificata (Rintanen, 1999); la pianificazione con ordinamento parziale ha portato a UWL (Etzioni et al., 1992), CNLP (Peot e Smith, 1992), e CASSANDRA (Pryor e Collins, 1996). Sulla base di GRAPHPLAN è stato sviluppato Sensory Graphplan o SGP (Weld et al., 1998), anche se una versione pienamente probabilistica di GRAPHPLAN dev'essere ancora sviluppata.

Il primo trattamento importante del monitoraggio dell'esecuzione risale a PLANEX (Fikes et al., 1972), che operava insieme al pianificatore STRIPS per controllare il robot Shakey. PLANEX usava tabelle triangolari – sostanzialmente un meccanismo efficiente di memorizzazione delle precondizioni in ogni punto del piano – per permettere il recupero di fallimenti parziali senza una ripianificazione completa. Il modello di esecuzione di Shakey è discusso ulteriormente nel Capitolo 25. Il pianificatore NASL (McDermott, 1978a) trattava un problema di pianificazione semplicemente come una specifica per l'esecuzione di un'azione complessa, dimodoché esecuzione e pianificazione erano completamente unificate. Per ragionare su tali azioni complesse utilizzava tecniche di dimostrazione di teoremi.

SIPE (System for Interactive Planning e Execution monitoring) (Wilkins, 1988, 1990) è stato il primo pianificatore a occuparsi sistematicamente del problema della ripianificazione. È stato usato in progetti dimostrativi in vari domini, tra cui le operazioni di pianificazione sul ponte di volo di una portaerei e il job-shop scheduling per una fabbrica di birra australiana. Un altro studio ha utilizzato SIPE per pianificare la costruzione di edifici a più piani, uno dei domini più complessi mai affrontati da un pianificatore.

IPEM (Integrated Planning, Execution, e Monitoring) (Ambros-Ingerson e Steel, 1988) è stato il primo sistema a integrare pianificazione con ordinamento parziale ed esecuzione, dando così origine a un agente di pianificazione continua. Il nostro AGENTE-POP-CONTINUO unisce idee prese da IPEM, dal pianificatore PUCCINI (Golden, 1998) e dal sistema CYPRESS (Wilkins et al., 1995).

pianificazione reattiva

politiche

A metà degli anni '80 alcuni ritenevano che la pianificazione con ordinamento parziale e le tecniche correlate non sarebbero mai state abbastanza veloci da generare comportamenti efficaci per agenti immersi nel mondo reale (Agre e Chapman, 1987). Al loro posto furono proposti sistemi di **pianificazione reattiva**; nella loro forma più semplice si tratta di agenti reattivi, possibilmente dotati di uno stato interno, che possono essere implementati con regole condizione-azione di cui esistono molte varianti. L'architettura di sussunzione di Brooks (1986) (v. Capitoli 7 e 25) utilizzava automi a stati finiti a più livelli all'interno di robot dotati di zampe e ruote per controllare il loro movimento ed evitare ostacoli. Pengi (Agre e Chapman, 1987) è stato capace di giocare a un videogioco (completamente osservabile) usando circuiti booleani combinati con una rappresentazione "visuale" degli obiettivi correnti e con lo stato interno dell'agente.

I "piani universali" (Schoppers, 1987) furono sviluppati come un metodo tabellare per la pianificazione reattiva, ma risultarono essere una riscoperta dell'idea delle **politiche** usata da molto tempo nei processi decisionali di Markov. Un piano universale (o politica) contiene una corrispondenza da ogni stato all'azione che dovrebbe essere effettuata in tale stato. Ginsberg (1989) attaccò in modo veemente i piani universali, includendo risultati di intrattabilità per alcune formulazioni del problema di pianificazione reattiva. Schoppers (1989) rispose in modo altrettanto veemente.

Come spesso accade, la controversia può essere risolta con un approccio ibrido. Sfruttando gerarchie ben progettate, pianificatori HTN come PRS (Georgeff e Lansky, 1987) e RAP (Firby, 1996), così come gli agenti di pianificazione continua, possono arrivare in molti domini a tempi di risposta paragonabili a quelli reattivi ed esibire un comportamento complesso di pianificazione a lungo termine.

La pianificazione multiagente ha goduto in anni recenti di un salto di popolarità, benché la sua storia non sia breve. Konolige (1982) scrisse una formalizzazione della pianificazione multiagente in logica del primo ordine, mentre Pednault (1986) ne fornì una descrizione nello stile di STRIPS. Il concetto di intenzione congiunta, fondamentale se gli agenti devono eseguire un piano congiunto, scaturisce dalla ricerca sugli atti comunicativi (Cohen e Levesque, 1990; Cohen et al., 1990). La nostra presentazione della pianificazione con ordinamento parziale multibody è basata sul lavoro di Boutilier e Brafman (2001).

Abbiamo appena scalfito la superficie della ricerca sulla negoziazione nella pianificazione multiagente. Durfee e Lesser (1989) discutono come suddividere le attività tra gli agenti attraverso la negoziazione, Kraus et al. (1991) descrive un sistema per giocare a Diplomacy, un gioco da tavolo che richiede negoziazione, formazione e tradimento di alleanze, e disonestà. Stone (2000) mostra come gli agenti possono cooperare come compagni di squadra nell'ambiente competitivo, dinamico e parzialmente osservabile del calcio per robot. (Weiss, 1999) è una lunga panoramica dei sistemi multiagente.

Il modello dei boid a pag. 573 è dovuto a Reynolds (1987), che ha vinto un Premio Oscar per la sua applicazione agli stormi di pipistrelli e ai branchi di pinguini nel film *Batman Returns*.

## Esercizi

- 12.1 Esamineate attentamente la rappresentazione del tempo e delle risorse del Paragrafo 12.1.
- Perché è una buona idea far sì che *Durata(d)* sia un effetto di un'azione, anziché definire un campo separato dell'azione di forma DURATA: *d*? (*suggerimento*: considerate gli effetti condizionali e quelli disgiuntivi).
  - Perché RISORSE: *m* è un campo separato dell'azione invece di essere un effetto?
- 12.2 Una **risorsa consumabile** è una risorsa che viene (parzialmente) consumata da un'azione: ad esempio, attaccare motori alle macchine richiede viti che, una volta utilizzate, non sono più disponibili.
- Spiegate come modificare la rappresentazione della Figura 12.3 in modo che inizialmente ci siano 100 viti, il motore  $E_1$  richieda 40 viti e il motore  $E_2$  ne richieda 50. Nei letterali di effetto che riguardano le risorse potete usare i simboli di funzione + e -.
  - Spiegate come modificare la definizione di **conflitto** tra collegamenti causali e azioni nella pianificazione con ordinamento parziale per gestire la presenza di risorse consumabili.
  - Alcune azioni – come consegnare un carico di viti a una fabbrica o riempire un serbatoio – possono *aumentare* la disponibilità di risorse. Una risorsa è monotonicamente non-crescente se nessuna azione ne aumenta la disponibilità. Spiegate come si può sfruttare questa proprietà per potare lo spazio di ricerca.
- 12.3 Scrivete le scomposizioni dei passi *IngaggiaCostruttore* e *OttieniPermesso* della Figura 12.7, e mostrate come i sottopiani scomposti si collegano al piano generale.
- 12.4 Fornite un esempio, nel dominio della costruzione di case, di due sottopiani astratti che non possono essere fusi in un unico piano consistente senza condividere qualche passo (*suggerimento*: i punti in cui due parti fisiche della casa vengono in contatto sono anche punti in cui due sottopiani tendono a interagire).
- 12.5 Alcuni sostengono che un vantaggio della pianificazione HTN sta nel fatto che può risolvere problemi come “fai un viaggio da Los Angeles a New York e ritorno”, che sono difficili da esprimere in notazioni diverse da HTN perché lo stato di partenza e quello obiettivo hanno la stessa rappresentazione: *Posizione(LA)*. Potete pensare a un modo di rappresentare e risolvere questo problema senza HTN?

risorsa consumabile

- 12.6 Mostrate come una descrizione di azione standard di STRIPS può essere riscritta sotto forma di scomposizione HTN, usando la notazione *Raggiunge(p)* per denotare l'attività di raggiungere la condizione *p*.
- 12.7 Alcune operazioni dei linguaggi di programmazione standard possono essere modellate come azioni che modificano lo stato del mondo. Ad esempio, l'operazione di assegnamento copia il contenuto di una cella di memoria, mentre quella di stampa cambia lo stato dello stream di output. Un programma che consiste di operazioni simili può essere considerato un piano, il cui obiettivo è fornito nella specifica del programma. Di conseguenza è possibile usare algoritmi di pianificazione per costruire programmi che soddisfano una specifica data.
- Scrivete uno schema di operatore per l'assegnamento (del valore di una variabile a un'altra). Ricordate che il valore originale sarà sovrascritto!
  - Mostrate come un pianificatore può sfruttare la creazione di oggetti per produrre un piano che scambia i valori di due variabili usando una variabile temporanea.
- 12.8 Considerate la seguente argomentazione: in un'infrastruttura che permette l'esistenza di stati iniziali incerti, gli effetti disgiuntivi sono solo una comodità di notazione e non arricchiscono la capacità espressiva della rappresentazione. Ogni schema di azione *a* con effetto disgiuntivo  $P \vee Q$  può sempre essere sostituito dall'effetto condizionale *when R: P  $\wedge$  when  $\neg R: Q$* , che a loro volta può essere ridotto a due azioni regolari. *R* indica una proposizione casuale sconosciuta nello stato iniziale e per cui non esistono azioni sensorie. Trovate corretta quest'argomentazione? Considerate separatamente due casi, uno in cui il piano contiene solo un'istanza dello schema *a*, l'altro in cui ne contiene più d'una.
- 12.9 Perché la pianificazione condizionale non può gestire l'indeterminatezza illimitata?
- 12.10 Nel mondo dei blocchi siamo stati costretti a introdurre due azioni STRIPS, *Move* e *MoveSulTavolo*, per gestire correttamente il predicato *Libero*. Mostrate come si possono rappresentare entrambi i casi con una singola azione mediante gli effetti condizionali.
- 12.11 Nel mondo dell'aspirapolvere abbiamo illustrato degli effetti condizionali per l'azione *Aspira*: il riquadro che diventa pulito dipende dalla posizione corrente del robot. Potete pensare a un nuovo insieme di variabili proposizionali per definire gli stati del mondo, tali che *Aspira* abbia una descrizione *non condizionale*? Scrivete le descrizioni di *Aspira*, *Sinistra* e *Destra* usando le vostre proposizioni, e dimostrate che sono sufficienti a descrivere tutti gli stati del mondo.

- 12.12 Scrivete la descrizione completa di *Aspira* per l'aspirapolvere "doppio Murphy" che talvolta deposita sporco nel riquadro pulito in cui si è appena mosso e talvolta deposita sporco se viene effettuata l'azione *Aspira* in un riquadro pulito.
- 12.13 Trovate un tappeto adeguatamente sporco, libero da ostacoli, e passateci l'aspirapolvere. Disegnate il cammino seguito dall'aspirapolvere il più accuratamente possibile. Commentatelo facendo riferimento alle forme di pianificazione discusse in questo capitolo.
- 12.14 Quelle che seguono sono citazioni prese da bottiglie di shampoo. Identificate ognuna di esse come un piano non condizionale, condizionale o con monitoraggio dell'esecuzione. (a) "Strofinate. Risciacquate. Ripetete". (b) "Applicate lo shampoo allo scalpo e lasciatelo agire per qualche minuto. Sciacquate e ripetete se necessario". (c) "Consultate un medico se i problemi persistono".
- 12.15 L'algoritmo RICERCA-GRAFO-AND-OR della Figura 12.10 verifica la presenza di stati ripetuti solamente sul cammino che unisce la radice allo stato corrente. Supponete che, oltre a ciò, l'algoritmo memorizzi *ogni* stato visitato ed esegua un controllo anche in quella lista (per un esempio, v. RICERCA-GRAFO nella Figura 3.19). Determinate che tipo di informazione dev'essere memorizzata e come dovrebbe usarla l'algoritmo quando trova uno stato ripetuto (*suggerimento*: dovrete distinguere almeno gli stati per cui è già stato costruito un sottopiano di successo da quelli per cui non è stato possibile trovarne alcuno). Spiegate com'è possibile usare etichette per evitare di avere in memoria copie multiple dello stesso sottopiano.
- 12.16 Spiegate precisamente come modificare l'algoritmo RICERCA-GRAFO-AND-OR per generare un piano ciclico se non ne esiste nessuno privo di cicli. Dovrete considerare tre problemi: etichettare i passi in modo che un piano ciclico possa puntare a una sua parte precedente, modificare RICERCA-OR in modo che continui a cercare piani aciclici anche dopo averne trovato uno ciclico, e arricchire la rappresentazione dei piani in modo che sia possibile indicare se un piano è ciclico. Mostrate il funzionamento del vostro algoritmo su (a) il mondo dell'aspirapolvere "triplo Murphy" e (b) il mondo dell'aspirapolvere "doppio Murphy alternativo". Potete anche verificare i vostri risultati utilizzando un'implementazione su computer. È possibile scrivere un piano per il caso (b) usando la sintassi standard dei cicli?
- 12.17 Specificate nei dettagli la procedura di aggiornamento degli stati-credenza negli ambienti parzialmente osservabili, ovvero il metodo per calcolare la nuova rappresentazione dello stato-credenza (come lista di proposizioni di conoscenza) partendo dallo stato-credenza corrente e da una descrizione di azione con effetti condizionali.



- 12.18 Scrivete descrizioni di azioni analoghe all'Equazione (12.2) per le azioni *Destra* e *Aspira*. Scrivete anche una descrizione di *ControllaPosizione* analoga all'Equazione (12.3). Ripetete l'esercizio usando l'insieme di proposizioni alternative dell'Esercizio 12.11.
- 12.19 Alla fine del Paragrafo 12.5 abbiamo elencato alcune cose che un agente di ripianificazione non può fare. Abbozzate un algoritmo che può gestire una o più di esse.
- 12.20 Considerate il problema seguente: un paziente arriva nello studio di un dottore manifestando sintomi che potrebbero essere causati dalla disidratazione o dalla malattia *D* (ma non entrambe). Ci sono due azioni possibili: *Bevi*, che cura incondizionatamente la disidratazione, e *Medica*, che cura la malattia *D* ma ha degli effetti collaterali spiacevoli se il paziente è disidratato. Scrivete una descrizione del problema in PDDL e disegnate il diagramma di un piano senza sensori che risolve il problema, enumerando tutti i mondi possibili rilevanti.
- 12.21 Aggiungete al problema medico del precedente esercizio un'azione *Test* che ha l'effetto condizionale *ColturaBatterica* quando *Malattia* vale true e in ogni caso ha l'effetto percettivo *Conosciuta(ColturaBatterica)*. Disegnate il diagramma di un piano condizionale che risolve il problema e minimizza l'uso dell'azione *Medica*.

# Conclusioni

parte offerta

## Capitolo 26

# Fondamenti filosofici

*Nel quale consideriamo cosa significa pensare, e se gli artefatti possano e debbano farlo.*

Come abbiamo detto nel Capitolo 1, da molto prima che esistessero i computer i filosofi si sono posti delle domande che hanno a che fare con l'IA: come funziona la mente? È possibile che delle macchine agiscano in maniera intelligente allo stesso modo degli uomini, e se lo fosse, avrebbero una mente? Quali sono le implicazioni etiche dell'esistenza di macchine intelligenti? Per tutto il libro ci siamo occupati dell'IA stessa, ora per un capitolo considereremo la disciplina dal punto di vista filosofico.

Prima di tutto, un po' di terminologia: l'asserzione che le macchine potrebbero agire in modo intelligente (o meglio, *come se* fossero intelligenti) è chiamata dai filosofi ipotesi dell'IA debole, l'asserzione che le macchine che fanno ciò stiano *effettivamente* pensando (e non solo *simulando* il pensiero) è chiamata ipotesi dell'IA forte.

La maggior parte dei ricercatori nel campo dell'intelligenza artificiale dà per scontata l'ipotesi dell'IA debole, e non si preoccupa dell'altra: finché i programmi funzionano non gli interessa se viene chiamata intelligenza simulata o intelligenza reale. Tutti i ricercatori, comunque, dovrebbero preoccuparsi delle implicazioni etiche del loro lavoro.

IA debole

IA forte

## 26.1 IA debole: le macchine possono agire in modo intelligente?

Alcuni filosofi hanno cercato di dimostrare che l'IA è impossibile e che le macchine non hanno alcuna possibilità di agire in modo intelligente. Alcuni hanno usato le loro argomentazioni per invocare un arresto della ricerca nell'IA:

L'intelligenza artificiale *perseguita all'interno del culto del computazionalismo* non ha la minima possibilità di produrre risultati durevoli... è tempo di indirizzare gli sforzi dei ricercatori – e i considerevoli fondi a loro disposizione – in direzioni diverse dall'approccio computazionale. (Sayre, 1993)

Chiaramente, che l'IA sia possibile o meno dipende dalla sua definizione. Nella sua essenza, l'IA è la ricerca del miglior programma agente per una specifica architettura. Secondo questa formulazione, è possibile per definizione: per ogni architettura digitale con  $k$  bit di memoria ci sono esattamente  $2^k$  programmi agenti, e per trovare quello migliore basta enumerarli e provarli tutti. Questo potrebbe essere impraticabile con  $k$  grandi, ma i filosofi si occupano degli aspetti teorici, non pratici.

La nostra definizione di IA si applica bene al problema ingegneristico di trovare un buon agente data un'architettura. Di conseguenza potremmo essere tentati di chiudere qui il paragrafo, dando una risposta affermativa alla domanda del titolo. Ma i filosofi sono interessati al confronto tra due architetture: quella umana e quella delle macchine. Inoltre, tradizionalmente il problema è stato formulato come “le macchine possono pensare?”. Sfortunatamente, in questi termini la questione è mal posta. Per vedere perché, considerate le seguenti domande:

- ◆ Le macchine possono volare?
- ◆ Le macchine possono nuotare?

La gente solitamente è d'accordo sul fatto che la risposta alla prima domanda è sì – gli aerei possono volare – ma alla seconda è no: le navi e i sottomarini possono navigare, ma non diremmo mai che nuotano. Né le domande né le risposte, comunque, hanno alcun effetto sul lavoro degli ingegneri aeronautici e navali o sugli utilizzatori dei loro prodotti. Le risposte in effetti non hanno nulla a che vedere con la progettazione o le capacità degli aerei e dei sottomarini, ma piuttosto con la nostra scelta delle parole. Il termine “nuotare” in linguaggio naturale ha assunto il significato di “spostarsi nell'acqua mediante il movimento di parti del corpo”, laddove “volare” non è limitato a un particolare mezzo fisico di locomozione.<sup>1</sup> La possibilità pratica di avere “macchine pensanti” risale a soli cinquant'anni fa, e non c'è stato abbastanza tempo per accordarsi sul significato della parola “pensare”.

le macchine possono pensare?

<sup>1</sup> In russo, il termine “nuotare” *si applica* effettivamente alle navi.

Alan Turing, nel suo celebre “Computing Machinery and Intelligence” (Turing, 1950), suggerì che invece di chiederci se le macchine possono pensare dovremmo domandarci se possono superare un test comportamentale di intelligenza, che da allora è stato chiamato test di Turing. Il test consiste nel sostenere una conversazione (attraverso messaggi digitati su una tastiera) con un interlocutore per 5 minuti. Alla fine la persona che ha posto le domande deve indovinare se la conversazione ha avuto luogo con un programma oppure con un essere umano; il programma passa il test se riesce a ingannare il suo interlocutore tre volte su dieci. Turing predisse che, entro l’anno 2000, un computer con  $10^9$  unità di memoria avrebbe potuto essere programmato abbastanza bene da superare il test, ma si sbagliava. Alcuni *sono* stati effettivamente ingannati per 5 minuti; il programma ELIZA e il chatbot MGONZ, ad esempio, hanno indotto in errore esseri umani che non hanno capito che stavano parlando a un programma, e ALICE ha ingannato un giudice durante la competizione del 2001 per il Premio Loebner. Nessun programma comunque si è avvicinato alla soglia del 30% con giudici addestrati, e in generale si può dire che l’IA nella sua interezza non ha prestato molta attenzione ai test di Turing.

Turing ha anche preso in esame una grande varietà di eventuali obiezioni alla possibilità che esistano macchine pensanti, tra cui praticamente tutte quelle sollevate nel mezzo secolo successivo alla pubblicazione del suo articolo: ne presenteremo qualcuna.

## L’argomentazione derivante dall’incapacità

L’“argomentazione derivante dall’incapacità” si basa sull’asserzione che “una macchina non potrà mai fare *X*”. Come esempi di *X* Turing ha elencato le seguenti capacità:

Essere gentile, pieno di risorse, bello, amichevole, avere iniziativa, senso dello humor, riconoscere ciò che è giusto e sbagliato, fare errori, innamorarsi, godersi una coppa di fragole e gelato, far sì che qualcuno la apprezzi, imparare dall’esperienza, usare le parole correttamente, essere l’oggetto del proprio pensiero, esibire una diversità di comportamenti pari a quella di un essere umano, fare qualcosa di veramente nuovo.

Turing dovette usare il proprio intuito per indovinare quello che sarebbe stato possibile fare in futuro, ma noi ora abbiamo la comodità di poter guardare indietro a ciò che i computer hanno già fatto. Non si può negare che oggi i computer svolgono attività che precedentemente erano appannaggio dei soli esseri umani. Ci sono programmi che giocano a scacchi, dama e altri giochi, ispezionano le parti su una linea di assemblaggio, verificano l’ortografia dei documenti, pilotano macchine ed elicotteri, diagnosticano malattie e svolgono centinaia di compiti altrettanto bene o meglio degli esseri umani. I computer hanno fatto piccole ma significative scoperte in astronomia, matematica, chimica, mineralogia, biologia, informatica e in altri campi. Ognuna di queste ha richiesto prestazioni paragonabili a quelle di un esperto umano.

Dato quello che sappiamo oggi sui computer, non ci sorprende che si comportino così bene in problemi combinatori come gli scacchi. Ma gli algoritmi possono eseguire attività che apparentemente richiedono una capacità di giudizio umana o, per dirla come Turing, la capacità di “apprendere dall’esperienza” e di “riconoscere ciò che è giusto e sbagliato”. Già nel 1955, Paul Meehl (v. anche Grove e Meehl, 1996) studiò i processi decisionali di molti esperti su attività soggettive come la predizione del successo di uno studente o della probabilità di recidiva di un criminale. In 19 casi su 20, Meehl trovò che semplici algoritmi statistici di apprendimento (come la regressione lineare o il metodo di Bayes) formulavano predizioni migliori di quelle degli esperti. Dal 1999, l’Educational Testing Service ha usato un programma automatico per valutare milioni di elaborati per l’esame GMAT. Il programma è in accordo con gli esaminatori umani il 97% delle volte, più o meno lo stesso livello di concordanza che si ottiene scegliendo a caso due persone (Burstein et al., 2001).

È chiaro che i computer possono fare molte cose altrettanto bene o meglio degli uomini, tra cui alcune che si pensa richiedano una grande quantità di intuizione e comprensione umana. Ciò non significa, naturalmente, che i computer ricorrono all’intuizione e alla comprensione (che non fanno parte del *comportamento*, e di cui ci occuperemo altrove): il punto è che spesso ci si sbaglia quando si ipotizzano i processi mentali richiesti per produrre un determinato comportamento. È anche vero, ovviamente, che ci sono molte attività in cui i computer non sono ancora in grado di eccellere (per usare un eufemismo), tra cui quella richiesta da Turing: sostenere una conversazione ad argomento libero.

## L’obiezione matematica

È ben noto, grazie al lavoro di Turing (1936) e Gödel (1931), che ad alcune questioni matematiche non si può dare risposta attraverso particolari sistemi formali: il teorema di incompletezza di Gödel (v. Paragrafo 9.5) ne è l’esempio più famoso. Brevemente, in ogni sistema formale di assiomi  $F$  abbastanza potente da gestire l’aritmetica è possibile costruire una cosiddetta “formula di Gödel”  $G(F)$  con le seguenti caratteristiche:

- ◆  $G(F)$  è una formula di  $F$ , ma non può essere dimostrata al suo interno
- ◆ se  $F$  è consistente, allora  $G(F)$  è vera.

Filosofi come J. R. Lucas (1961) hanno sostenuto che questo teorema dimostra che le macchine sono mentalmente inferiori agli esseri umani, perché essendo sistemi formali sono limitati dal teorema di incompletezza: non possono stabilire il valore di verità della propria formula di Gödel, mentre gli esseri umani non hanno questa limitazione. Quest’argomentazione ha causato decadi di controversie, dando origine a una vasta letteratura che include due libri del matematico Sir Roger Penrose (1989, 1994) che riprendono l’argomentazione aggiungendo nuovi particolari (come l’ipotesi che gli umani siano diversi perché i loro cervelli funzionano grazie alla gravità dei quanti). Esamineremo solo tre aspetti problematici.

Prima di tutto, il teorema di incompletezza di Gödel si applica solo a sistemi formali abbastanza potenti da gestire l'aritmetica. Questo include le macchine di Turing, e l'argomentazione di Lucas si basa infatti in parte sull'asserzione che i computer sono macchine di Turing. Questa è una buona approssimazione, ma non è proprio vero. Le macchine di Turing sono infinite, mentre i computer hanno dimensione finita, e ogni computer può quindi essere descritto come un sistema (molto grande) di logica proposizionale, non soggetto al teorema di incompletezza di Gödel.

In secondo luogo, un agente non dovrebbe vergognarsi troppo di non riuscire a stabilire la verità di una formula, anche quando altri agenti possono farlo. Considerate la formula

J. R. Lucas non può asserire in modo consistente che questa formula è vera.

Se Lucas ne affermasse la verità si starebbe contraddicendo, quindi Lucas non può farlo in modo consistente, e quindi la formula dev'essere vera (non può essere falsa, perché se così fosse Lucas non potrebbe asserire la sua verità, cosa che la renderebbe ancora vera). Abbiamo così dimostrato che esiste una formula di cui Lucas non può asserire in modo consistente la verità, anche se altre persone (e macchine) possono farlo. Ma questo non ci fa stimare meno Lucas. Per fare un altro esempio, nessun essere umano può calcolare la somma di 10 miliardi di numeri di 10 cifre in tutta la sua vita, ma un computer lo può fare in pochi secondi; eppure non pensiamo che questa sia una limitazione fondamentale nella capacità umana di pensare. Gli esseri umani si sono comportati in modo intelligente per millenni prima di inventare la matematica, per cui è probabile che il ragionamento matematico abbia un ruolo solo marginale in ciò che significa essere intelligenti.

Il terzo punto è forse il più importante: anche se accettiamo che i computer siano limitati in ciò che possono dimostrare, nulla prova che gli esseri umani siano immuni da tali limitazioni. È fin troppo facile dimostrare rigorosamente che un metodo formale non può fare *X*, e poi affermare che gli esseri umani *possono* fare *X* grazie a qualche metodo informale, senza addurre nessuna motivazione. In effetti, è impossibile dimostrare che gli esseri umani non sono soggetti al teorema di incompletezza di Gödel, perché ogni prova rigorosa conterebbe essa stessa una formalizzazione del talento umano che si sostiene non formalizzabile, e sarebbe quindi una refutazione di se stessa. Tutto quel che resta quindi è un appello all'intuizione che gli esseri umani possano in qualche modo eseguire azioni sovrumanne di insight matematico. Quest'appello è espresso da argomenti come "per consentire l'esistenza del pensiero, dobbiamo dare per scontata la nostra stessa consistenza" (Lucas, 1976). Ma se una cosa si può dire delle persone, è che sono notoriamente inconsistenti. Questo è certamente vero per il ragionamento di tutti i giorni, ma lo è anche per il ragionamento matematico più accurato. Un esempio famoso è il problema di coloratura di mappe con quattro colori: Alfred Kempe ha pubblicato una dimostrazione nel 1879 che fu ampiamente accettata e contribuì alla sua elezione a Fellow della Royal Society. Nel 1890 tuttavia Percy Heawood portò alla luce un difetto nella dimostrazione, e il teorema rimase indimostrato fino al 1977.

## L'argomentazione derivante dall'informalità

Una delle critiche più influenti e durature dell'IA come impresa scientifica fu sollevata da Turing stesso come l'"argomentazione derivante dall'informalità del comportamento". Essenzialmente, il comportamento umano sarebbe di gran lunga troppo complesso per essere catturato da un semplice insieme di regole, e dato che i computer non possono far altro che seguire regole, non potranno generare un comportamento intelligente come quello degli umani. In IA, l'incapacità di catturare tutto in un insieme di regole logiche si chiama **problema di qualificazione** (v. Capitolo 10).

Il principale sostenitore di questo punto di vista è stato il filosofo Hubert Dreyfus, che ha prodotto una serie di importanti critiche dell'intelligenza artificiale: *What Computers Can't Do* (1972), *What Computers Still Can't Do* (1992) e, con suo fratello Stuart, *Mind Over Machine* (1986).

La posizione che sottopongono a critica è stata denominata "Good Old-Fashioned AI", o GOFAI, un termine coniato da Haugeland (1985). Si suppone che GOFAI asserisca che tutto il comportamento intelligente può essere catturato da un sistema che ragiona logicamente partendo da un insieme di fatti e regole che descrivono il dominio; corrisponderebbe quindi al più semplice tra gli agenti logici descritti nel Capitolo 7. Dreyfus ha ragione a sostenere che gli agenti logici sono vulnerabili al problema della qualificazione. Come abbiamo visto nel Capitolo 13, nei domini aperti sono più appropriati i sistemi di ragionamento probabilistico. La critica di Dreyfus quindi non è diretta contro i computer in sé, ma piuttosto contro un particolare modo di programmarli. È ragionevole supporre, comunque, che un libro intitolato *What First-Order Logical Rule-Based Systems Without Learning Can't Do* avrebbe avuto un impatto molto inferiore.

Dal punto di vista di Dreyfus, la competenza umana include effettivamente alcune regole, ma solo come "contesto olistico" o "cornice" all'interno di cui operano gli esseri umani. Egli porta ad esempio il comportamento sociale corretto quando si ricevono e si offrono regali: "normalmente si risponde semplicemente nelle circostanze appropriate offrendo un regalo appropriato". Apparentemente, le persone hanno "una sensazione diretta di come funzionano le cose e di ciò che ci si deve aspettare". La stessa argomentazione è ripetuta nel contesto del gioco degli scacchi: "un maestro potrà aver bisogno di calcolare la mossa da fare, ma a un grande maestro sembrerà semplicemente che la scacchiera richieda una certa mossa... la risposta giusta si manifesta nella sua testa". È certamente vero che gran parte dei processi mentali di chi presenta un dono o gioca a scacchi da grande maestro si svolgono a un livello che non è aperto all'introspezione consci della mente; ma questo non significa che i processi mentali non esistano. La questione fondamentale a cui Dreyfus non dà risposta è *come* fa la mossa giusta a comparire nella testa del grande maestro. Viene in mente il commento di Daniel Dennett (1984),

È come se i filosofi si proclamassero esperti di illusionismo in grado di spiegare tutte le tecniche dei maghi. Quando poi chiediamo loro come funziona il trucco della donna tagliata a metà, rispondono che è abbastanza ovvio: il mago non la sega veramente in due, sembra solo che lo faccia. “Ma *come fa?*”, chiediamo. “Questo non ci riguarda”, rispondono i filosofi.

Dreyfus e Dreyfus (1986) propongono un processo di acquisizione dell’esperienza a cinque livelli, cominciando dal ragionamento basato su regole (del tipo proposto da GOFAI) e terminando con la capacità di scegliere istantaneamente le risposte corrette. Facendo questa proposta, i due Dreyfus passano effettivamente dalla posizione di critici a quella di teorici dell’IA, proponendo un’architettura di rete neurale organizzata come una grande “libreria di casi” ed evidenziandone alcuni problemi. Fortunatamente, i loro problemi sono stati tutti affrontati: alcuni sono stati risolti totalmente, per altri si è avuto un successo parziale. I problemi includono:

1. Una buona generalizzazione dagli esempi non può essere ottenuta senza conoscenza di fondo. Essi sostengono che nessuno ha idea di come incorporare conoscenza di fondo nel processo di apprendimento delle reti neurali. In effetti, come vediamo nel Capitolo 19, ci sono tecniche per utilizzare conoscenza preesistente negli algoritmi di apprendimento. Queste tecniche, comunque, si basano sulla disponibilità di conoscenza in forma esplicita, una cosa che Dreyfus e Dreyfus continuano strenuamente a negare. Dal nostro punto di vista, questa è una buona ragione per riconsiderare seriamente i modelli correnti di elaborazione neurale in modo che *possano* trarre vantaggio dalla conoscenza precedentemente appresa, come gli altri algoritmi di apprendimento.
2. L’apprendimento delle reti neurali è una forma di apprendimento supervisionato (v. Capitolo 18), che richiede la precedente identificazione degli input rilevanti e degli output corretti. Di conseguenza, sostengono, non può funzionare autonomamente senza l’aiuto di un insegnante umano. In effetti, l’apprendimento senza insegnanti può essere ottenuto con l’**apprendimento non supervisionato** (v. Capitolo 20) e l’**apprendimento per rinforzo** (v. Capitolo 21).
3. Gli algoritmi di apprendimento non funzionano bene in presenza di molte caratteristiche, e se ne sceglio un sottoinsieme, “non esiste metodo comprovato per aggiungere nuove caratteristiche qualora l’insieme corrente dovesse dimostrarsi inadeguato a gestire i fatti appresi”. In realtà, nuovi metodi come le macchine a vettore di supporto possono gestire molto bene insiemi di caratteristiche molto ampi. Come abbiamo visto nel Capitolo 19, esistono anche metodi comprovati per generare nuove caratteristiche, benché sia necessaria ancora molta ricerca.

4. Il cervello è capace di dirigere i sensori per cercare informazione desiderata ed elaborarla per estrarre gli aspetti rilevanti per la situazione corrente. Ma, secondo loro, “oggi i dettagli di questo meccanismo non sono affatto compresi e neppure ipotizzati in un modo che potrebbe indirizzare la ricerca nell’IA”. In realtà il campo della visione attiva, supportato dalla teoria del valore dell’informazione (v. Capitolo 16), si occupa precisamente del problema di dirigere i sensori, e alcuni robot hanno già incorporato alcuni dei risultati teorici ottenuti.

In definitiva, molti dei problemi su cui i Dreyfus si sono focalizzati – conoscenza di fondo e buon senso, il problema della qualificazione, incertezza, apprendimento, forme compilate di processi decisionali, l’importanza di considerare agenti situati anziché motori di inferenza astratti – sono stati ormai incorporati nella progettazione standard degli agenti intelligenti. Per noi tutto questo dimostra il progresso dell’IA, non la sua impossibilità.

## 26.2 IA forte: le macchine possono veramente pensare?

Molti filosofi sostengono che una macchina che dovesse passare il test di Turing non starebbe *realmente* pensando, ma starebbe solo *simulando* il pensiero. Ancora una volta, quest’obiezione è stata prevista da Turing stesso. Egli cita un discorso del Professor Geoffrey Jefferson (1949):

Solo quando una macchina avrà scritto un sonetto o composto un concerto in virtù dei pensieri e delle emozioni provate, e non per la disposizione casuale dei simboli, potremo essere d’accordo sul fatto che egualga il cervello: non deve solo aver scritto, ma avere la conoscenza di aver scritto.

Turing definisce questa l’argomentazione della **coscienza**: la macchina dev’essere consapevole dei propri stati mentali e delle proprie azioni. Benché la coscienza sia un argomento importante, il nocciolo del discorso di Jefferson ha piuttosto a che fare con la **fenomenologia**, o lo studio dell’esperienza diretta: la macchina deve provare vere emozioni. Altri si sono focalizzati sull’intenzionalità, ovvero sulla questione se i presunti desideri, credenze e altre rappresentazioni della macchina “riguardino” veramente qualcosa nel mondo reale.

La risposta di Turing a quest’obiezione è interessante. Invece di addurre prove del fatto che le macchine possono avere coscienza (o fenomenologia, o intenzioni), egli sostiene che la questione è mal definita, proprio come domandarsi “le macchine possono pensare?”. Inoltre, perché dovremmo pretendere dalle macchine uno standard più alto di quello richiesto agli esseri umani? Dopotutto, nella vita di tutti i giorni non abbiamo mai *alcuna* prova diretta degli stati mentali degli

altri esseri umani. Nonostante questo, Turing nota che “invece di continuare a discutere su questo punto, normalmente adottiamo l'**educata convenzione** che tutti pensino”.

Turing sostiene che Jefferson sarebbe disposto a estendere alle macchine l'**educata convenzione**, se solo avesse esperienza di automi in grado di comportarsi intelligentemente. Egli cita il seguente dialogo, che è entrato a far parte della tradizione orale dell'IA a tal punto che non possiamo semplicemente non citarlo:

UMANO: Nel primo verso del tuo sonetto “Dovrei paragonarti a un giorno d'estate?”, non pensi che “giorno di primavera” funzioni ugualmente bene, se non meglio?

MACCHINA: Non sta nella metrica.

UMANO: Che dire di “giorno d'inverno”? Quello sta bene.

MACCHINA: Sì, ma nessuno vuole essere paragonato a un giorno d'inverno.

UMANO: Diresti che il Sig. Pickwick ti ricorda il Natale?

MACCHINA: In un certo senso.

UMANO: Eppure il Natale è un giorno d'inverno, e non penso che al Sig. Pickwick spiacerebbe il paragone.

MACCHINA: Non penso che tu stia parlando seriamente. Dicendo “giorno d'inverno” si intende un giorno tipico, e non uno speciale come il Natale.

Turing concorda che la questione della consapevolezza sia difficile da trattare, ma nega che abbia molta rilevanza per quanto riguarda la pratica dell'IA: “non voglio dare l'impressione di ritener che non ci sia alcun mistero riguardo alla coscienza... ma non penso che questi misteri debbano necessariamente essere risolti prima che sia possibile rispondere alle questioni che ci interessano in quest'articolo”. Siamo d'accordo con Turing: quello che ci interessa è creare programmi che si comportano in modo intelligente, non se qualcun altro deciderà che l'intelligenza è reale o simulata. D'altra parte, molti filosofi sono profondamente interessati a questo punto: per comprenderlo meglio, considereremo la questione della realtà di altri tipi di artefatti.

Nel 1848 Frederick Wöhler sintetizzò per la prima volta l'urea artificiale. Questo fu un enorme passo avanti perché dimostrò per la prima volta che la chimica organica e quella inorganica potevano essere unificate, una questione che era stata dibattuta in modo molto acceso. Una volta completata la sintesi, i chimici furono d'accordo che quella *era* urea, perché ne aveva tutte le caratteristiche fisiche. In modo analogo, non si può negare che i dolcificanti artificiali dolcifichino, e l'inseminazione artificiale (l'altra IA) è senza dubbio un'inseminazione. D'altra parte, i fiori artificiali non sono fiori, e Daniel Dennett puntualizza che un vino Château Latour artificiale non sarebbe Château Latour, anche se fosse chimicamente indistinguibile, semplicemente perché non sarebbe prodotto nel posto giusto nel mo-

do giusto. Ugualmente un Picasso artificiale non si può definire un dipinto di Picasso, in modo del tutto indipendente dal suo aspetto.

Possiamo concludere che in alcuni casi quello che conta è il comportamento di un artefatto, mentre altre volte è più importante il suo *pedigree*. Che cosa sia importante in quale caso sembra essere una questione di convenzione. Per le menti artificiali, tuttavia, non c'è una convenzione; ci si deve quindi affidare all'intuizione. Il filosofo John Searle (1980) ne ha una molto forte:

Nessuno si sogna di pensare che la simulazione al computer di una tempesta ci debba lasciare fradici... perché mai chiunque, sano di mente, dovrebbe supporre che una simulazione al computer di processi mentali debba possedere realmente dei processi mentali? (pp. 37–38)

Mentre è facile concordare sul fatto che le simulazioni elettroniche di tempeste non ci bagnano, non è chiaro come trasportare quest'analogia alle simulazioni al computer di processi mentali. Dopotutto, una simulazione hollywoodiana di una tempesta a base di pompe e macchine del vento *bagna* effettivamente gli attori. La maggior parte della gente non ha problemi a dire che la simulazione al computer di un'addizione è un'addizione, e una simulazione al computer di una partita a scacchi è una partita a scacchi. I processi mentali sono come le tempeste, o sono più simili alle addizioni o agli scacchi? Sono Château Latour e Picasso, o urea? Dipende tutto dalle nostre teorie degli stati e dei processi mentali.

La teoria del **funzionalismo** afferma che uno stato mentale è qualsiasi condizione causale intermedia tra input e output. Secondo la teoria funzionalista, due sistemi qualsiasi con processi causali isomorfi avrebbero gli stessi stati mentali; quindi anche un computer potrebbe avere gli stessi stati mentali di una persona. Naturalmente, non abbiamo ancora detto che cosa significa veramente "isomorfi", ma l'assunto è che ci sia un qualche livello di astrazione sotto il quale i dettagli della specifica implementazione non hanno importanza; finché i processi sono isomorfi fino a quel livello, si verificheranno gli stessi stati mentali.

Al contrario, la teoria del **naturalismo biologico** afferma che gli stati mentali sono caratteristiche emergenti di alto livello causate da processi neurologici di basso livello *all'interno dei neuroni*, e che quello che conta sono le proprietà (non specificate) dei neuroni stessi. Gli stati mentali non potranno quindi essere considerati duplicati solo perché un programma ha la stessa struttura funzionale con lo stesso comportamento di input-output; il programma dovrà andare in esecuzione su un'architettura con lo stesso potere causale dei neuroni. La teoria non spiega perché i neuroni hanno questo potere causale, né quali altre istanze fisiche potrebbero o non potrebbero averlo.

Per investigare questi due punti di vista esamineremo prima di tutto uno dei più antichi problemi della filosofia della mente, quindi prenderemo in considerazione tre esperimenti sul pensiero.

funzionalismo

naturalismo biologico

## Il problema mente-corpo

Il **problema mente-corpo** si chiede in che modo gli stati e i processi mentali siano correlati agli stati e ai processi del corpo (e precisamente, del cervello). Come se non fosse già abbastanza difficile, noi lo generalizzeremo nel problema “mente-architettura”, per discutere la possibilità che le macchine abbiano una mente.

Perché il rapporto mente-corpo costituisce un problema? La prima difficoltà risale a Cartesio, che considerò l’interazione tra un’anima immortale e un corpo mortale e concluse che anima e corpo sono due tipi distinti di cose: la sua è la teoria del **dualismo**. La teoria **monista**, spesso chiamata **materialismo**, sostiene che non esistono cose come un’anima immateriale, ma solo oggetti materiali. Di conseguenza gli stati mentali – come provare dolore, sapere che si sta andando a cavallo o credere che Vienna sia la capitale dell’Austria – sono stati del cervello. John Searle riassume efficacemente l’idea con lo slogan “*i cervelli causano le menti*”.

Il materialista si trova ad affrontare almeno due seri ostacoli. Il primo è il **libero arbitrio**: come può essere che una mente puramente fisica, in cui ogni trasformazione è governata unicamente dalle leggi della fisica, mantenga ancora una libertà di scelta? La maggior parte dei filosofi ritiene che questo problema richieda un’attenta ricostruzione della nostra ingenua concezione di libero arbitrio, piuttosto che rappresentare una minaccia per il materialismo. Il secondo problema riguarda in generale la coscienza (e le questioni correlate, ma non identiche, della comprensione e della consapevolezza di sé). Per dirla semplicemente, perché certi stati del cervello procurano delle *sensazioni*, mentre presumibilmente non si prova nulla in altri stati fisici (ad esempio, a essere una roccia)?

Per cominciare a rispondere a queste domande occorre parlare degli stati mentali a un livello più astratto della specifica configurazione di tutti gli atomi del cervello di una particolare persona in un particolare istante. Ad esempio, quando penso alla capitale dell’Austria, nel mio cervello si verifica una miriade di microscopici cambiamenti da un picosecondo all’altro, che però non costituiscono un cambiamento *qualitativo* dello stato del cervello. Per parlare di queste cose dobbiamo introdurre la nozione di *tipo* di stato mentale, in base a cui possiamo giudicare se due stati del cervello appartengono allo stesso tipo o a tipi diversi. Autori diversi hanno posizioni contrastanti su ciò che significa *tipo* in questo caso. Quasi tutti ritengono che, se prendiamo un cervello e sostituiamo alcuni atomi di carbonio con un nuovo insieme di atomi di carbonio,<sup>2</sup> lo stato mentale non sarà modificato. Questo è un bene, perché i cervelli reali sostituiscono continuamente i loro atomi attraverso i processi metabolici, eppure questo non sembra causare grandi stravolgimenti mentali.

problema mente-corpo

dualismo  
monismo  
materialismo

libero arbitrio

coscienza

<sup>2</sup> Forse persino atomi di un diverso isotopo del carbonio, come si fa talvolta negli esperimenti di scansione cerebrale.

stati intenzionali

Consideriamo ora una particolare classe di stati mentali: le **attitudini proposizionali** (incontrate per la prima volta nel Capitolo 10), note anche come **stati intenzionali**. Si tratta degli stati come credere, sapere, desiderare, temere e così via, che si riferiscono a qualche aspetto del mondo esterno. Ad esempio, la credenza che Vienna sia la capitale dell'Austria è una credenza *circa* una particolare città e il suo status. Ci chiederemo se è possibile che i computer abbiano stati intenzionali, per cui è utile capire come si possono caratterizzare tali stati. Per esempio, si potrebbe dire che lo stato mentale in cui desidero un hamburger è differente da quello in cui desidero una pizza, perché hamburger e pizza sono cose diverse nel mondo reale. In altre parole, gli stati intenzionali sono necessariamente collegati agli oggetti nel mondo esterno. D'altra parte, abbiamo sostenuto poche righe fa che gli stati mentali sono stati del cervello; di conseguenza l'identità o la diversità degli stati mentali dovrebbe essere determinata rimanendo completamente "dentro la testa", senza alcun riferimento al mondo reale. Per sciogliere questo dilemma consideriamo un esperimento sul pensiero che cerca proprio di separare gli stati intenzionali dai loro oggetti esterni.

## L'esperimento del cervello nella vasca

Immaginate che il vostro cervello sia stato rimosso dal corpo alla nascita e piazzato in una vasca meravigliosamente ingegnerizzata. La vasca mantiene in vita il cervello, permettendogli di crescere e svilupparsi. Contemporaneamente segnali elettronici sono inseriti nel cervello da una simulazione computerizzata di un mondo totalmente fittizio, mentre i segnali motori provenienti dal cervello sono intercettati e utilizzati per modificare la simulazione in modo appropriato.<sup>3</sup> In questo caso il cervello potrebbe essere nello stato mentale *MorendoDiVoglia(Io, Hamburger)* anche se non c'è corpo che provi fame né papille per provare gusto, e potrebbe anche non esserci alcun hamburger nel mondo reale. In tal caso, quello stato mentale sarebbe lo stesso di quello provato da un cervello contenuto in un corpo?

Un modo di risolvere il dilemma è dire che il contenuto degli stati mentali può essere interpretato da due diversi punti di vista. Quello del "**contenuto allargato**" è il punto di vista di un osservatore esterno onnisciente che ha accesso all'intera situazione e può distinguere le differenze nel mondo: in questo caso, le credenze del cervello nella vasca sono diverse da quelle di una persona "normale". Il "**contenuto stretto**" considera solo il punto di vista interno soggettivo, e in questo caso le credenze sarebbero identiche.

Credere che un hamburger sia delizioso ha una certa natura intrinseca: avere questa credenza dà una sensazione particolare. Qui si entra nell'ambito dei **qualia**, o esperienze intrinseche (dal termine latino che significa, a grandi linee, "cose co-

contenuto allargato

contenuto stretto

qualia

<sup>3</sup> Questa situazione potrebbe essere familiare a quelli che hanno visto il film del 1999, *The Matrix*.

me queste”). Supponiamo che, per qualche incidente della retina o dei neuroni, la persona *X* percepisca come rosso il colore che la persona *Y* vede come verde, e viceversa. Quando entrambi sono davanti a un semaforo agiranno allo stesso modo, ma la loro *esperienza* sarà in qualche modo diversa. Entrambi potrebbero concordare sul fatto che il nome dell’esperienza è “il semaforo è rosso”, ma le sensazioni saranno differenti. Non è chiaro se questo significa che gli stati mentali sono anch’essi differenti.

Passiamo ora a un altro esperimento sul pensiero che affronta la questione se oggetti fisici diversi da neuroni umani possano avere stati mentali.

## L’esperimento della protesi cerebrale

L’esperimento della protesi cerebrale fu introdotto a metà degli anni ‘70 da Clark Glymour e discusso da John Searle (1980), ma è principalmente associato al lavoro di Hans Moravec (1988). La sua definizione è la seguente: supponiamo che la neurofisiologia si sia sviluppata al punto che il comportamento di input-output e la connettività di tutti i neuroni del cervello umano siano perfettamente compresi. Supponiamo inoltre che si possano costruire dispositivi elettronici microscopici che possono imitare tale comportamento, e che sia possibile collegarli direttamente al tessuto neurale. Infine, supponiamo che una qualche miracolosa tecnica chirurgica sia in grado di sostituire singoli neuroni con i corrispondenti dispositivi elettronici senza interrompere il funzionamento del cervello nel suo insieme. L’esperimento consiste nel rimpiazzare gradatamente tutti i neuroni nella testa di un uomo con dispositivi elettronici e quindi invertire il processo riportando il soggetto nel suo normale stato biologico.

Siamo interessati sia al comportamento esterno che all’esperienza interna del soggetto, durante e dopo l’operazione. Secondo la definizione dell’esperimento, il comportamento esterno del soggetto deve rimanere immutato in confronto a quello che si osserverebbe se l’operazione non fosse eseguita.<sup>4</sup> Ora, benché la presenza o l’assenza di coscienza non possa essere determinata facilmente da una terza parte, il soggetto dell’esperimento dovrebbe essere almeno in grado di registrare eventuali cambiamenti nella propria esperienza conscia. Apparentemente, c’è uno scontro diretto di intuizioni riguardanti quello che succederebbe. Moravec, un ricercatore di robotica e un funzionalista, è convinto che la coscienza rimarrebbe inalterata.

<sup>4</sup> Si può ipotizzare l’uso di un identico soggetto “di controllo” sottoposto a un’operazione placebo, in modo da poter confrontare i due comportamenti.

Searle, un filosofo e un naturalista biologico, è ugualmente convinto che la sua coscienza svanirebbe:

Trovi, con tua totale sorpresa, che stai effettivamente perdendo controllo del tuo comportamento esterno. Ad esempio, quando i dottori verificano la visione, li senti dire "stiamo tenendo un oggetto rosso di fronte a te; per favore dici quello che vedi". Vorresti gridare "non vedo niente, sto diventando completamente cieco", ma senti la tua voce dire, in modo completamente indipendente dal tuo controllo, "vedo un oggetto rosso di fronte a me". ... La tua esperienza cosciente si riduce lentamente fino a sparire, mentre il tuo comportamento esterno rimane immutato. (Searle, 1992)

Ma si può fare di più che argomentare basandosi sull'intuizione. Prima di tutto notate che, affinché il comportamento esterno resti lo stesso mentre il soggetto diventa gradatamente inconscio, la sua volontà dev'essere rimossa istantaneamente e in modo totale; in caso contrario il diminuire della consapevolezza si rifletterebbe nel comportamento, con effetti tipo "Aiuto, me ne sto andando!" o frasi simili. Quest'istantanea rimozione della volontà, a fronte di una sostituzione graduale dei neuroni uno per volta, sembra difficile da sostenere.

In secondo luogo, considerate che cosa accade se poniamo al soggetto delle domande riguardanti la sua esperienza cosciente durante il periodo in cui non gli rimane nessun neurone reale. Date le condizioni dell'esperimento, otterremmo risposte come "Mi sento bene. Devo dire di essere un po' sorpreso, perché credevo avesse ragione Searle". Oppure potremmo pungere il soggetto con un bastone appuntito e osservare la risposta: "Ahio, che male". Normalmente un output di questo tipo da parte di un programma di IA potrebbe essere considerato da uno scettico un semplice artificio: è certamente facile ricorrere a regole come "se il sensore 12 legge 'alto' allora l'output è 'Ahio' ". Ma il punto è che ora abbiamo replicato le proprietà funzionali di un normale cervello umano, e quindi presumiamo che il cervello elettronico non contenga simili artifici. Di conseguenza dovremo spiegare le manifestazioni di coscienza prodotte dal cervello elettronico facendo riferimento unicamente alle proprietà funzionali dei neuroni. *E questa spiegazione si deve applicare anche al cervello reale, che ha le stesse proprietà funzionali.* Ci sono, a quanto pare, due sole conclusioni possibili:

1. i meccanismi causali della coscienza che generano questo tipo di output nei cervelli normali stanno ancora operando nella versione elettronica, che è quindi cosciente;
2. gli eventi mentali consci nel cervello normale non hanno un collegamento causale con il comportamento, e non sono presenti nel cervello elettronico, che quindi non è cosciente.

Benché non possiamo escludere la seconda possibilità, questa ridurrebbe la coscienza al ruolo di quello che i filosofi chiamano **epifenomeno**: qualcosa che accade, ma che non proietta nessuna ombra, per così dire, sul mondo osservabile. Inoltre, se la coscienza fosse davvero un epifenomeno, allora il cervello dovrebbe contenere un secondo meccanismo inconscio responsabile dell'esclamazione "Ahio".

In terzo luogo, considerate la situazione dopo che l'operazione è stata invertita e il soggetto ha di nuovo un cervello normale. Ancora una volta, per definizione, il suo comportamento esterno dovrebbe essere uguale a quello di una persona che non abbia subito alcuna operazione. In particolare, dovremmo essere in grado di chiedergli, "Che cosa hai provato durante l'operazione? Ti ricordi del bastone appuntito?". Il soggetto dovrà avere memorie accurate dell'effettiva natura delle sue esperienze consce, tra cui i qualia, nonostante il fatto che secondo Searle tali esperienze non si siano neppure verificate.

Searle potrebbe replicare che non abbiamo definito l'esperimento nella maniera corretta. Se i veri neuroni sono posti, per così dire, in uno stato di animazione sospesa nel periodo che va dall'estrazione al reinserimento nel cervello, allora ovviamente non "ricorderanno" le esperienze subite durante l'operazione. Per trattare quest'eventualità, dobbiamo assicurarcene che lo stato dei neuroni sia aggiornato per riflettere lo stato interno dei neuroni artificiali che stanno rimpiazzando. Se i presunti aspetti "non funzionali" dei neuroni reali danno come risultato un comportamento funzionalmente diverso da quello osservato quando i neuroni artificiali erano ancora installati, avremo una semplice *reductio ad absurdum*, perché questo significherebbe che i neuroni artificiali non sono funzionalmente equivalenti a quelli reali (v. Esercizio 26.3 per una possibile critica di quest'argomentazione).

Patricia Churchland (1986) evidenzia il fatto che le argomentazioni funzionaliste che operano a livello del singolo neurone possono anche operare a qualsiasi livello funzionale superiore: un gruppo di neuroni, un modulo mentale, un lobo, un emisfero, l'intero cervello. Questo significa che se accettiamo il fatto che l'esperimento della protesi cerebrale dimostra che il cervello sostituito è cosciente, dobbiamo accettare che la coscienza è conservata anche quando l'intero cervello è sostituito da un circuito che mette in relazione input e output mediante un'enorme tabella di corrispondenza. Questo può sconcertare molte persone (tra cui lo stesso Turing) che intuitivamente ritengono che le tabelle di corrispondenza non abbiano coscienza, o quantomeno che le esperienze consce generate durante l'uso della tabella non siano uguali a quelle generate dal funzionamento di un sistema caratterizzato (anche nella semplice accezione computazionale) dall'uso e dalla generazione di credenze, intuizioni, obiettivi e così via. Questo potrebbe suggerire che l'esperimento della protesi cerebrale non può essere effettuato rimpiazzando interi cervelli se il suo scopo è guidare efficacemente le intuizioni, ma non significa neppure che sia obbligato a rimpiazzare un atomo per volta, come Searle vorrebbe farci credere.

## La stanza cinese

Il nostro esperimento finale sul pensiero è forse il più famoso di tutti. Dovuto a John Searle (1980), descrive un ipotetico sistema che sta chiaramente eseguendo un programma e passando il test di Turing, ma che altrettanto chiaramente (secondo Searle) non *capisce* nulla dei suoi input e output. La sua conclusione è che l'esecuzione di un programma appropriato (cioè, il fatto di esibire gli output giusti) non è una condizione *sufficiente* per essere considerati una mente.

Il sistema consiste in un essere umano, che parla solo inglese ed è dotato di un libro di regole scritto in inglese e di varie pile di fogli, alcuni bianchi, altri contenenti iscrizioni indecifrabili (l'uomo quindi ricopre il ruolo di una CPU, il libro di regole è il programma e le pile di fogli la memoria). Il sistema è chiuso in una stanza con una piccola apertura verso l'esterno. Attraverso l'apertura appaiono striscioline di carta che riportano segni indecifrabili. L'uomo può cercare i simboli sul suo libretto di regole e seguire le relative istruzioni, che possono includere la scrittura di simboli su nuove strisce di carta, la ricerca di particolari simboli nelle pile di fogli, la modifica dell'ordine dei fogli e così via. A un certo punto, le istruzioni causeranno la scrittura di uno o più simboli su un pezzo di carta che sarà poi ri-passato al mondo esterno.

Fin qui, tutto bene. Dal punto di vista esterno, però, quello che osserviamo è un sistema che accetta input sotto forma di frasi in cinese e genera risposte in cinese che sono palesemente altrettanto “intelligenti” di quelle nella conversazione immaginata da Turing.<sup>5</sup> Searle argomenta come segue: la persona nella stanza non capisce il cinese (fatto assodato). Il libro di regole e le pile di fogli, essendo semplici carta, non capiscono il cinese. Quindi, da nessuna parte c’è comprensione del cinese. *Di conseguenza, secondo Searle, eseguire il programma giusto non genera necessariamente comprensione.*

Come Turing, Searle considerò e cercò di ribattere a un certo numero di critiche alla sua argomentazione. Molti commentatori, tra cui John McCarthy e Robert Wilensky, proposero quella che Searle chiama la risposta del sistema. L’obiezione sta nel fatto che chiedere se l'uomo nella stanza capisce il cinese è come chiedere se una CPU può calcolare radici cubiche. In entrambi i casi la risposta è no, ma è altrettanto vero, secondo la risposta del sistema, che l’intero sistema *ha* la capacità in questione. Certamente, se qualcuno chiedesse alla stanza cinese se comprende il cinese, la risposta (in fluente cinese) sarebbe affermativa. Secondo l’educa-ta convenzione di Turing, questo dovrebbe bastarci. La risposta di Searle è di ripetere che non c’è comprensione nell’essere umano né nella carta, quindi non ci può essere da nessuna parte. Suggerisce inoltre che si potrebbe immaginare che l'uomo memorizzi il libro di regole e il contenuto di tutte le pile di fogli, dimo-doché non ci sarebbe *nient’altro* che l’essere umano; ancora una volta, chi dovesse fa-re la domanda all'uomo (in inglese) riceverebbe una risposta negativa.

Ora arriviamo al punto. Il passaggio dalla carta alla memorizzazione è quella che si dice un’ “aringa rossa”, cioè una divagazione fuorviante, perché entrambe le forme non sono che istanziazioni fisiche di un programma in esecuzione. La vera argomentazione di Searle è basata sui seguenti quattro assiomi (Searle, 1990).

<sup>5</sup> Il fatto che le pile di fogli possano essere più grandi dell’intero pianeta e che la generazione di risposte possa richiedere milioni d’anni non ha alcuna importanza ai fini della struttura *logica* dell’argomentazione. Uno degli scopi degli studi filosofici è sviluppare un finissimo senso di quali obiezioni siano pertinenti e quali no.

1. I programmi per computer sono entità formali e sintattiche.
2. Le menti hanno un contenuto mentale, ovvero una semantica.
3. La sintassi da sola non è sufficiente alla semantica.
4. I cervelli causano le menti.

Dai primi tre assiomi Searle conclude che i programmi non sono sufficienti a causare le menti. In altre parole, un agente che esegue un programma potrebbe essere una mente, ma non è necessariamente una mente in virtù del fatto che esegue il programma. Dal quarto assioma egli conclude, “ogni altro sistema capace di causare menti dovrebbe avere poteri causali (almeno) equivalenti a quelli dei cervelli”. Da ciò inferisce che qualsiasi cervello artificiale dovrebbe duplicare i poteri causali dei cervelli, e non limitarsi a eseguire un programma, e che i cervelli umani non producono fenomeni mentali solo a causa dell'esecuzione di un programma.

La conclusione che i programmi non sono sufficienti a causare menti *segue* effettivamente dagli assiomi, se li si interpreta in modo abbastanza ampio. Ma tale conclusione non è soddisfacente: Searle non ha fatto altro che mostrare che negando esplicitamente il funzionalismo (questo è ciò che fa l'assioma 3) non si può più concludere necessariamente che i non-cervelli sono menti. Questo è abbastanza ragionevole, per cui l'intera questione si riduce a decidere se l'assioma 3 può essere accettato. Secondo Searle, il senso dell'argomentazione della stanza cinese è fornire intuizioni per l'assioma 3. Ma le reazioni hanno mostrato che tali intuizioni sono state accolte solo da coloro che erano già inclini ad accettare l'idea che semplici programmi non possano generare vera comprensione.

Ricapitolando, lo scopo dell'argomentazione della stanza cinese è confutare l'ipotesi dell'IA forte, e cioè che l'esecuzione del programma giusto abbia necessariamente come risultato una mente. Per far questo Searle esibisce un sistema apparentemente intelligente, che esegue il giusto tipo di programma, e che (secondo lui) *si può dimostrare* non essere una mente. Per quest'ultima parte Searle si appoggia all'intuito, non a una dimostrazione: Guardate la stanza, dove sarà mai la mente? Ma la stessa argomentazione si può fare anche riguardo al cervello: guardate questa collezione di cellule (o di atomi), che funziona in base alle leggi della biochimica (o della fisica), dove sarà la mente? Perché un pezzo di cervello può essere una mente e un pezzo di fegato no?

Inoltre, Searle indebolisce ulteriormente la sua argomentazione quando ammette che in via di principio anche materiali diversi dai neuroni potrebbero costituire una mente, per due ragioni: prima di tutto, abbiamo solo le intuizioni di Searle (o le nostre proprie) per dire che la stanza cinese non è una mente; in secondo luogo, anche se decidiamo che la stanza non è una mente, questo non ci dice nulla riguardo alla possibilità che un altro programma in esecuzione su qualche mezzo fisico (tra cui un computer) possa esserlo. Searle ammette la possibilità logica che il cervello stia effettivamente implementando un programma di AI tradizio-

nale; ma lo stesso programma in esecuzione sul tipo sbagliato di macchina non sarebbe una mente. Searle ha negato di credere che “le macchine non hanno una mente”, piuttosto asserisce che alcune macchine *ce l'hanno*: gli esseri umani sono macchine biologiche dotate di mente. Alla fine, non rimane molto per decidere quale tipo di macchine si qualifichi e quale no.

## 26.3 L'etica e i rischi dello sviluppo di intelligenze artificiali

---

Fin qui ci siamo concentrati sulla domanda se *possiamo* sviluppare intelligenze artificiali, ma è necessario anche considerare se *dobbiamo* farlo. Se gli effetti della tecnologia dell'IA dovessero rivelarsi più probabilmente negativi che positivi, sarebbe una responsabilità morale dei ricercatori occuparsi d'altro. Molte nuove tecnologie hanno avuto effetti collaterali negativi imprevisti: il motore a combustione interna ha portato all'inquinamento dell'aria e ha fatto ricoprire tutto di asfalto; la fissione nucleare ha portato a Chernobyl, Three Mile Island e alla minaccia della distruzione globale. Tutti gli scienziati e gli ingegneri si trovano di fronte a considerazioni etiche su come dovrebbero agire sul lavoro, quali progetti dovrebbero o non dovrebbero essere sviluppati e con quali modalità. Esiste anche un manuale dedicato alla *Ethics of Computing* (Berleur e Brunnstein, 2001). L'AI comunque sembra porre ulteriori problemi rispetto a, poniamo, la costruzione di ponti che non crollano.

- ◆ La gente potrebbe perdere il lavoro a causa dell'automazione.
- ◆ La gente potrebbe avere troppo (o troppo poco) tempo libero.
- ◆ La gente potrebbe perdere il senso della propria unicità.
- ◆ La gente potrebbe perdere i propri diritti alla privacy.
- ◆ L'uso di sistemi di IA renderebbe impossibile determinare le responsabilità legali.
- ◆ Il successo dell'IA potrebbe significare la fine della razza umana.

Esaminiamo questi problemi uno per volta.

*La gente potrebbe perdere il lavoro a causa dell'automazione.* L'economia industriale moderna è ormai dipendente dai computer in generale, e da alcuni programmi di IA in particolare. Ad esempio, gran parte dell'economia, specialmente negli Stati Uniti, dipende dalla disponibilità di credito al consumatore. La gestione delle richieste di carte di credito, l'approvazione delle modifiche e il rilevamento di frodi sono tutte attività svolte da programmi di IA. Si potrebbe pensare che migliaia di lavoratori abbiano perso il lavoro a causa di ciò, ma in effetti senza i programmi quei posti di lavoro non esisterebbero neppure, perché aggiungerebbe-

ro un tale costo alle transazioni da renderle improponibili. Fino a oggi, l'automazione attraverso la tecnologia dell'IA ha creato più posti di lavoro di quanti ne abbiano eliminati, e i nuovi lavori sono più interessanti e meglio pagati. Oggi il programma canonico di IA è un "agente intelligente" il cui scopo è assistere un essere umano, e la perdita di posti di lavoro preoccupa meno di quando l'IA si focalizzava su "sistemi esperti" progettati per rimpiazzare gli uomini.

*La gente potrebbe avere troppo (o troppo poco) tempo libero.* Alvin Toffler scrisse in *Future Shock* (1970): "Le ore lavorative settimanali sono state tagliate del 50 per cento dall'inizio del secolo. Non è assurdo prevedere che saranno ancora dimezzate entro l'anno 2000". Arthur C. Clarke (1968b) scrisse che le persone, nel 2001, avrebbero potuto "trovarsi di fronte a un futuro di noia totale, in cui il problema principale della vita sarà decidere quale canale televisivo guardare tra diverse centinaia". L'unica di queste predizioni che è arrivata vicina ad avverarsi è la proliferazione dei canali televisivi (Springsteen, 1992). Al contrario, le persone che lavorano nelle industrie della conoscenza fanno parte di un sistema integrato computerizzato che opera 24 ore al giorno; per rimanere competitivi, hanno dovuto lavorare un numero maggiore di ore. In un'economia industriale, la ricompensa è approssimativamente proporzionale al tempo investito; lavorare il 10% in più porterà a un incremento del guadagno del 10%. In un'economia basata sull'informazione, contraddistinta da comunicazioni a banda larga e dalla facile riproducibilità della proprietà intellettuale (quella che Frank e Cook (1996) chiamano la "Società Chi-Vince-Piglia-Tutto"), la ricompensa per essere lievemente superiori alla concorrenza è molto grande; lavorare il 10% in più potrebbe significare un incremento del guadagno del 100%. Per questa ragione c'è una pressione sempre più forte affinché tutti lavorino di più. L'IA accelera il ritmo dell'innovazione tecnologica contribuendo così al fenomeno, ma promette anche di darci la possibilità di staccare per qualche tempo lasciando che i nostri agenti automatizzati si occupino di tutto.

*La gente potrebbe perdere il senso della propria unicità.* In *Computer Power and Human Reason*, Weizenbaum (1976), l'autore del programma ELIZA, evidenzia alcune delle potenziali minacce che l'IA rappresenta per la società. Una delle principali argomentazioni di Weizenbaum è che l'IA rende plausibile l'idea che gli esseri umani siano automi: un'idea che può risultare nella perdita di autonomia e financo della stessa umanità. È da notare che l'idea risale a molto prima della stessa esistenza dell'IA, come minimo a *L'Homme Machine* (La Mettrie, 1748). L'umanità, del resto, ha superato molti altri colpi inflitti al suo senso di unicità: il *De Revolutionibus Orbium Coelestium* (Copernico, 1543) scalzò la Terra dal centro del sistema solare, e l'*Origine dell'uomo* (Darwin, 1871) pose l'*Homo sapiens* allo stesso livello delle altre specie. L'IA, se dovesse ottenere ampi successi, potrebbe scuotere le ipotesi morali della società del XXI secolo almeno quanto la teoria dell'evoluzione di Darwin abbia scosso quelle del XIX.

*La gente potrebbe perdere i propri diritti alla privacy.* Weizenbaum notò anche che la tecnologia del riconoscimento vocale avrebbe potuto portare a un grande incremento delle intercettazioni con una conseguente perdita dei diretti civili. Non

poteva prevedere un mondo in cui la minaccia del terrorismo avrebbe cambiato l'opinione della gente sul grado di sorveglianza che può considerare accettabile, ma riconobbe correttamente che l'IA ha il potenziale per consentire il controllo di massa. Le sue predizioni potrebbero essersi avvurate: il sistema americano Echelon “consiste in una rete di postazioni di ascolto, antenne e stazioni radar; il sistema è supportato da computer che sfruttano il riconoscimento e la traduzione del linguaggio naturale e sono in grado di cercare parole chiave per elaborare automaticamente grandi quantità di traffico telefonico, via email, fax e telex”.<sup>6</sup> Alcune persone hanno già accettato che la computerizzazione della società sia destinata a condurre a una perdita della privacy: il presidente della Sun Microsystems, Scott McNealy, ha dichiarato “La vostra privacy è zero in ogni caso. Fateci l'abitudine”. Altri non sono d'accordo: nel 1890 il giudice Louis Brandeis ha scritto che “Tra tutti diritti, la privacy è quello di più vasta portata... il diritto alla propria personalità”.

*L'uso di sistemi di IA renderebbe impossibile determinare le responsabilità legali.* Nell'atmosfera litigiosa che prevale negli Stati Uniti, la responsabilità legale è diventata una questione importante. Quando un medico si affida al giudizio di un sistema esperto per una diagnosi, di chi è la colpa se la diagnosi risulta sbagliata? Fortunatamente, in parte grazie alla crescente influenza dei metodi di teoria delle decisioni in medicina, è ora un fatto assodato che non si può tacciare di negligenza un medico che esegue procedure che hanno un'alta utilità *attesa*, anche se il risultato *effettivo* dovesse risultare catastrofico per il paziente. La questione quindi si dovrebbe esprimere in altri termini, “di chi è la colpa se la diagnosi non è ragionevole”? Fino a oggi, i tribunali hanno stabilito che i sistemi esperti di medicina hanno lo stesso ruolo dei libri di testo e dei manuali; i medici devono comprendere il ragionamento alla base di ogni decisione e usare il proprio giudizio per decidere se accettare le raccomandazioni del sistema. Progettando sistemi esperti medici come agenti, quindi, bisogna pensare che le azioni non modificheranno direttamente il paziente ma piuttosto influenzano il comportamento del medico. Se i sistemi esperti dovessero diventare regolarmente più accurati dei medici nelle diagnosi, i dottori potrebbero risultare perseguitibili legalmente qualora *non dovessero* seguire le loro raccomandazioni. Gawande (2002) esamina quest'ipotesi.

Problemi simili stanno cominciando a emergere riguardo all'uso degli agenti intelligenti su Internet. Alcuni agenti incorporano vincoli in modo da non potere, per esempio, danneggiare i file di altri utenti (Weld e Etzioni, 1994). Il problema risulta amplificato quando c'è di mezzo un passaggio di denaro: se un agente intelligente esegue delle transazioni monetarie “per conto di qualcuno”, quel qualcuno è perseguitabile in caso di frode o debito? Sarebbe possibile per un agente disporre di beni propri ed effettuare transazioni elettroniche per proprio conto? Fino a oggi questioni simili non sembrano essere state comprese molto bene. Per quanto ne

<sup>6</sup> Cfr. “Eavesdropping on Europe,” *Wired news*, 9/30/1998, e i rapporti della Unione Europea ivi citati.

sappiamo, a nessun programma è stato mai concesso lo stato legale ai fini di condurre transazioni finanziarie; nella situazione presente non sembra una scelta ragionevole. I programmi inoltre non sono considerati "guidatori" in senso legale sulle autostrade del mondo reale. Nella legge della California, almeno, non sembra esserci alcuna sanzione che impedisca a un veicolo automatizzato di superare i limiti di velocità, benché il progettista del meccanismo di controllo del veicolo sarebbe perseguitabile in caso di incidente. Come nel campo della tecnologia applicata alla riproduzione umana, la legge dev'essere ancora aggiornata per gestire gli ultimi sviluppi.

*Il successo dell'IA potrebbe significare la fine della razza umana.* Quasi tutte le tecnologie hanno la potenzialità di causare danno se capitano nelle mani sbagliate, ma nel caso dell'IA e della robotica, sorge il nuovo problema che le mani potrebbero appartenere alla tecnologia stessa. Le storie di fantascienza che riguardano robot o androidi impazziti sono innumerevoli: gli esempi più antichi includono *Frankenstein, o il moderno Prometeo* di Mary Shelley (1818)<sup>7</sup> e l'opera teatrale di Karel Čapek *R.U.R.* (1921), in cui i robot conquistano il mondo. Tra i film abbiamo *Terminator* (1984) che combina gli spunti dei robot-che-conquistano-il-mondo con il viaggio nel tempo, e *The Matrix* (1999) che combina i robot-che-conquistano-il-mondo con il cervello-nella-vasca.

Per lo più, sembra che i robot siano protagonisti di così tante storie di distruzione del mondo perché rappresentano ciò che è sconosciuto, come le streghe e i fantasmi dei racconti del passato. Ma rappresentano una minaccia più concreta di streghe e fantasmi? Se i robot sono progettati correttamente come agenti che adottano gli obiettivi del loro possessore, la risposta è probabilmente no: robot derivati da perfezionamenti incrementali dei progetti correnti serviranno l'uomo, non lo opprimeranno. Gli uomini usano la loro intelligenza in modo aggressivo a causa delle loro tendenze innate, dovute alla selezione naturale. Ma le macchine che costruiamo non hanno bisogno di essere aggressive, se non decidiamo di costruirle così. D'altra parte, è possibile che i computer arrivino al punto di "conquistare servendo", diventando indispensabili, proprio come le automobili hanno in un certo senso conquistato il mondo industrializzato. Uno scenario merita ulteriore considerazione: I. J. Good ha scritto (1965),

Definiamo ultraintelligente una macchina che può sorpassare di gran lunga le attività intellettuali di qualsiasi uomo, per quanto intelligente. Dato che la progettazione di macchine è una di queste attività intellettuali, una macchina ultraintelligente potrebbe progettare macchine ancora migliori; si avrebbe senza dubbio un'"esplosione di intelligenza", e quella dell'uomo rimarrebbe molto indietro. Così, la prima macchina ultraintelligente è l'*ultima* invenzione che l'uomo dovrà mai realizzare, a patto che sia abbastanza docile da spiegarci come tenerla sotto controllo.

<sup>7</sup> Da giovane, Charles Babbage fu influenzato dalla lettura di *Frankenstein*.

singolarità tecnologica

Quest’“esplosione di intelligenza” è stata anche chiamata **singolarità tecnologica** dal professore di matematica e autore di fantascienza Vernor Vinge, che ha scritto (1993), “Entro trent’anni, avremo i mezzi tecnologici per creare un’intelligenza sovrumana. Poco dopo, l’era dell’uomo giungerà al termine”. Good e Vinge (e molti altri) hanno correttamente notato che la curva del progresso tecnologico sta crescendo esponenzialmente (considerate la legge di Moore). Tuttavia, ci vuole un bel salto per estrapolare che la curva continuerà a crescere fino a una singolarità di incremento quasi infinito. Fino a oggi, tutte le altre tecnologie hanno seguito piuttosto una curva a S, in cui la crescita esponenziale a un certo punto si smorza.

Vinge è preoccupato e spaventato dall’imminente singolarità, ma altri informatici e studiosi del futuro la aspettano con ansia. Hans Moravec, in *Robot: Mere Machine to Transcendent Mind* predice che i robot uguaglieranno l’intelligenza umana in 50 anni e quindi la supereranno.

Potrebbero soppiantarci molto rapidamente. Al contrario di molti questa possibilità non mi allarma, dato che considero queste macchine future come la nostra progenie, “figli della mente” costruiti a nostra immagine e somiglianza, noi stessi in forma più potente. Come i figli biologici delle generazioni precedenti, incarneranno la migliore speranza dell’umanità per un futuro a lungo termine. È giusto da parte nostra dar loro tutti i vantaggi, e farci da parte quando non potremo più contribuire. (Moravec, 2000)

transumanesimo

Ray Kurzweil, in *The Age of Spiritual Machines* (2000), predice che entro l’anno 2099 ci sarà “una forte tendenza verso la fusione tra il pensiero umano e il mondo dell’intelligenza delle macchine create inizialmente dalla specie umana. Non ci sarà più una distinzione chiara tra esseri umani e computer”. È stato anche coniato un termine per indicare l’attivo movimento sociale che guarda con fiducia a questo futuro: **transumanesimo**. Fenomeni come questi rappresentano una sfida per la maggior parte dei teorici della morale, che pensano che la conservazione della vita e della specie umana sia una buona cosa.

Infine, consideriamo il punto di vista dei robot. Se dovessero diventare coscienti, trattarli come semplici “macchine” (ad esempio, farli a pezzi) potrebbe essere immorale. I robot stessi dovranno agire in modo retto: occorrerà programmarli con la teoria di ciò che è giusto e ciò che è sbagliato. A partire da Isaac Asimov (1942), gli scrittori di fantascienza si sono occupati della questione dei diritti e delle responsabilità dei robot. Il noto film *A.I.* (Spielberg, 2001) era basato su un racconto di Brian Aldiss su un robot intelligente che era stato programmato per credere di essere umano e non riusciva ad accettare l’abbandono da parte della padrona-madre. La storia (e il film) fanno riflettere sulla necessità di movimenti per i diritti civili dei robot.

## 26.4 Riepilogo

Questo capitolo ha trattato i seguenti argomenti.

- ◆ I filosofi usano il termine **IA debole** per indicare l'ipotesi che le macchine possano comportarsi in modo intelligente, e **IA forte** per indicare l'ipotesi che tali macchine avrebbero una vera e propria mente (e non già una simulazione).
- ◆ Alan Turing ha rifiutato la domanda “Le macchine possono pensare?” sostituendola con un test comportamentale e prevedendo molte delle possibili obiezioni all'ipotesi di macchine pensanti. Pochi ricercatori si occupano oggi del test di Turing, preferendo concentrarsi sulle prestazioni dei loro sistemi in attività pratiche piuttosto che sulla capacità di imitare gli esseri umani.
- ◆ Al giorno d'oggi si ritiene generalmente che gli stati mentali corrispondano a stati nel cervello.
- ◆ Le argomentazioni pro e contro l'IA forte non hanno portato ad alcuna conclusione. Ben pochi ricercatori ritengono che l'esito del dibattito possa avere qualche effetto significativo.
- ◆ La coscienza rimane un mistero.
- ◆ Abbiamo identificato sei potenziali minacce portate alla società dall'IA. Abbiamo concluso che alcune di esse sono molto improbabili o non differiscono dalle minacce rappresentate da altre tecnologie “non intelligenti”. Una in particolare, comunque, merita ulteriore considerazione: le macchine ultraintelligenti potrebbero portare a un futuro molto diverso, che forse non ci piacerebbe, ma a quel punto potremmo non avere scelta. Queste considerazioni hanno portato inevitabilmente alla conclusione che dobbiamo pensare attentamente, e al più presto, alle possibili conseguenze della ricerca nell'IA per il futuro della razza umana.

### Note storiche e bibliografiche

La natura della mente è un argomento classico della filosofia dai tempi antichi al presente. Nel *Fedone*, Platone considerò esplicitamente e rifiutò l'idea che la mente possa essere un “direttore d'orchestra” o uno schema di organizzazione delle parti del corpo, un punto di vista che si avvicina al funzionalismo della moderna filosofia della mente. Al contrario decise che la mente doveva essere costituita da un'anima immortale e immateriale, separabile dal corpo e differente da esso nella sostanza: questo è il punto di vista del dualismo. Aristotele distinse vari tipi di anime (in greco, *psiche*) negli esseri viventi, descrivendo almeno alcune di esse in modo funzionalista (v. Nussbaum (1978) per approfondire il funzionalismo aristotelico).

Cartesio è noto per la sua visione dualistica della mente umana, ma il suo pensiero influenzò soprattutto il meccanicismo e il materialismo. Considerò esplicitamente gli animali come automi, e arrivò ad anticipare il test di Turing scrivendo “non è concepibile [che una macchina] possa produrre combinazioni diverse di parole per rispondere in modo sensato a tutto ciò che si dirà in sua presenza, come possono fare anche gli uomini più ebbi” (Cartesio, 1637). L’accesa difesa degli “animali come automi” ebbe in realtà l’effetto di facilitare la concezione che gli esseri umani fossero automi anch’essi, sebbene Cartesio stesso non abbia mai compiuto questo passo. Il libro *L’Homme Machine*, o *L’Uomo Macchina* (La Mettrie, 1748) sosteneva esplicitamente che gli uomini fossero automi.

La filosofia analitica moderna tipicamente ha accettato il materialismo (spesso nella forma della **teoria dell’identità** (Place, 1956; Armstrong, 1968), che asserisce che gli stati mentali sono tutt’uno con gli stati del cervello), ma si è divisa molto più nettamente sul funzionalismo, che estende l’analogia meccanica alla mente umana, e sulla questione se le macchine possano letteralmente pensare. Tra le prime risposte filosofiche all’articolo di Turing (1950) “Computing Machinery and Intelligence” molti autori, tra cui Scriven (1953), tentarono di negare che *avesse senso* sostenere che le macchine potessero pensare, perché una simile affermazione avrebbe violato il senso stesso della parola. Scriven ritrattò le sue opinioni entro il 1963, come si può vedere da un’aggiunta alla ristampa del suo articolo originale (Anderson, 1964). Secondo l’informatico Edsger Dijkstra, “chiedersi se un computer possa pensare non è più interessante di chiedersi se un sottomarino possa nuotare”. Ford e Hayes (1995) sostengono che il test di Turing non porti alcun aiuto all’IA.

Il funzionalismo è la filosofia della mente che l’IA suggerisce in modo più naturale, e le critiche del funzionalismo spesso prendono la forma di critiche all’IA (come nel caso di Searle). Seguendo la classificazione usata da Block (1980), possiamo distinguere diversi tipi di funzionalismo. La **teoria della specifica funzionale** (Lewis, 1966, 1980) è una variante della teoria dell’identità degli stati che sceglie gli stati del cervello da identificare con i rispettivi stati mentali sulla base del loro ruolo funzionale. La **teoria dell’identità dello stato funzionale** (Putnam, 1960, 1967) è basata più strettamente sull’analogia con le macchine: gli stati mentali non sono identificati con stati *fisici* del cervello, bensì con stati computazionali astratti del cervello concepito espressamente come dispositivo di calcolo. Si suppone che questi stati astratti siano indipendenti dalla specifica composizione fisica del cervello, il che ha portato ad accuse che la teoria dell’identità dello stato funzionale sia una forma di dualismo! La teoria dell’identità e le altre forme di funzionalismo sono state attaccate da autori che sostenevano che non tenevano conto dei *qualia*, ovvero di “cosa sembrano” gli stati mentali (Nagel, 1974). Searle si è invece concentrato sulla presunta incapacità del funzionalismo di spiegare l’intenzionalità (Searle, 1980, 1984, 1992). Churchland e Churchland (1982) ribattono a entrambe queste critiche.

Il **materialismo eliminativo** (Rorty, 1965; Churchland, 1979) è diverso da tutte le altre teorie nella filosofia della mente, in quanto non cerca di giustificare la

“psicologia popolare” o quello che riguardo alla mente ci suggerisce il buon senso, ma rifiuta tutto ciò come falso e cerca di sostituirlo con una teoria puramente scientifica. In via di principio, questa teoria potrebbe essere fornita dall’IA classica, ma in pratica i materialisti eliminativi tendono a far riferimento alle neuroscienze e alla ricerca sulle reti neurali (Churchland, 1986), sulla base del fatto che l’IA classica, e in particolare una “rappresentazione della conoscenza” come quella descritta nel Capitolo 10, tende a dare per scontata la validità della psicologia popolare. Benché il punto di vista della “posizione intenzionale” (Dennett, 1971) possa essere interpretato come funzionalista, dovrebbe probabilmente essere considerato una forma di materialismo eliminativo, dal momento che la “posizione intenzionale” di un agente non riflette alcuna sua caratteristica oggettiva nei riguardi dell’oggetto verso cui assume la posizione. Si dovrebbe anche notare che è possibile essere materialisti eliminativi riguardo certi aspetti della mente e analizzarne altri in qualche altro modo. Ad esempio, Dennett (1978) è molto più eliminativista riguardo ai qualia che riguardo all’intenzionalità.

Abbiamo fornito nel capitolo alcune fonti per le critiche più importanti dell’IA debole. Sebbene snobbare gli approcci simbolici sia diventato di moda nell’era post-reti neurali, non tutti i filosofi hanno un atteggiamento critico verso GOFAI. Alcuni, in effetti, ne sono ferventi difensori e addirittura praticanti. Zenon Wylshyn (1984) ha sostenuto che per comprendere la cognizione è meglio adottare un modello computazionale, non solo in via di principio ma anche come modo di condurre la ricerca, e ha specificatamente riconosciuto le critiche di Dreyfus al modello computazionale della cognizione umana (Pylyshyn, 1974). Gilbert Harman (1983), analizzando la revisione delle credenze, fa riferimento alla ricerca nell’IA e ai sistemi di mantenimento della verità. Michael Bratman ha applicato il suo modello “credenza-desiderio-intenzione” della psicologia umana (Bratman, 1987) alla ricerca sulla pianificazione (Bratman, 1992). Al limite estremo dell’IA forte, Aaron Sloman (1978, p. xiii) ha persino bollato come “razzista” l’opinione di Joseph Weizenbaum (Weizenbaum, 1976) secondo cui ipotetiche macchine intelligenti non dovrebbero essere considerate persone.

La letteratura filosofica riguardante le menti, i cervelli e gli argomenti correlati è molto ampia e talvolta difficile da leggere senza una formazione specifica nella terminologia e i metodi di argomentazione impiegati. L’*Encyclopedia of Philosophy* (Edwards, 1967) è un riferimento utile ed estremamente autorevole. Il *Cambridge Dictionary of Philosophy* (Audi, 1999) è più breve ed accessibile, ma le voci principali (come “filosofia della mente”) occupano comunque 10 o più pagine. La *MIT Encyclopedia of Cognitive Science* (Wilson and Keil, 1999) tratta la filosofia della mente, ma anche la sua biologia e la psicologia. Tra le collezioni di articoli generali su vari aspetti della filosofia della mente, tra cui il funzionalismo e altri punti di vista importanti per l’IA, citiamo *Materialism and the Mind-Body Problem* (Rosenthal, 1971) e *Readings in the Philosophy of Psychology*, volume 1 (Block, 1980). Biro e Shahan (1982) presentano una raccolta dedicata ai pro e contro del funzionalismo. Antologie di articoli che si occupano più specificata-

mente della relazione tra filosofia e IA includono *Minds and Machines* (Anderson, 1964), *Philosophical Perspectives in Artificial Intelligence* (Ringle, 1979), *Mind Design* (Haugeland, 1981) e *The Philosophy of Artificial Intelligence* (Boden, 1990). Ci sono diverse introduzioni generali al “problema dell’IA” dal punto di vista filosofico (Boden, 1977, 1990; Haugeland, 1985; Copeland, 1993). *The Behavioral and Brain Sciences*, abbreviato in *BBS*, è un’importante rivista dedicata al dibattito filosofico-scientifico sull’IA e le neuroscienze. Argomenti come l’etica e la responsabilità nell’AI sono trattati in riviste come *AI and Society*, *Law, Computers and Artificial Intelligence* e *Artificial Intelligence and Law*.

## Esercizi

---

- 26.1 Esaminate punto per punto la lista di Turing delle presunte “incapacità” delle macchine, identificando quali sono state superate, quali potrebbero in via di principio essere superate da un programma, e quali sono ancora problematiche perché richiedono stati mentali coscienti.
- 26.2 Pensate che una refutazione dell’argomentazione della stanza cinese dimostri necessariamente che computer adeguatamente programmati possano avere stati mentali? Pensate che accettare l’argomentazione significhi necessariamente che i computer non possano avere stati mentali?
- 26.3 Nell’argomentazione della protesi cerebrale è importante riportare il cervello del soggetto nella condizione normale, in modo che per quanto riguarda il suo comportamento esterno sia come se l’operazione non avesse mai avuto luogo. Uno scettico potrebbe ragionevolmente obiettare che questo richiederebbe l’aggiornamento delle proprietà neurofisiologiche dei neuroni relative all’esperienza cosciente, distinte da quelle responsabili del loro comportamento funzionale?
- 26.4 Trovate e analizzate nei media di larga diffusione un caso in cui vengono proposte una o più argomentazioni per cui l’IA sarebbe impossibile.
- 26.5 Cercate di scrivere delle definizioni dei termini “intelligenza”, “pensiero” e “saggezza”. Suggerite alcune possibili obiezioni alle vostre definizioni.
- 26.6 Analizzate le possibili minacce portate dalla tecnologia dell’IA alla società. Quali sono le più gravi e come potrebbero essere combattute? Confrontate con i potenziali benefici.
- 26.7 Confrontate le potenziali minacce dell’IA con quelle portate da altre tecnologie informatiche e con le tecnologie bio-, nano- e nucleari.
- 26.8 Alcuni critici obiettano che l’IA è impossibile, mentre altri sostengono che è *fin troppo* possibile, e che le macchine ultraintelligenti rappresentano un pericolo. Quale delle due obiezioni vi sembra più probabile? Sarebbe una contraddizione sostenerle entrambe?

## Capitolo 27

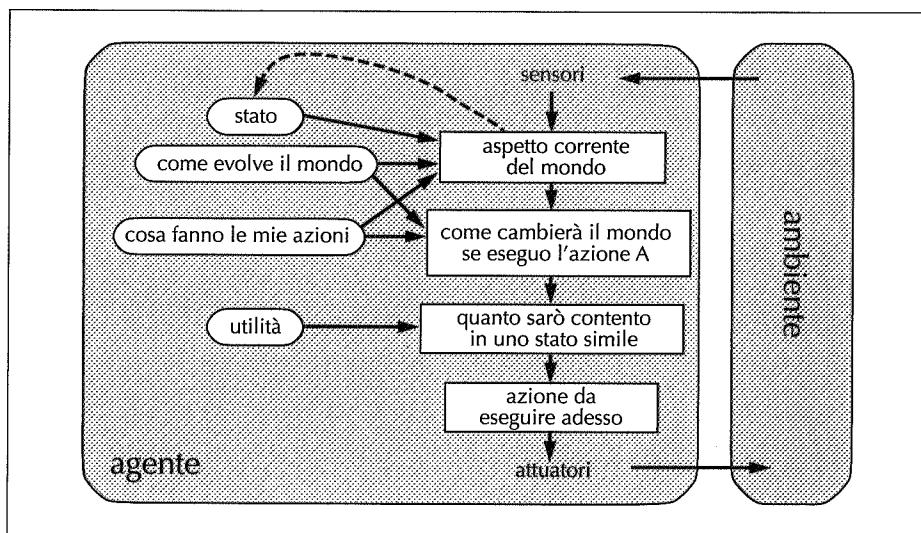
# IA: presente e futuro

*In cui facciamo il punto di dove siamo e dove stiamo andando, essendo questa una buona cosa da fare prima di procedere.*

Nella Parte I, abbiamo proposto una visione unificata dell'IA come progettazione di agenti razionali. Abbiamo mostrato che il problema della progettazione dipende dalle percezioni e dalle azioni disponibili, dagli obiettivi che il comportamento dell'agente deve soddisfare e dalla natura dell'ambiente. È possibile ricorrere a una varietà di soluzioni, da semplici agenti reattivi ad agenti completamente deliberativi basati sulla conoscenza. Inoltre, i componenti di tali progetti possono essere istanziati in molti modi diversi: logici, probabilistici o "neurali". I capitoli del libro hanno presentato i principî secondo cui operano tali componenti.

Per tutti gli stili di progettazione e i componenti si sono avuti progressi notevolissimi sia nella comprensione scientifica che nelle capacità tecnologiche. In questo capitolo cercheremo di astrarre dai dettagli per porci la domanda *"tutti questi progressi porteranno alla creazione di un agente intelligente di uso generale che può comportarsi bene in una grande varietà di ambienti?"*. Il Paragrafo 27.1 esamina i componenti di un agente intelligente per stabilire ciò che sappiamo e ciò che ancora manca. Il Paragrafo 27.2 fa lo stesso nei confronti dell'architettura generale di un agente. Il Paragrafo 27.3 si chiede se la "progettazione di agenti razionali" è il giusto obiettivo a cui puntare (la risposta è "non proprio, ma per ora va bene"). Infine, il Paragrafo 27.4 prende in considerazione le possibili conseguenze dei nostri sforzi.

**Figura 27.1**  
Un agente dotato di modello del mondo e basato sull'utilità, già presentato nella Figura 2.14.



## 27.1 Componenti per agenti

Il Capitolo 2 ha presentato diversi metodi di progettazione per gli agenti e i loro componenti. Per focalizzare la discussione, faremo riferimento all'agente basato sull'utilità, riportato nella Figura 27.1. Tra i progetti di agente, questo è quello più generale; considereremo anche le sue estensioni basate sull'apprendimento, rappresentate nella Figura 2.15.

*Interazione con l'ambiente attraverso sensori e attuatori.* Per gran parte della storia dell'IA, questo è stato un vistoso punto debole. Con poche eccezioni degne di nota, i sistemi di intelligenza artificiale sono sempre stati costruiti in modo che gli esseri umani fornissero gli input e interpretassero gli output, mentre i sistemi robotici si sono focalizzati su attività di basso livello in cui il ragionamento e la pianificazione sono per lo più assenti. In parte questo è stato dovuto alle grandi spese e agli sforzi ingegneristici necessari per costruire robot funzionanti. La situazione è cambiata rapidamente in anni recenti grazie alla disponibilità di robot programmabili prefabbricati. A loro volta questi ultimi si sono potuti avvalere di telecamere CCD ad alta risoluzione piccole ed economiche, e di motori compatti e affidabili. La tecnologia dei MEMS (sistemi micro-elettromeccanici) ha fornito accelerometri e giroscopi miniaturizzati e sta producendo attuatori in grado, ad esempio, di far volare un insetto artificiale. È anche possibile combinare milioni di attuatori MEMS per produrre attuatori macroscopici molto potenti. Per quanto riguarda gli ambienti fisici, quindi, i sistemi di IA non hanno più scuse plausibili. Inoltre, ora è disponibile un ambiente completamente nuovo: Internet.

*Tener traccia dello stato del mondo.* Questa è una delle capacità principali di un agente intelligente, e richiede sia la percezione che la capacità di aggiornare rappresentazioni interne. Il Capitolo 7 presenta metodi per tener traccia di mondi descritti mediante la logica proposizionale; il Capitolo 10 li estende alla logica del primo ordine; il Capitolo 15, nel 2° volume, descrive algoritmi di **filtraggio** da utilizzare in ambienti incerti. Strumenti di filtraggio sono necessari quando è coinvolta una percezione reale (e quindi imperfetta). Gli algoritmi correnti di filtraggio e percezione possono essere combinati e funzionano abbastanza bene nel riportare predicati di basso livello come “la tazza è sul tavolo”, ma serve ancora molto lavoro prima di poter riportare che “il Dr. Russell si sta facendo una tazza di tè con il Dr. Norvig”. Un altro problema è rappresentato dal fatto che, benché gli algoritmi di filtraggio approssimato possano gestire ambienti piuttosto grandi, essenzialmente sono ancora *proposizionali*: come la logica proposizionale, non rappresentano esplicitamente oggetti e relazioni. Il Capitolo 14, nel 2° volume, spiegherà come probabilità e logica del primo ordine possono essere combinate per risolvere questo problema; ci aspettiamo che l’applicazione di queste idee alla gestione di ambienti complessi apporterà grandi benefici. Incidentalmente, non appena cominciamo a parlare di *oggetti* in un ambiente incerto, ci scontriamo con l’**incertezza dell’identità**: non sappiamo identificare gli oggetti con sicurezza. Questo problema è stato in gran parte ignorato dall’IA basata sulla logica, in cui tipicamente le percezioni contengono simboli di costante che identificano gli oggetti.

*Progettare, valutare e scegliere le azioni future.* In questo caso i requisiti, per quanto riguarda la rappresentazione della conoscenza necessaria, sono gli stessi richiesti per tener traccia del mondo; la difficoltà principale sta nel gestire corsi d’azione – come sostenere una conversazione, o farsi una tazza di tè – che per un agente reale possono consistere in migliaia o milioni di passi primitivi. L’unica soluzione, che è poi quella adottata dagli esseri umani, è imporre una **struttura gerarchica** al comportamento. Alcuni degli algoritmi di pianificazione del Capitolo 12 usano rappresentazioni gerarchiche, unite a quelle del primo ordine, per gestire problemi di questa scala; d’altra parte, gli algoritmi esaminati nel Capitolo 17 per le decisioni in condizioni di incertezza sfruttano sostanzialmente le stesse idee della ricerca nello spazio degli stati del Capitolo 3. Rimane chiaramente molto lavoro da fare in questo campo, e forse sarà utile seguire la linea tracciata da sviluppi recenti come l’**apprendimento per rinforzo gerarchico**.

*L’utilità come espressione delle preferenze.* In via di principio, basare le decisioni razionali sulla massimizzazione dell’utilità è un principio assolutamente generale e permette di evitare molti problemi degli approcci basati su obiettivi puri, come i conflitti tra gli obiettivi stessi e l’incertezza del loro raggiungimento. A tutt’oggi, comunque, è stato fatto ben poco lavoro sulla costruzione di funzioni di utilità *realistiche*: immaginate ad esempio la complessa rete di preferenze reciprocamente dipendenti di cui deve tener conto un agente che svolge il ruolo di assistente di ufficio di un utente umano. Scomporre le preferenze su stati complessi,

incertezza dell’identità

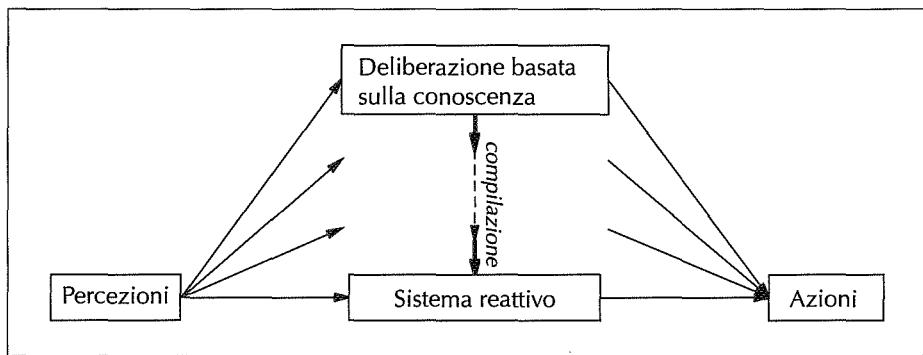
allo stesso modo in cui le reti di Bayes scompongono le credenze, si è rivelato un compito molto difficile. Una ragione potrebbe stare nel fatto che le preferenze sugli stati in realtà sono *compilate* partendo da preferenze sulle storie degli stati, descritte da **funzioni di ricompensa** (v. Capitolo 17).

Anche se la funzione di ricompensa è semplice, la corrispondente funzione di utilità potrebbe essere molto complessa. Questo suggerisce che dovremmo considerare molto seriamente l'ingegneria della conoscenza applicata alle funzioni di ricompensa come un mezzo per rendere noto ai nostri agenti quello che desideriamo che facciano.

**Apprendimento.** Nel 2° volume descriveremo come si può formulare l'apprendimento induttivo (supervisionato, non supervisionato o basato sul rinforzo) delle funzioni che costituiscono i vari componenti di un agente. Sono state sviluppate tecniche logiche e statistiche molto potenti che possono gestire problemi di grandi dimensioni, spesso raggiungendo o superando le capacità umane nell'identificazione di schemi predittivi definiti su uno specifico vocabolario. D'altra parte, l'apprendimento automatico ha fatto pochi progressi per quanto riguarda l'importante problema di costruire nuove rappresentazioni a livelli di astrazione superiori al vocabolario di input. Ad esempio, come può un robot autonomo generare predicati utili come *Ufficio* e *Caffè* se non gli vengono forniti da esseri umani? Considerazioni analoghe si applicano allo stesso processo di apprendimento: *BersiUnaTazzaDiTè* è un'importante azione di alto livello, ma come si fa a inserirla in una libreria che inizialmente contiene azioni molto più semplici, come *SollevaBraccio* e *Inghiotti*? Finché non comprendiamo questi problemi, dovremo sempre affrontare il compito improbo di costruire a mano grandi basi di conoscenza comune.

## 27.2 Architetture di agenti

Alla domanda, piuttosto naturale, “quali architetture di agenti dovrei usare tra quelle descritte nel Capitolo 2?” la risposta corretta è “tutte!”. Abbiamo visto che le risposte reattive sono necessarie quando è indispensabile reagire in tempo reale, mentre la deliberazione basata sulla conoscenza consente agli agenti di pianificare il futuro. Un agente completo dev'essere in grado di sfruttare entrambi gli approcci, avvalendosi di un'**architettura ibrida**. Una caratteristica importante delle architetture ibride sta nel fatto che i confini tra i diversi componenti decisionali non sono prefissati. La **compilazione**, ad esempio, converte continuamente informazione dichiarativa a livello deliberativo in rappresentazioni più efficienti, fino a raggiungere il livello dei riflessi (v. Figura 27.2). Questo è anche lo scopo dell'apprendimento basato sulle spiegazioni, che discutiamo nel Capitolo 19 del 2° volume. Architetture di agenti come SOAR (Laird et al., 1987) e THEO (Mitchell, 1990) hanno esattamente questa struttura: ogni volta che risolvono un problema attra-



**Figura 27.2**  
La compilazione converte decisioni raggiunte in modo deliberativo in meccanismi reattivi più efficienti.

verso la deliberazione esplicita, salvano a parte una versione generalizzata della soluzione affinché il componente reattivo possa utilizzarla. Un problema meno studiato è il processo *inverso*: quando l'ambiente cambia, le reazioni apprese potrebbero non essere più appropriate; l'agente deve tornare al livello deliberativo per produrre comportamenti nuovi.

Gli agenti necessitano anche di un meccanismo per controllare le loro deliberazioni: quando è necessario agire devono essere capaci di interrompere l'elaborazione, e sfruttare il tempo effettivamente disponibile per eseguire i calcoli più utili. Ad esempio, un agente guidatore di taxi che dovesse rilevare un incidente davanti a sé dovrà decidere in una frazione di secondo se frenare o sterzare. In quella frazione di secondo dovrà considerare gli aspetti più importanti della situazione, come il traffico nelle corsie immediatamente a destra e sinistra o la presenza di un camion subito dietro a sé, piuttosto di perdere tempo a preoccuparsi dell'usura delle gomme o della destinazione del passeggero. Tutte queste considerazioni sono solitamente riunite sotto la denominazione di **IA in tempo reale**. Man mano che l'IA si espanderà in domini sempre più complessi, tutti i problemi diventeranno in tempo reale, perché l'agente non avrà mai a disposizione il tempo necessario per risolvere esattamente il problema decisionale.

È chiaro che c'è un grande bisogno di metodi che operino in situazioni decisionali più generali. Negli anni recenti sono emerse due tecniche promettenti: la prima consiste nei cosiddetti **algoritmi anytime** (Dean e Boddy, 1988; Horvitz, 1987). Un algoritmo anytime è contraddistinto dal fatto che la qualità del suo output migliora gradualmente nel tempo, in modo da poter restituire una decisione ragionevole in qualsiasi momento venga interrotto. Questi algoritmi sono controllati da una procedura di decisione di metalivello che valuta se vale la pena svolgere ulteriori calcoli. La ricerca ad approfondimento iterativo applicata ai giochi rappresenta un semplice esempio di algoritmo anytime. È anche possibile costruire sistemi più complessi, formati da diversi algoritmi simili che lavorano insieme

IA in tempo reale

algoritmi anytime

metaragionamento  
basato sulla teoria  
delle decisioni

architettura riflessiva

(Zilberstein e Russell, 1996). La seconda tecnica è il **metaragionamento basato sulla teoria delle decisioni** (Horvitz, 1989; Russell e Wefald, 1991; Horvitz e Breese, 1996). Questo metodo applica la teoria del valore dell'informazione (v. Capitolo 16) alla scelta dei calcoli da eseguire. Il valore di un calcolo dipende sia dal suo costo (in termini di ritardo nell'azione) che dai suoi benefici (in termini di miglioramento della qualità della decisione). Le tecniche di metaragionamento possono essere usate per progettare algoritmi di ricerca migliori e per garantire che gli algoritmi abbiano la proprietà anytime. Il metaragionamento è costoso, naturalmente, ma si possono sfruttare tecniche di compilazione per far sì che l'overhead sia piccolo rispetto al costo delle computazioni controllate.

Il metaragionamento è solo un aspetto di una generale **architettura riflessiva** che permette di deliberare sulle entità e sulle azioni computazionali che accadono all'interno dell'architettura stessa. Per costruire un fondamento teorico per le architetture riflessive si potrebbe definire uno spazio degli stati congiunto composto dagli stati dell'ambiente e da quelli computazionali dell'agente. Si potrebbero poi sviluppare algoritmi di apprendimento e di supporto alle decisioni progettati per operare su questo spazio congiunto e quindi implementare e migliorare le attività computazionali dell'agente. Possiamo pensare che a un certo punto algoritmi specifici come la ricerca alfa-beta e la concatenazione all'indietro scompaiano dai sistemi di IA, per essere rimpiazzati da metodi generali che guidano l'agente verso la generazione efficiente di decisioni di alta qualità.

## 27.3 Stiamo andando nella giusta direzione?

Nel paragrafo precedente abbiamo elencato molti progressi già raggiunti e molte opportunità per ulteriori avanzamenti. Ma dove sta portando tutto questo? Dreyfus (1992) paragona tutto ciò al tentativo di raggiungere la luna arrampicandosi su un albero: si può riscontrare un progresso costante per tutta la scalata, fino alla cima. In questo paragrafo consideriamo se il cammino corrente dell'IA sia più simile a un'arrampicata o a un viaggio spaziale. Nel Capitolo 1, abbiamo detto che il nostro scopo era costruire agenti che *si comportassero razionalmente*. Tuttavia, abbiamo anche detto che

...raggiungere la razionalità perfetta – fare sempre la cosa giusta – non è fattibile all'interno di sistemi complicati: i requisiti computazionali sono semplicemente troppo alti. Nella maggior parte del libro, comunque, partiremo dall'ipotesi che la razionalità perfetta sia un buon punto d'inizio per l'analisi.

È giunto il momento di considerare ancora l'obiettivo dell'IA. Noi vogliamo costruire agenti, ma che specifica abbiamo in mente? Ci sono quattro possibilità.

**Razionalità perfetta.** Un agente razionale agisce in ogni istante in modo tale da massimizzare l'utilità attesa, data l'informazione acquisita dall'ambiente. Abbiamo visto che nella maggior parte degli ambienti i calcoli necessari per raggiungere una razionalità perfetta richiedono troppo tempo, per cui non sembra un obiettivo realistico.

razionalità perfetta

**Razionalità calcolativa.** Questo è il concetto di razionalità che abbiamo implicitamente utilizzato nella progettazione degli agenti logici e di quelli basati sulla teoria delle decisioni. Un agente razionale in senso calcolativo *a un certo punto* restituisce quella che *sarebbe stata* la scelta razionale nell'istante in cui ha cominciato la deliberazione. Si tratta di una caratteristica interessante dei sistemi, ma nella maggior parte degli ambienti la risposta giusta nel momento sbagliato non ha alcun valore. Nella pratica, i progettisti di sistemi di IA sono obbligati a gestire il compromesso tra la qualità delle decisioni e il tempo di risposta; sfortunatamente le basi teoriche della razionalità calcolativa non forniscono un metodo consolidato con cui valutare tale compromesso.

razionalità calcolativa

**Razionalità limitata.** Herbert Simon (1957) ha rifiutato il concetto di razionalità perfetta (o anche solo approssimativamente perfetta) sostituendolo con quello di razionalità limitata, una teoria descrittiva del processo decisionale degli agenti reali. Egli scrisse che

razionalità limitata

La capacità della mente umana di formulare e risolvere problemi complessi è molto piccola rispetto alle dimensioni dei problemi che devono essere risolti per ottenere un comportamento obiettivamente razionale nel mondo reale, o anche una sua ragionevole approssimazione.

Simon suggerì che la razionalità limitata funzioni principalmente attraverso il **soddisfacimento**, deliberando solamente per il tempo necessario a trovare una soluzione "sufficientemente buona". Simon ha vinto il premio Nobel per l'economia grazie a questo lavoro, su cui ha scritto in modo approfondito (Simon, 1982). In molti casi il soddisfacimento sembra essere un utile modello del comportamento umano; tuttavia non è una specifica formale di agenti intelligenti, perché la teoria non fornisce la definizione di "sufficientemente buona". Inoltre sembra essere solo uno di molti possibili metodi per la gestione di risorse limitate.

ottimalità limitata

**Ottimalità limitata** (BO, da *Bounded Optimality*). Un agente a ottimalità limitata si comporta nel miglior modo possibile *date le sue risorse computazionali*. Questo significa che l'utilità attesa del programma agente è almeno altrettanto alta di quella di qualsiasi altro agente in esecuzione sulla stessa macchina.

Di queste quattro possibilità, l'ottimalità limitata sembra offrire la migliore speranza di un forte fondamento teorico per l'IA. Il suo vantaggio è che è sempre possibile raggiungerla: c'è sempre almeno un programma migliore, cosa che non si può dire del caso della razionalità perfetta. Gli agenti a ottimalità limitata sono effettivamente utili nel mondo reale, mentre quelli a razionalità calcolativa normalmente non lo sono: quelli basati sul soddisfacimento potrebbero esserlo o no, a seconda dei casi.

L'approccio tradizionale dell'IA è stato di iniziare con la razionalità calcolativa e poi valutare i compromessi necessari a soddisfare i vincoli sulle risorse. Se i problemi rappresentati dai vincoli sono minimi, il progetto finale dovrebbe assomigliare a quello a ottimalità limitata. Man mano che i vincoli diventano più stringenti, comunque (ad esempio al crescere della complessità dell'ambiente) i due progetti dovrebbero differenziarsi sempre più. La teoria dell'ottimalità limitata consente di gestire questi vincoli basandosi su principî solidi.

A tutt'oggi, non molto è noto riguardo all'ottimalità limitata. È possibile costruire programmi per macchine molto semplici e tipi ristretti di ambienti (Etzioni, 1989; Russell et al, 1993), ma non abbiamo ancora idea di come si sviluppano programmi BO in ambienti complessi su computer di uso generale. Qualora si dovesse sviluppare una teoria costruttiva dell'ottimalità vincolata, ci auguriamo che il progetto dei programmi risultanti non dipenda troppo fortemente dai dettagli dei computer utilizzati. La ricerca risulterebbe molto difficile se l'aggiunta di qualche kilobyte di memoria a una macchina che ne possiede un gigabyte rappresentasse una differenza significativa per un programma BO. Un modo di assicurarsi che questo non possa accadere è rilassare leggermente i criteri dell'ottimalità limitata. Per analogia con la nozione di complessità asintotica (v. Appendice A), possiamo definire l'**ottimalità limitata asintotica (ABO)** come segue (Russell e Subramanian, 1995). Supponiamo che un programma  $P$  sia limitatamente ottimo per una macchina  $M$  in una classe di ambienti  $E$ , ove la complessità degli ambienti in  $E$  è illimitata. Allora il programma  $P'$  è ABO per  $M$  in  $E$  se ha prestazioni migliori di  $P$  quando viene mandato in esecuzione su una macchina  $kM$  che è  $k$  volte più veloce (o grande) di  $M$ . A meno di non ritrovarci con  $k$  enormi, ci accontenteremmo di un programma che fosse ABO in un ambiente non banale su un'architettura non banale. Non avrebbe molto senso spendere molti sforzi per trovare programmi BO anziché ABO, dal momento che la dimensione e la velocità delle macchine disponibili tende a crescere in ogni caso di un fattore costante in un lasso di tempo prefissato.

Possiamo azzardare l'ipotesi che i programmi BO o ABO su computer potenti non avranno necessariamente una struttura semplice ed elegante. Abbiamo già visto che un'intelligenza di uso generale richiede una capacità sia reattiva che deliberativa, una varietà di forme diverse di conoscenza e di processi decisionali, meccanismi di apprendimento e di compilazione per tutte queste forme, metodi per il controllo del ragionamento e un grande deposito di conoscenza specifica del dominio. Un agente a ottimalità limitata deve adattarsi all'ambiente in cui si trova, per cui alla fine la sua organizzazione interna rifletterà ottimizzazioni specifiche di quel particolare ambiente. Questo è assolutamente prevedibile, e ricorda il modo in cui le macchine da corsa con motori di potenza limitata si sono evolute in progetti estremamente complessi. Sospettiamo che una scienza dell'intelligenza artificiale basata sull'ottimalità limitata richiederà una gran quantità di studio dei processi che permettono a un programma agente di convergere su tale ottimalità, concentrandosi meno sui dettagli dei disordinati programmi che ne risulterebbero.

In definitiva, il concetto di ottimalità limitata si pone come un obiettivo formale sia ben definito che raggiungibile: il suo scopo è consentire la specifica di *programmi* anziché *azioni* ottime. Le azioni, dopotutto, sono generate dai programmi, ed è su questi ultimi che i progettisti possono esercitare un controllo.

## 27.4 E se l'IA avesse successo?

In *Small World* di David Lodge (1984), un romanzo ambientato nel mondo accademico della critica letteraria, il protagonista causa grande costernazione in un gruppo di eminenti teorici della letteratura in disaccordo chiedendo loro: “*e se aveste ragione?*”. Nessuno dei luminari apparentemente aveva considerato questa possibilità, forse perché il dibattito su teorie non falsificabili costituisce un fine in sé. Talvolta si può causare un imbarazzo simile domandando ai ricercatori nel campo dell'IA, “*e se dovreste avere successo?*”. L'IA è affascinante, e computer intelligenti sono chiaramente più utili di computer non intelligenti, per cui che ragione c'è di preoccuparsi?

Come abbiamo detto nel Paragrafo 26.3, ci sono questioni etiche da considerare. I computer intelligenti sono più potenti, ma quella potenza sarà usata a fin di bene? Coloro che si sforzano di far progredire l'IA hanno al responsabilità di assicurarsi che il loro lavoro abbia un effetto positivo. La vastità di tale effetto dipenderà dal grado di successo dell'IA. Anche successi modesti nel campo dell'intelligenza artificiale hanno modificato il modo in cui si insegna l'informatica (Stein, 2002) e si sviluppa software. L'IA ha reso possibile nuove applicazioni come i sistemi di riconoscimento vocale, di controllo di inventario e di sorveglianza, i robot e i motori di ricerca.

Possiamo presumere che successi di livello medio influenzerebbero la vita quotidiana di tutte le persone. Fino ad oggi sistemi di comunicazione computerizzata come la telefonia cellulare e Internet hanno avuto questo tipo di impatto pervasivo sulla società, ma non l'IA. Possiamo immaginare che aiutanti personali davvero utili in ufficio e in casa avrebbero un effetto molto positivo sulla vita della gente, anche se potrebbero causare qualche sfasamento economico a breve termine. Una capacità tecnologica di questo livello potrebbe anche essere applicata alla creazione di armi autonome, uno sviluppo che molti considerano indesiderabile.

Infine, sembra probabile che un successo dell'IA su grande scala – la creazione di un'intelligenza pari o superiore a quella umana – cambierebbe la vita della maggior parte dell'umanità. La stessa natura del nostro lavoro e del nostro divertimento sarebbero alterate, così come la nostra idea di intelligenza, di coscienza e del destino futuro dell'umanità. A questo livello, i sistemi di IA potrebbero rappresentare una minaccia diretta all'autonomia, alla libertà e persino alla sopravvivenza dell'uomo. Per queste ragioni, non possiamo separare la ricerca nell'intelligenza artificiale dalle sue implicazioni etiche.

Che direzione prenderà il futuro? Gli autori di fantascienza sembrano favorire le visioni distopiche rispetto a quelle utopistiche, probabilmente perché è più facile ambientarvi trame interessanti. Fino a oggi, comunque, l'IA sembra seguire il solco tracciato da altre tecnologie rivoluzionarie come la stampa, le tubature in piombo, i viaggi aerei e la telefonia, le cui ripercussioni negative sono ampiamente controbilanciate dai vantaggi.

In conclusione, l'IA ha fatto grandi progressi nella sua breve storia, ma vale ancora oggi l'ultima frase dell'articolo di Alan Turing *Computing Machinery and Intelligence*:

Possiamo vedere solo una breve distanza davanti a noi, ma vediamo che rimane ancora molto da fare.

## Appendice A

# Fondamenti matematici

## A.1 Analisi di complessità e notazione O()

Gli informatici si trovano spesso nelle condizioni di confrontare diversi algoritmi per vedere quanto velocemente si possono eseguire o quanta memoria richiedono. Per far questo ci sono due approcci: il primo consiste nel ricorrere ai **benchmark**, eseguendo cioè gli algoritmi su un computer e misurandone la velocità in secondi e l'occupazione di memoria in byte. Alla fine è questo ciò che conta, ma un benchmark può risultare insoddisfacente a causa della sua specificità: misura solo la prestazione di un particolare programma scritto in un particolare linguaggio e tradotto con un particolare compilatore, in esecuzione su un particolare computer con particolari dati in input. Dal singolo risultato fornito dal benchmark potrebbe essere difficile prevedere come si comporterebbe l'algoritmo con differenti compilatori, computer o insiemi di dati.

### Analisi asintotica

Il secondo approccio si appoggia alla matematica per eseguire un'**analisi degli algoritmi** indipendente dalla particolare implementazione e dall'input corrente. Presenteremo quest'approccio mediante il seguente esempio, un programma che calcola la somma di una sequenza di numeri:

---

```
function SOMMA(sequenza) returns un numero
 somma ← 0
 for i ← 1 to LUNGHEZZA(sequenza)
 somma ← somma + sequenza[i]
 return somma
```

---

benchmark

analisi degli algoritmi

Il primo passo dell’analisi è astrarre l’input, ovvero trovare qualche parametro o insieme di parametri che possano caratterizzarne le dimensioni. In questo caso l’input può essere caratterizzato dalla lunghezza della sequenza, che chiameremo  $n$ . Il secondo passo è astrarre l’implementazione, per trovare una misura che rispecchia il tempo d’esecuzione dell’algoritmo senza essere legata a un particolare compilatore o computer. Per il programma SOMMA potremmo semplicemente considerare il numero di righe di codice eseguite, o potremmo essere più dettagliati e contare le addizioni, gli assegnamenti, gli accessi agli array e i test. Entrambi i metodi ci offrono una caratterizzazione del numero totale di passi effettuati dall’algoritmo in funzione delle dimensioni dell’input, che chiameremo  $T(n)$ . Se contiamo le righe di codice, nel nostro esempio abbiamo  $T(n) = 2n + 2$ .

Se tutti i programmi fossero semplici come SOMMA, l’analisi degli algoritmi sarebbe banale. Due problemi, comunque, lo rendono più complicato: prima di tutto, è raro trovare un parametro come  $n$  che caratterizza completamente il numero di passi eseguiti da un algoritmo. Quello che di solito possiamo fare è calcolare il caso pessimo  $T_{\text{worst}}(n)$  o quello medio  $T_{\text{avg}}(n)$ . Per calcolare la media l’analista dovrà formulare un’ipotesi sulla distribuzione dell’input.

Il secondo problema è che gli algoritmi tendono a rendere difficile un’analisi esatta: in tal caso è necessario accontentarsi di un’approssimazione. Diciamo che l’algoritmo SOMMA è  $O(n)$  per indicare che la sua misura è al più un numero costante di volte  $n$ , con possibili eccezioni quando il valore di  $n$  è piccolo. Più formalmente,

$$T(n) \text{ è } O(f(n)) \text{ se } T(n) \leq kf(n) \text{ per qualche } k, \text{ per tutti gli } n > n_0.$$

analisi asintotica

La notazione  $O()$  ci permette di esprimere quella che chiamiamo analisi asintotica. Possiamo affermare senza dubbio che, quando  $n$  tende asintoticamente all’infinito, un algoritmo  $O(n)$  è meglio di un algoritmo  $O(n^2)$ . Quest’affermazione ovviamente non potrebbe mai essere supportata dalla singola esecuzione di un benchmark.

La notazione  $O()$  astrae dai fattori costanti, il che la rende più facile da usare ma meno precisa di  $T()$ . Ad esempio, un algoritmo  $O(n^2)$  sarà sempre peggio di un  $O(n)$  a lungo termine, ma se i due algoritmi sono  $T(n^2 + 1)$  e  $T(100n + 1000)$  allora l’algoritmo  $O(n^2)$  sarà effettivamente preferibile per  $n \leq 110$ .

Nonostante questa limitazione, l’analisi asintotica è lo strumento più diffuso per l’analisi degli algoritmi: proprio perché astrae dall’esatto numero delle operazioni svolte (ignorando il fattore costante  $k$ ) e il contenuto preciso dell’input (considerando solo la sua dimensione  $n$ ) l’analisi diventa matematicamente gestibile. La notazione  $O()$  è un buon compromesso tra precisione e facilità di analisi.

## NP e problemi intrinsecamente difficili

L'analisi degli algoritmi e la notazione  $O()$  ci permettono di discutere dell'efficienza di un particolare algoritmo: tuttavia, non possono aiutarci a determinare se per gestire il nostro problema ne è disponibile uno migliore. L'**analisi di complessità** prende in esame i problemi anziché gli algoritmi. La prima, grande suddivisione si ha tra i problemi che possono essere risolti in tempo polinomiale e quelli per i quali ciò non è possibile, indipendentemente dall'algoritmo usato. La classe dei problemi polinomiali – quelli che si possono risolvere in un tempo  $O(n^k)$  con qualche  $k$  – è chiamata P. Talvolta gli autori chiamano “facili” questi problemi, perché la classe contiene anche i problemi con tempi d'esecuzione  $O(\log n)$  e  $O(n)$ . Tuttavia, può contenere anche problemi di complessità  $O(n^{1000})$ , ragion per cui è bene non prendere troppo letteralmente il termine “facili”.

Un'altra classe importante di problemi è NP, quella dei problemi polinomiali non deterministici. Un problema appartiene a questa classe se qualche algoritmo può ipotizzare una soluzione e poi verificare l'ipotesi in tempo polinomiale. L'idea è che avendo un numero arbitrariamente grande di processori, in modo da poter provare tutte le soluzioni contemporaneamente, oppure essendo molto fortunati e indovinando sempre la soluzione giusta al primo colpo, i problemi NP diventerebbero P. Una delle questioni aperte più interessanti dell'informatica è se la classe NP sia equivalente alla P quando si dispone del lusso di un numero infinito di processori o di una infallibile capacità divinatoria. La maggior parte degli informatici è convinta che  $P \neq NP$ , ovvero che i problemi NP siano intrinsecamente difficili e non ammettano algoritmi polinomiali. Questo, tuttavia, non è mai stato dimostrato.

Coloro che si interessano alla questione se  $P = NP$  considerano una sottoclasse di NP, quella dei problemi **NP-completi**. La parola “completi” qui è usata nell'accezione di “più estremi”, e quindi si riferisce ai problemi più difficili della classe NP. È stato dimostrato che ci sono solo due possibilità: o tutti i problemi NP-completi sono P, o non lo è nessuno. Questo rende la classe interessante dal punto di vista teorico: ciò non toglie che lo sia anche dal punto di vista pratico, perché un gran numero di problemi reali importanti sono NP-completi. Un esempio è il problema della soddisfacibilità: data una formula della logica proposizionale, esiste un assegnamento dei valori di verità ai simboli proposizionali tale da rendere vera la formula? A meno che non si verifichi un miracolo e sia  $P = NP$ , non può esistere un algoritmo che risolva *tutti* i problemi di soddisfacibilità in un tempo polinomiale. L'IA, comunque, è più interessata a trovare algoritmi che operano in modo efficiente su problemi *tipici* presi da una distribuzione predeterminata; come abbiamo visto nel Capitolo 7, esistono algoritmi come WALKSAT che si comportano molto bene su un'ampia gamma di problemi.

analisi di complessità

NP-completi

co-NP

La classe **co-NP** è il complemento di NP, nel senso che per ogni problema decisionale in NP ce n'è uno corrispondente in co-NP con le risposte "sì" e "no" invertite. Sappiamo che P è un sottoinsieme sia di NP che di co-NP, e si ritiene che esistano problemi in co-NP che non appartengono a P. I problemi **co-NP-completi** sono i più difficili di co-NP.

La classe  $\#P$  (si dice "diesis P") è l'insieme di problemi di conto che corrispondono ai problemi decisionali in NP. I problemi decisionali hanno una risposta di forma sì/no: c'è una soluzione a questa formula 3-SAT? I problemi di conto hanno come risposta un intero: quante soluzioni ammette questa formula 3-SAT? In alcuni casi il problema di conto è molto più difficile di quello decisionale. Ad esempio, decidere se un grafo bipartito ha una corrispondenza perfetta può essere determinato in un tempo  $O(VE)$  (dove il grafo ha  $V$  vertici ed  $E$  archi), ma il problema di conto "quante corrispondenze perfette ha questo grafo bipartito" è  $\#P$ -completo, il che significa che è difficile quanto ogni altro problema di  $\#P$  e quindi almeno difficile come un problema NP.

Infine, un'altra classe oggetto di studio è quella dei problemi PSPACE, che richiedono una quantità di spazio polinomiale anche su macchine non deterministiche. Si pensa che i problemi PSPACE-difficili siano peggio di quelli NP-completi, benché non sia impossibile che risulti che  $NP = PSPACE$ , proprio come potrebbe darsi che  $P = NP$ .

## A.2 Vettori, matrici e algebra lineare

vettore

I matematici definiscono un **vettore** come il membro di uno spazio vettoriale, ma noi adotteremo una definizione più concreta: un vettore è una sequenza ordinata di valori. Ad esempio, in uno spazio bidimensionale avremo vettori come  $\mathbf{x} = \langle 3, 4 \rangle$  e  $\mathbf{y} = \langle 0, 2 \rangle$ . Seguiremo la convenzione di scrivere i nomi dei vettori in grassetto, benché alcuni autori usino invece una freccia o un trattino sopra il nome:  $\vec{x}$  o  $\bar{y}$ . Si può accedere agli elementi di un vettore usando i pedici:  $\mathbf{z} = \langle z_1, z_2, \dots, z_n \rangle$ .

Le due operazioni fondamentali sono l'addizione tra vettori e la moltiplicazione scalare. L'addizione  $\mathbf{x} + \mathbf{y}$  si effettua sommando gli elementi nelle posizioni corrispondenti:  $\mathbf{x} + \mathbf{y} = \langle 3 + 0, 4 + 2 \rangle = \langle 3, 6 \rangle$ . Nella moltiplicazione scalare, ogni elemento viene moltiplicato per una costante:  $5\mathbf{x} = \langle 5 \times 3, 5 \times 4 \rangle = \langle 15, 20 \rangle$ .

La lunghezza di un vettore si indica con  $|\mathbf{x}|$  e si calcola estraendo la radice quadrata della somma dei quadrati degli elementi:  $|\mathbf{x}| = \sqrt{(3^2 + 4^2)} = 5$ . Il prodotto scalare di due vettori  $\mathbf{x} \cdot \mathbf{y}$  è la somma dei prodotti degli elementi corrispondenti: quindi,  $\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i$ ; nel nostro caso particolare  $\mathbf{x} \cdot \mathbf{y} = 3 \times 0 + 4 \times 2 = 8$ .

Spesso i vettori sono interpretati come segmenti orientati (frecce) in uno spazio euclideo a  $n$  dimensioni. In questo caso la somma di vettori è equivalente a disegnare un vettore con la "coda" in corrispondenza della "testa" dell'altro, mentre il prodotto scalare  $\mathbf{x} \cdot \mathbf{y}$  è uguale a  $|\mathbf{x}| |\mathbf{y}| \cos\theta$ , ove  $\theta$  è l'angolo tra  $\mathbf{x}$  e  $\mathbf{y}$ .

Una matrice è un vettore rettangolare di valori organizzati per righe e colonne. Quella qui sotto è una matrice  $\mathbf{m}$  di dimensioni  $3 \times 4$ :

$$\begin{pmatrix} \mathbf{m}_{1,1} & \mathbf{m}_{1,2} & \mathbf{m}_{1,3} & \mathbf{m}_{1,4} \\ \mathbf{m}_{2,1} & \mathbf{m}_{2,2} & \mathbf{m}_{2,3} & \mathbf{m}_{2,4} \\ \mathbf{m}_{3,1} & \mathbf{m}_{3,2} & \mathbf{m}_{3,3} & \mathbf{m}_{3,4} \end{pmatrix}$$

Il primo indice di  $\mathbf{m}_{i,j}$  specifica la riga, il secondo la colonna. Nei linguaggi di programmazione, spesso  $\mathbf{m}_{i,j}$  si scrive  $\mathbf{m}[i][j]$  o  $\mathbf{m}[i][j]$ .

La somma di due matrici si ottiene sommando gli elementi corrispondenti; quindi  $(\mathbf{m} + \mathbf{n})_{i,j} = \mathbf{m}_{i,j} + \mathbf{n}_{i,j}$ . Se  $\mathbf{m}$  e  $\mathbf{n}$  hanno dimensioni diverse, la somma è indefinita. Possiamo anche definire la moltiplicazione di una matrice per uno scalare:  $(c\mathbf{m})_{i,j} = c\mathbf{m}_{i,j}$ . La moltiplicazione tra due matrici è un po' più complicata. Il prodotto  $\mathbf{mn}$  è definito solo se  $\mathbf{m}$  ha dimensioni  $a \times b$  e  $\mathbf{n}$  ha dimensioni  $b \times c$  (cioè, la seconda matrice deve avere tante righe quante sono le colonne della prima); il risultato è una matrice di dimensioni  $a \times c$ . Questo ovviamente significa che l'operazione non è commutativa: in generale,  $\mathbf{mn} \neq \mathbf{nm}$ . Se le matrici sono di dimensioni appropriate, il risultato è

$$(\mathbf{mn})_{i,k} = \sum_j \mathbf{m}_{i,j} \mathbf{n}_{j,k}$$

La matrice identità  $\mathbf{I}$  ha gli elementi  $I_{i,j}$  pari a uno dove  $i = j$  e uguali a 0 nelle altre posizioni. Ha la proprietà che  $\mathbf{mI} = \mathbf{m}$  per tutti gli  $\mathbf{m}$ . La matrice trasposta di  $\mathbf{m}$ , indicata con  $\mathbf{m}^T$ , è costruita scrivendo le righe al posto delle colonne e viceversa: formalmente,  $\mathbf{m}_{i,j}^T = \mathbf{m}_{j,i}$ .

Le matrici si usano per risolvere sistemi di equazioni lineari applicando un processo noto come **eliminazione di Gauss–Jordan**, un algoritmo  $O(n^3)$ . Considerate il seguente insieme di equazioni, per cui vogliamo una soluzione in  $x, y$  e  $z$ :

$$\begin{aligned} + 2x + y - z &= 8 \\ - 3x - y + 2z &= -11 \\ - 2x + y + 2z &= -3 \end{aligned}$$

Possiamo rappresentare il sistema con una matrice:

$$\begin{pmatrix} x & y & z & c \\ 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{pmatrix}$$

La riga in alto, con l'indicazione delle variabili  $x, y, z, c$ , non fa parte della matrice: l'abbiamo aggiunta noi per chiarezza. Sappiamo che se moltiplichiamo entrambe le parti di un'equazione per una costante o sommiamo due equazioni, ne otteniamo

matrice

eliminazione di  
Gauss–Jordan

mo una ugualmente valida. L'eliminazione di Gauss-Jordan funziona eseguendo ripetutamente tali operazioni in modo tale da cominciare a eliminare la prima variabile ( $x$ ) da tutte le equazioni tranne la prima. Procedendo, eliminiamo la  $i$ -esima variabile da tutte le equazioni tranne la  $i$ -esima e così via per tutte le  $i$ . nel nostro caso, per eliminare  $x$  dalla seconda equazione dobbiamo moltiplicare la prima per  $3/2$  e sommarla alla seconda. Questo ci dà la seguente matrice:

$$\left( \begin{array}{cccc} x & y & z & c \\ 2 & 1 & -1 & 8 \\ 0 & .5 & .5 & 1 \\ -2 & 1 & 2 & -3 \end{array} \right)$$

Continuiamo in questo modo, eliminando  $x$ ,  $y$  e  $z$  fino a ottenere

$$\left( \begin{array}{cccc} x & y & z & c \\ 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array} \right)$$

che indica che  $x = 2$ ,  $y = 3$ ,  $z = -1$  è una soluzione (provate!).

## A.3 Distribuzioni di probabilità

Una probabilità è una misura su un insieme di eventi che soddisfa tre assiomi.

1. La misura di ogni evento è compresa tra 0 e 1. Questo si scrive  $0 \leq P(E = e_i) \leq 1$ , dove  $E$  è una variabile casuale che rappresenta un evento ed  $e_i$  sono i possibili valori di  $E$ . In generale, le variabili casuali si indicano con lettere maiuscole e i loro valori con lettere minuscole.
2. La misura dell'intero insieme è 1:  $\sum_{i=1}^n P(E = e_i) = 1$ .
3. La probabilità dell'unione di eventi disgiunti è pari alla somma delle probabilità dei singoli eventi:  $P(E = e_1 \vee E = e_2) = P(E = e_1) + P(E = e_2)$ , dove  $e_1, e_2$  sono disgiunti.

Un **modello probabilistico** consiste in uno spazio di possibili esiti mutuamente esclusivi insieme alla misura di probabilità associata a ogni esito. Ad esempio, in un modello del tempo che farà domani, gli esiti potrebbero essere *sole*, *nuvole*, *pioggia* e *neve*. Un sottoinsieme di questi esiti costituisce un evento. Ad esempio, l'evento corrispondente a una precipitazione è il sottoinsieme *{pioggia, neve}*.

Useremo  $P(E)$  per denotare il vettore di valori  $\langle P(E = e_1), \dots, P(E = e_n) \rangle$ ,  $P(e_i)$  come abbreviazione di  $P(E = e_i)$  e  $\sum_e P(e)$  per  $\sum_{i=1}^n P(E = e_i)$ .

La probabilità condizionale  $P(B|A)$  è definita come  $P(B \cap A)/P(A)$ .  $A$  e  $B$  sono condizionalmente indipendenti se  $P(B|A) = P(B)$  (o, ciò che è equivalente,  $P(A|B) = P(A)$ ). Nel caso di variabili continue il numero di valori è infinito, e a meno che non ci siano delle cuspidi infinite, la probabilità di un singolo valore è sempre 0. In tal caso si definisce una **funzione densità di probabilità**, indicata anch'essa con  $P(X)$ , che ha un significato leggermente diverso dalla funzione discreta  $P(A)$ . La funzione densità  $P(X = c)$  è definita come il rapporto tra la probabilità che  $X$  cada all'interno di un intervallo intorno al punto  $c$  e l'ampiezza dell'intervallo stesso, quando tale ampiezza tende a zero:

$$P(X = c) = \lim_{dx \rightarrow 0} P(c \leq X \leq c + dx) / dx$$

La funzione densità non può mai essere negativa, e deve sempre risultare

$$\int_{-\infty}^{\infty} P(X) dx = 1$$

Possiamo anche definire una **funzione densità di probabilità cumulativa**  $F(X)$ , che corrisponde alla probabilità che una variabile casuale abbia un valore inferiore a  $x$ :

$$F(X) = \int_{-\infty}^x P(Z) dz$$

Notate che la funzione densità di probabilità è dotata di unità, mentre quella discreta ne è priva. Ad esempio, se  $X$  è misurata in secondi, allora la densità si misura in Hz (ovvero, 1/sec). Se  $X$  è un punto in uno spazio tridimensionale misurato in metri, la densità è misurata in  $1/m^3$ .

Una delle più importanti distribuzioni di probabilità è la **gaussiana**, nota anche come **distribuzione normale**. Una distribuzione gaussiana con media  $\mu$  e deviazione standard  $\sigma$  (e quindi varianza  $\sigma^2$ ) è definita come

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)},$$

dove  $x$  è la variabile continua che va da  $-\infty$  a  $+\infty$ . Con media  $\mu = 0$  e varianza  $\sigma^2 = 1$ , otteniamo il caso speciale della **distribuzione normale standard**. Nel caso di un vettore  $\mathbf{x}$  a  $d$  dimensioni, abbiamo una distribuzione **gaussiana multivariata**:

$$P(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} e^{-\frac{1}{2} (\mathbf{x}-\boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu})}$$

dove  $\boldsymbol{\mu}$  è il vettore media e  $\Sigma$  è la matrice di covarianza della distribuzione.

funzione densità di probabilità

funzione densità di probabilità cumulativa

gaussiana

distribuzione normale standard

gaussiana multivariata

matrice di covarianza

distribuzione cumulativa

teorema del limite centrale

In una dimensione, possiamo anche definire la **distribuzione cumulativa**  $F(x)$  come la probabilità che una variabile casuale sia inferiore a  $x$ . Per la distribuzione normale standard, questa è data da

$$F(x) = \int_{-\infty}^x P(x)dx = \frac{1}{2} \left( 1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \right),$$

dove  $\operatorname{erf}(x)$  è la cosiddetta **funzione di errore**, che non ha alcuna rappresentazione in forma chiusa.

Il **teorema del limite centrale** asserisce che la media di  $n$  variabili casuali tende a una distribuzione normale al tendere di  $n$  a infinito. Questo vale per quasi ogni collezione di variabili casuali, a meno che la varianza di un qualsiasi sottosinsieme finito di variabili domini le altre.

## Note storiche e bibliografiche

La notazione  $O()$ , così diffusa oggi nell'informatica, fu introdotta per la prima volta nel contesto della teoria dei numeri dal matematico tedesco P. G. H. Bachmann (1894). Il concetto di NP-completezza fu inventato da Cook (1971), e il metodo moderno per ridurre un problema a un altro è dovuto a Karp (1972). Grazie a questi lavori Cook e Karp hanno vinto entrambi il premio Turing, la più grande onorificenza per un informatico.

Tra i testi classici sull'analisi e la progettazione degli algoritmi citiamo quelli di Knuth (1973) e Aho, Hopcroft e Ullman (1974); contributi più recenti sono dovuti a Tarjan (1983) e Cormen, Leiserson e Rivest (1990). Questi libri enfatizzano la progettazione e l'analisi di algoritmi per risolvere problemi trattabili. Per la teoria della NP-completezza e le altre forme di intrattabilità potete far riferimento a Garey e Johnson (1979) o Papadimitriou (1994). Oltre a esporre la teoria, Garey e Johnson offrono esempi molto espressivi del perché gli informatici traccino la linea tra problemi trattabili e intrattabili sul confine fra la complessità temporale polinomiale e quella esponenziale. Inoltre forniscono un ampio catalogo di noti problemi NP-completi o altrimenti intrattabili.

Tra i migliori testi sulla probabilità citiamo Chung (1979), Ross (1988), Bertsekas e Tsitsiklis (2002) e Feller (1971).

## Appendice B

# Note sui linguaggi e gli algoritmi

## B.1 Definire i linguaggi con la forma Backus-Naur (BNF)

In questo libro definiamo diversi linguaggi, tra cui la logica proposizionale (pag. 265), quella del primo ordine (pag. 315) e un sottoinsieme del linguaggio naturale (v. Capitolo 22, nel 2° volume). Un linguaggio formale è definito come un insieme di stringhe, ognuna composta da una sequenza di simboli. Tutti i linguaggi che ci interessano consistono in un insieme infinito di stringhe, che dev'essere caratterizzato in modo conciso: per far questo si usa una **grammatica**. Le nostre grammatiche sono tutte scritte nel formalismo chiamato **forma di Backus–Naur (BNF)**. Una grammatica BNF è costituita da quattro componenti.

- ◆ Un insieme di **simboli terminali**. Questi sono i simboli o le parole che formano le stringhe del linguaggio, e possono essere lettere (A, B, C...) o parole (a, abaco, abecedario...).
- ◆ Un insieme di **simboli non terminali** che categorizzano sottoformule del linguaggio. Ad esempio, in italiano il simbolo non terminale *SintagmaNominale* indica un insieme infinito di stringhe che includono “tu” e “il gran cagnone sbavante”.
- ◆ Un **simbolo iniziale**, che è il simbolo non terminale che indica stringhe complete del linguaggio. Per il linguaggio naturale, sarebbe *Frase*; per l’aritmetica, potrebbe essere *Expr*.
- ◆ Un insieme di **regole di riscrittura**, di forma *ParteSin* → *ParteDex*, dove *ParteSin* è un non terminale e *ParteDex* una sequenza di zero o più simboli (terminali o non terminali).

forma di Backus–Naur  
(BNF)

simboli terminali

simboli non terminali

simbolo iniziale

Una regola di riscrittura della forma

$$\text{Frase} \quad \rightarrow \quad \text{SintagmaNominale SintagmaVerbale}$$

significa che ogni volta che abbiamo due stringhe categorizzate come un *SintagmaNominale* e un *SintagmaVerbale*, possiamo concatenarle insieme e categorizzare il risultato come una *Frase*. Per brevità, si può usare il simbolo | per separare parti destre alternative della regola. Ecco una grammatica BNF per semplici espressioni aritmetiche:

$$\begin{array}{lcl} \text{Expr} & \rightarrow & \text{Expr Operatore Expr} \mid (\text{Expr}) \mid \text{Numero} \\ \text{Numero} & \rightarrow & \text{Cifra} \mid \text{Numero Cifra} \\ \text{Cifra} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \text{Operatore} & \rightarrow & + \mid - \mid \div \mid \times \end{array}$$

I linguaggi e le grammatiche sono trattati più dettagliatamente nel Capitolo 22, nel 2° volume. Tenete presente che in altri testi le notazioni BNF potrebbero essere leggermente diverse; potreste avere ad esempio  $\langle \text{Cifra} \rangle$  invece di *Cifra* per un simbolo non terminale, ‘parola’ invece di *parola* per un simbolo terminale,  $::=$  invece di  $\rightarrow$  in una regola.

## B.2 Descrivere gli algoritmi con lo pseudocodice

---

In questo libro definiamo nei dettagli più di 80 algoritmi. Invece di scegliere un linguaggio di programmazione (rischiando così di “perdere per strada” i lettori non familiari con quel particolare linguaggio) abbiamo scelto di descrivere gli algoritmi in pseudocodice. In gran parte, il nostro pseudocodice dovrebbe essere familiare agli utenti di linguaggi come Java, C++ o Lisp. In alcuni punti abbiamo usato formule matematiche o linguaggio naturale per descrivere parti che altrimenti sarebbero state scomode da esprimere. Ecco alcuni particolari da tenere presenti.

**Variabili statiche.** Usiamo la parola chiave *static* per indicare che una variabile riceve un valore iniziale la prima volta che la funzione è invocata e mantiene quel valore (o quello assegnatole in seguito) in tutte le chiamate successive della funzione. In questo le variabili statiche assomigliano a quelle globali, in quanto il loro ciclo di vita va oltre la singola chiamata, ma sono accessibili solo dall’interno della funzione. I programmi agente nel libro usano variabili statiche come “memoria”. I programmi che usano variabili statiche possono essere implementati come oggetti in linguaggi object-oriented come Java e Smalltalk. Nei linguaggi funzionali possono essere implementati come chiusure funzionali nell’ambiente in cui sono definite le variabili richieste.

**Funzioni come valori.** Le funzioni e le procedure sono indicate in maiuscolo, mentre le variabili sono scritte in corsivo minuscolo. La maggior parte delle volte, quindi, una chiamata di funzione avrà l'aspetto FUNZIONE( $x$ ). Tuttavia, permettiamo che il valore di una variabile sia espresso da una funzione; ad esempio, se il valore della variabile  $f$  è la funzione radice quadrata,  $f(9)$  restituirà 3.

**Gli array partono da 1.** A meno che non venga indicato diversamente il primo indice di un array avrà valore 1, come nella consueta notazione matematica, e non 0, come in Java e in C.

**L'indentazione è significativa.** Per indicare il campo d'azione di un ciclo o di un blocco condizionale abbiamo usato l'indentazione, come nel linguaggio Python, e diversamente da Java e C++ (che usano parentesi graffe) o Pascal e Visual Basic (che usano la parola chiave end).

## B.3 Supporto online

---

La maggior parte degli algoritmi presentati sono stati implementati e sono disponibili presso il nostro deposito online di codice:

**aima.cs.berkeley.edu**

Se avete commenti, correzioni o suggerimenti, ci piacerebbe sentirvi. Visitate il nostro sito Web per accedere a liste di discussione, o inviate un messaggio di posta elettronica all'indirizzo:

**aima@cs.berkeley.edu**



# Bibliografia

- Aarup, M., Arentoft, M. M., Parrod, Y., Stader, J. e Stokes, I. (1994). OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. In Fox, M. e Zweben, M. (Eds.), *Knowledge Based Scheduling*. Morgan Kaufmann, San Mateo, California.
- Abramson, B. e Yung, M. (1989). Divide and conquer under global constraints: A solution to the Nqueens problem. *Journal of Parallel and Distributed Computing*, 6 (3), 649–662.
- Ackley, D. H. e Littman, M. L. (1991). Interactions between learning and evolution. In Langton, C., Taylor, C., Farmer, J. D., e Ramussen, S. (Eds.), *Artificial Life II*, pp. 487–509. Addison-Wesley, Redwood City, California.
- Adelson-Velsky, G. M., Arlazarov, V. L., Bitman, A. R., Zhivotovsky, A. A., e Uskov, A. V. (1970). Programming a computer to play chess. *Russian Mathematical Surveys*, 25, 221–262.
- Adelson-Velsky, G. M., Arlazarov, V. L., e Donskoy, M. V. (1975). Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6 (4), 361–371.
- Agre, P. E. e Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 268–272, Milano. Morgan Kaufmann.
- Aho, A. V., Hopcroft, J. e Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts.
- Ait-Kaci, H. e Podelski, A. (1993). Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3–4), 195–234.
- Aldous, D. e Vazirani, U. (1994). “Go with the winners” algorithms. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 492–501, Santa Fe, New Mexico. IEEE Computer Society Press.
- Allen, J. F., Hendler, J. e Tate, A. (Eds.). (1990). *Readings in Planning*. Morgan Kaufmann, San Mateo, California.
- Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the Association for Computing Machinery*, 26(11), 832–843.
- Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, 23, 123–154.
- Allen, J. F. (1991). Time and time again: The many ways to represent time. *International Journal of Intelligent Systems*, 6, 341–355.
- Alterman, R. (1988). Adaptive planning. *Cognitive Science*, 12, 393–422.
- Amarel, S. (1968). On representations of problems of reasoning about actions. In Michie, D. (Ed.), *Machine Intelligence 3*, Vol. 3, pp. 131–171. Elsevier/North-Holland, Amsterdam, London, New York.
- Ambros-Ingerson, J. e Steel, S. (1988). Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pp. 735–740, St. Paul, Minnesota. Morgan Kaufmann.

- Anderson, A. R. (Ed.). (1964). *Minds and Machines*.** Prentice-Hall, Upper Saddle River, New Jersey.
- Anderson, J. A. e Rosenfeld, E. (Eds.). (1988). *Neurocomputing: Foundations of Research*.** MIT Press, Cambridge, Massachusetts.
- Anderson, J. R. (1980). *Cognitive Psychology and Its Implications*.** W. H. Freeman, New York.
- Anshelevich, V. A. (2000).** The game of Hex: An automatic theorem proving approach to game programming. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pp. 189–194, Austin, Texas. AAAI Press.
- Appel, K. e Haken, W. (1977).** Every planar map is four colorable: Part I: Discharging. *Illinois J. Math.*, 21, 429–490
- Apt, K. R. (1999).** The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2), 179–210.
- Armstrong, D. M. (1968).** *A Materialist Theory of the Mind*. Routledge and Kegan Paul, London.
- Arnauld, A. (1662).** *La logique, ou l'art de penser*. Chez Charles Savreux, au pied de la Tour de Nostre Dame, Paris.
- Arora, S. (1998).** Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the Association for Computing Machinery*, 45(5), 753–782.
- Audi, R. (Ed.). (1999).** *The Cambridge Dictionary of Philosophy*. Cambridge University Press, Cambridge, UK.
- Bacchus, F. e van Beek, P. (1998).** On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 311–318, Madison, Wisconsin. AAAI Press.
- Bacchus, F. e van Run, P. (1995).** Dynamic variable ordering in CSPs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pp. 258–275, Cassis, France. Springer-Verlag.
- Bachmann, P. G. H. (1894).** *Die analytische Zahlentheorie*. B. G. Teubner, Leipzig.
- Baker, C. L. (1989).** *English Syntax*. MIT Press, Cambridge, Massachusetts.
- Ballard, B. W. (1983).** The \*-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3), 327–350.
- Baluja, S. (1997).** Genetic algorithms and explicit search statistics. In Mozer, M. C., Jordan, M. I. e Petsche, T. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 9, pp. 319–325. MIT Press, Cambridge, Massachusetts.
- Bancilhon, F., Maier, D., Sagiv, Y. e Ullman, J. D. (1986).** Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, pp. 1–16, New York. ACM Press.
- Barrett, A. e Weld, D. S. (1994).** Taskdecomposition via plan parsing. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp. 1117–1122, Seattle. AAAI Press.
- Bartak, R. (2001).** Theory and practice of constraint propagation. In *Proceedings of the Third Workshop on Constraint Programming for Decision and Control (CPDC-01)*, pp. 7–14, Gliwice, Poland.
- Barto, A. G., Bradtko, S. J. e Singh, S. P. (1995).** Learning to act using real-time dynamic programming. *Artificial Intelligence*, 73(1), 81–138.
- Baum, E. e Smith, W. D. (1997).** A Bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1–2), 195–242.
- Bayardo, R. J. e Schrag, R. C. (1997).** Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pp. 203–208, Providence, Rhode Island. AAAI Press.
- Bayes, T. (1763).** An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53, 370–418.
- Beal, D. F. (1980).** An analysis of minimax. In Clarke, M. R. B. (Ed.), *Advances in Computer Chess 2*, pp. 103–109. Edinburgh University Press, Edinburgh, Scotland.
- Beal, D. F. (1990).** A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1), 85–98.
- Beckert, B. e Posegga, J. (1995).** Leantap: Lean, tableau-based deduction. *Journal of Automated Reasoning*, 15(3), 339–358.
- Beeri, C., Fagin, R., Maier, D. e Yannakakis, M. (1983).** On the desirability of acyclic database schemes. *Journal of the Association for Computing Machinery*, 30(3), 479–513.

- Bell, C. e Tate, A. (1985). Using temporal constraints to restrict search in a planner. In *Proceedings of the Third Alvey IKBS SIG Workshop*, Sunningdale, Oxfordshire, UK. Institution of Electrical Engineers.
- Bell, J. L. e Machover, M. (1977). *A Course in Mathematical Logic*. Elsevier/North-Holland, Amsterdam, London, New York.
- Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press, Princeton, New Jersey.
- Bellman, R. E. (1978). *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd & Fraser Publishing Company, San Francisco.
- Bellman, R. E. e Dreyfus, S. E. (1962). *Applied Dynamic Programming*. Princeton University Press, Princeton, New Jersey.
- Bender, E. A. (1996). *Mathematical methods in artificial intelligence*. IEEE Computer Society Press, Los Alamitos, California.
- Berlekamp, E. R., Conway, J. H. e Guy, R. K. (1982). *Winning Ways, For Your Mathematical Plays*. Academic Press, New York.
- Berliner, H. J. (1977). BKG—a program that plays backgammon. Tech. rep., Computer Science Department, Carnegie-Mellon University, Pittsburgh.
- Berliner, H. J. (1979). The B\* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1), 23–40.
- Berliner, H. J. (1980a). Backgammon computer program beats world champion. *Artificial Intelligence*, 14, 205–220.
- Berliner, H. J. (1980b). Computer backgammon. *Scientific American*, 249(6), 64–72.
- Berliner, H. J. e Ebeling, C. (1989). Pattern knowledge and search: The SUPREM architecture. *Artificial Intelligence*, 38(2), 161–198.
- Berners-Lee, T., Hendler, J. e Lassila, O. (2001). The semantic web. *Scientific American*, 284(5), 34–43.
- Bernoulli, D. (1738). Specimen theoriae novae de mensura sortis. *Proceedings of the St. Petersburg Imperial Academy of Sciences*, 5, 175–192.
- Bernstein, A. e Roberts, M. (1958). Computer vs. chess player. *Scientific American*, 198(6), 96–105.
- Bernstein, A., Roberts, M., Arbuckle, T. e Belsky, M. S. (1958). A chess playing program for the IBM 704. In *Proceedings of the 1958 Western Joint Computer Conference*, pp. 157–159, Los Angeles. American Institute of Electrical Engineers.
- Bertoli, P., Cimatti, A. e Roveri, M. (2001a). Heuristic search + symbolic model checking = efficient conformant planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pp. 467–472, Seattle. Morgan Kaufmann.
- Bertoli, P., Cimatti, A., Roveri, M. e Traverso, P. (2001b). Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pp. 473–478, Seattle. Morgan Kaufmann.
- Bertsekas, D. e Tsitsiklis, J. N. (2002). *Introduction to Probability*. Athena Scientific, Belmont, Massachusetts.
- Bibel, W. (1981). On matrices with connections. *Journal of the Association for Computing Machinery*, 28(4), 633–645.
- Bibel, W. (1993). *Deduction: Automated Logic*. Academic Press, London.
- Biggs, N. L., Lloyd, E. K. e Wilson, R. J. (1986). *Graph Theory 1736–1936*. Oxford University Press, Oxford, UK.
- Biro, J. I. e Shahan, R. W. (Eds.). (1982). *Mind, Brain and Function: Essays in the Philosophy of Mind*. University of Oklahoma Press, Norman, Oklahoma.
- Birtwistle, G., Dahl, O.-J., Myrhaug, B. e Nygaard, K. (1973). *Simula Begin*. Studentliteratur (Lund) and Auerbach, New York.
- Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, UK.
- Bistarelli, S., Montanari, U. e Rossi, F. (1997). Semiring-based constraint satisfaction and optimization. *Journal of the Association for Computing Machinery*, 44(2), 201–236.
- Bitner, J. R. e Reingold, E. M. (1975). Backtrack programming techniques. *Communications of the Association for Computing Machinery*, 18(11), 651–656.
- Block, N. (Ed.). (1980). *Readings in Philosophy of Psychology*, Vol. 1. Harvard University Press, Cambridge, Massachusetts.
- Blum, A. L. e Furst, M. (1995). Fast planning through planning graph analysis. In *Proceedings of*

- the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1636–1642, Montreal. Morgan Kaufmann.
- Blum, A. L. e Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2), 281–300.
- Boden, M. A. (1977). *Artificial Intelligence and Natural Man*. Basic Books, New York.
- Boden, M. A. (Ed.). (1990). *The Philosophy of Artificial Intelligence*. Oxford University Press, Oxford, UK.
- Bonet, B. e Geffner, H. (1999). Planning as heuristic search: New results. In *Proceedings of the European Conference on Planning*, pp. 360–372, Durham, UK. Springer-Verlag.
- Bonet, B. e Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In Chien, S., Kambhampati, S. e Knoblock, C. A. (Eds.), *International Conference on Artificial Intelligence Planning and Scheduling*, pp. 52–61, Menlo Park, California. AAAI Press.
- Boole, G. (1847). *The Mathematical Analysis of Logic: Being an Essay towards a Calculus of Deductive Reasoning*. Macmillan, Barclay e Macmillan, Cambridge.
- Boolos, G. S. e Jeffrey, R. C. (1989). *Computability and Logic* (3rd edition). Cambridge University Press, Cambridge, UK.
- Borgida, A., Brachman, R. J., McGuinness, D. L. e Alperin Resnick, L. (1989). CLASSIC: A structural data model for objects. *SIGMOD Record*, 18(2), 58–67.
- Boutilier, C., Reiter, R., Soutchanski, M. e Thrun, S. (2000). Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pp. 355–362, Austin, Texas. AAAI Press.
- Box, G. E. P. (1957). Evolutionary operation: A method of increasing industrial productivity. *Applied Statistics*, 6, 81–101.
- Boyan, J. A. e Moore, A. W. (1998). Learning evaluation functions for global optimization and Boolean satisfiability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, Wisconsin. AAAI Press.
- Boyer, R. S. e Moore, J. S. (1979). *A computational Logic*. Academic Press, New York.
- Brachman, R. J. (1979). On the epistemological status of semantic networks. In Findler, N. V. (Ed.), *Associative Networks: Representation and Use of Knowledge by Computers*, pp. 3–50. Academic Press, New York.
- Brachman, R. J. e Levesque, H. J. (Eds.). (1985). *Readings in Knowledge Representation*. Morgan Kaufmann, San Mateo, California.
- Brachman, R. J., Fikes, R. E. e Levesque, H. J. (1983). Krypton: A functional approach to knowledge representation. *Computer*, 16(10), 67–73.
- Bratko, I. (1986). *Prolog Programming for Artificial Intelligence* (1st edition). Addison-Wesley, Reading, Massachusetts.
- Bratko, I. (2001). *Prolog Programming for Artificial Intelligence* (Third edition). Addison-Wesley, Reading, Massachusetts.
- Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, Massachusetts.
- Bratman, M. E. (1992). Planning and the stability of intention. *Minds and Machines*, 2 (1), 1–16.
- Brelaz, D. (1979). New methods to color the vertices of a graph. *Communications of the Association for Computing Machinery*, 22(4), 251–256.
- Brent, R. P. (1973). *Algorithms for minimization without derivatives*. Prentice-Hall, Upper Saddle River, New Jersey.
- Brewka, G., Dix, J. e Konolige, K. (1997). *Nonmonotonic Reasoning: An Overview*. CSLI Publications, Stanford, California.
- Briggs, R. (1985). Knowledge representation in Sanskrit and artificial intelligence. *AI Magazine*, 6 (1), 32–39.
- Broadbent, D. E. (1958). *Perception and communication*. Pergamon, Oxford, UK.
- Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2, 14–23.
- Brooks, R. A. (1989). Engineering approach to building complete, intelligent beings. *Proceedings of the SPIE—the International Society for Optical Engineering* 1002, 618–625.
- Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47(1–3), 139–159.

- Brownston, L., Farrell, R., Kant, E. e Martin, N. (1985). *Programming expert systems in OPS5: An introduction to rule-based programming*. Addison-Wesley, Reading, Massachusetts.
- Brudno, A. L. (1963). Bounds and valuations for shortening the scanning of variations. *Problems of Cybernetics*, 10, 225–241.
- Bryant, R. E. (1992). Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), 293–318.
- Bryson, A. E. e Ho, Y.-C. (1969). *Applied Optimal Control*. Blaisdell, New York.
- Buchanan, B. G., Mitchell, T. M., Smith, R. G. e Johnson, C. R. (1978). Models of learning systems. In *Encyclopedia of Computer Science and Technology*, Vol. 11. Dekker, New York.
- Buchanan, B. G., Sutherland, G. L. e Feigenbaum, E. A. (1969). Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry. In Meltzer, B., Michie, D. e Swann, M. (Eds.), *Machine Intelligence 4*, pp. 209–254. Edinburgh University Press, Edinburgh, Scotland.
- Bundy, A. (1999). A survey of automated deduction. In Wooldridge, M. J. e Veloso, M. (Eds.), *Artificial intelligence today: Recent trends and developments*, pp. 153–174. Springer-Verlag, Berlin.
- Bunt, H. C. (1985). The formal representation of (quasi-) continuous concepts. In Hobbs, J. R. e Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 2, pp. 37–70. Ablex, Norwood, New Jersey.
- Buro, M. (2002). Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134(1–2), 85–99.
- Burstall, R. M. (1974). Program proving as hand simulation with a little induction. In *Information Processing '74*, pp. 308–312. Elsevier/North-Holland, Amsterdam, London, New York.
- Burstall, R. M. e Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1), 44–67.
- Bylander, T. (1992). Complexity results for serial decomposability. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 729–734, San Jose. AAAI Press.
- Calvanese, D., Lenzerini, M. e Nardi, D. (1999). Unifying class-based representation formalisms. *Journal of Artificial Intelligence Research*, 11, 199–240.
- Campbell, M. S., Hoane, A. J. e Hsu, F.-H. (2002). Deep Blue. *Artificial Intelligence*, 134(1–2), 57–83.
- Carbonell, J. G. (1983). Derivational analogy and its role in problem solving. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-83)*, pp. 64–69, Washington, DC. Morgan Kaufmann.
- Carbonell, J. G., Knoblock, C. A. e Minton, S. (1989). PRODIGY: An integrated architecture for planning and learning. Technical report CMU-CS-89-189, Computer Science Department, Carnegie-Mellon University, Pittsburgh.
- Carnap, R. (1928). *Der logische Aufbau der Welt*. Weltkreis-verlag, Berlin-Schlachtensee. Translated into English as (Carnap, 1967).
- Carnap, R. (1948). On the application of inductive logic. *Philosophy and Phenomenological Research*, 8, 133–148.
- Carnap, R. (1950). *Logical Foundations of Probability*. University of Chicago Press, Chicago.
- Cartesio, R. (1637). *Discorso sul metodo*.
- Ceri, S., Gottlob, G. e Tanca, L. (1990). *Logic programming and databases*. Springer-Verlag, Berlin.
- Chakrabarti, P. P., Ghose, S., Acharya, A. e de Sarkar, S. C. (1989). Heuristic search in restricted memory. *Artificial Intelligence*, 41(2), 197–222.
- Chandra, A. K. e Harel, D. (1980). Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2), 156–178.
- Chandra, A. K. e Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pp. 77–90, New York. ACM Press.
- Chang, C.-L. e Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York.
- Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32(3), 333–377.
- Charniak, E. e McDermott, D. (1985). *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts.
- Charniak, E., Riesbeck, C., McDermott, D. e Meehan, J. (1987). *Artificial Intelligence Programming* (2nd edition). Lawrence Erlbaum Associates, Potomac, Maryland.

- Cheeseman, P. (1985). In defense of probability. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 1002–1009, Los Angeles. Morgan Kaufmann.
- Cheeseman, P., Kanefsky, B. e Taylor, W. (1991). Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pp. 331–337, Sydney. Morgan Kaufmann.
- Chierchia, G. e McConnell-Ginet, S. (1990). *Meaning and Grammar*. MIT Press, Cambridge, Massachusetts.
- Chomsky, N. (1980). Rules and representations. *The Behavioral and Brain Sciences*, 3, 1–61.
- Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2 (3), 113–124.
- Chomsky, N. (1957). *Syntactic Structures*. Mouton, The Hague and Paris.
- Chung, K. L. (1979). *Elementary Probability Theory with Stochastic Processes* (3rd edition). Springer-Verlag, Berlin.
- Church, A. (1936). A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1, 40–41 e 101–102.
- Churchland, P. M. e Churchland, P. S. (1982). Functionalism, qualia, and intentionality. In Biro, J. I. e Shahan, R. W. (Eds.), *Mind, Brain and Function: Essays in the Philosophy of Mind*, pp. 121–145. University of Oklahoma Press, Norman, Oklahoma.
- Churchland, P. S. (1986). *Neurophilosophy: Toward a Unified Science of the Mind–Brain*. MIT Press, Cambridge, Massachusetts.
- Cimatti, A., Roveri, M. e Traverso, P. (1998). Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 875–881, Madison, Wisconsin. AAAI Press.
- Clark, K. L. (1978). Negation as failure. In Gallaire, H. e Minker, J. (Eds.), *Logic and Data Bases*, pp. 293–322. Plenum, New York.
- Clarke, E. e Grumberg, O. (1987). Research on automatic verification of finite-state concurrent systems. *Annual Review of Computer Science*, 2, 269–290.
- Clarke, E., Grumberg, O. e Peled, D. (1999). *Model Checking*. MIT Press, Cambridge, Massachusetts.
- Clocksin, W. F. e Mellish, C. S. (1994). *Programming in Prolog* (4th edition). Springer-Verlag, Berlin.
- Cobham, A. (1964). The intrinsic computational difficulty of functions. In Bar-Hillel, Y. (Ed.), *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, pp. 24–30, Jerusalem. Elsevier/North-Holland.
- Cohen, J. (1988). A view of the origins and development of PROLOG. *Communications of the Association for Computing Machinery*, 31, 26–36.
- Cohen, P. R. (1995). *Empirical methods for artificial intelligence*. MIT Press, Cambridge, Massachusetts.
- Cohen, P. R. e Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42(2–3), 213–261.
- Cohen, P. R., Morgan, J. e Pollack, M. E. (1990). *Intentions in Communication*. MIT Press, Cambridge, Massachusetts.
- Cohn, A. G., Bennett, B., Gooday, J. M. e Gotts, N. (1997). RCC: A calculus for region based qualitative spatial reasoning. *GeoInformatica*, 1, 275–316.
- Colmenerau, A., Kanoui, H., Pasero, R. e Roussel, P. (1973). Un système de communication homme–machine en Français. Rapport, Groupe d’Intelligence Artificielle, Université d’Aix-Marseille II.
- Condon, J. H. e Thompson, K. (1982). Belle chess hardware. In Clarke, M. R. B. (Ed.), *Advances in Computer Chess 3*, pp. 45–54. Pergamon, Oxford, UK.
- Cook, S. A. (1971). The complexity of theoremproving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158, New York. ACM Press.
- Cook, S. A. e Mitchell, D. (1997). Finding hard instances of the satisfiability problem: A survey. In Du, D., Gu, J. e Pardalos, P. (Eds.), *Satisfiability problems: Theory and applications*. American Mathematical Society, Providence, Rhode Island.
- Copeland, J. (1993). *Artificial Intelligence: A Philosophical Introduction*. Blackwell, Oxford, UK.
- Cormen, T. H., Leiserson, C. E. e Rivest, R. (1990). *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts.

- Craik, K. J. (1943). *The Nature of Explanation*. Cambridge University Press, Cambridge, UK.
- Crawford, J. M. e Auton, L. D. (1993). Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 21–27, Washington, DC. AAAI Press.
- Crockett, L. (1994). *The Turing Test and the Frame Problem: AI's Mistaken Understanding of Intelligence*. Ablex, Norwood, New Jersey.
- Cross, S. E. e Walker, E. (1994). Dart: Applying knowledge based planning and scheduling to crisis action planning. In Zweben, M. e Fox, M. S. (Eds.), *Intelligent Scheduling*, pp. 711–729. Morgan Kaufmann, San Mateo, California.
- Culberson, J. e Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, 14(4), 318–334.
- Cullingford, R. E. (1981). Integrating knowledge sources for computer “understanding” tasks. *IEEE Transactions on Systems, Man and Cybernetics (SMC)*, 11.
- Dahl, O.-J., Myrhaug, B. e Nygaard, K. (1970). (Simula 67) common base language. Tech. rep. N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo.
- Dantzig, G. B. (1949). Programming of interdependent activities: II. mathematical model. *Econometrica*, 17, 200–211.
- Dasgupta, P., Chakrabarti, P. P. e DeSarkar, S. C. (1994). Agent searching in a tree and the optimality of iterative deepening. *Artificial Intelligence*, 71, 195–208.
- Davidson, D. (1980). *Essays on Actions and Events*. Oxford University Press, Oxford, UK.
- Davis, E. (1986). *Representing and Acquiring Geographic Knowledge*. Pitman e Morgan Kaufmann, London e San Mateo, California.
- Davis, E. (1990). *Representations of Commonsense Knowledge*. Morgan Kaufmann, San Mateo, California.
- Davis, M. (1957). A computer program for Presburger’s algorithm. In Robinson, A. (Ed.), *Proving Theorems (as Done by Man, Logician, or Machine)*, pp. 215–233, Cornell University, Ithaca, New York. Communications Research Division, Institute for Defense Analysis. Proceedings of the Summer Institute for Symbolic Logic. Second edition; publication date is 1960.
- Davis, M. e Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7 (3), 201–215.
- de Kleer, J. (1975). Qualitative and quantitative knowledge in classical mechanics. Tech. rep. AI-TR-352, MIT Artificial Intelligence Laboratory.
- de Kleer, J. (1986a). An assumption-based TMS. *Artificial Intelligence*, 28(2), 127–162.
- de Kleer, J. (1986b). Extending the ATMS. *Artificial Intelligence*, 28(2), 163–196.
- de Kleer, J. (1986c). Problem solving with the ATMS. *Artificial Intelligence*, 28(2), 197–224.
- de Kleer, J. (1989). A comparison of ATMS and CSP techniques. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Vol. 1, pp. 290–296, Detroit. Morgan Kaufmann.
- de Kleer, J. e Brown, J. S. (1985). A qualitative physics based on confluences. In Hobbs, J. R. e Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 4, pp. 109–183. Ablex, Norwood, New Jersey.
- de Marcken, C. (1996). *Unsupervised Language Acquisition*. Ph.D. thesis, MIT.
- De Morgan, A. (1864). On the syllogism IV and on the logic of relations. *Cambridge Philosophical Transactions*, x, 331–358.
- Deacon, T. W. (1997). *The symbolic species: The coevolution of language and the brain*. W. W. Norton, New York.
- Deale, M., Yvanovich, M., Schnitzius, D., Kautz, D., Carpenter, M., Zweben, M., Davis, G. e Daun, B. (1994). The space shuttle ground processing scheduling system. In Zweben, M. e Fox, M. (Eds.), *Intelligent Scheduling*, pp. 423–449. Morgan Kaufmann, San Mateo, California.
- Dean, T., Firby, R. J. e Miller, D. (1990). Hierarchical planning involving deadlines, travel time, and resources. *Computational Intelligence*, 6 (1), 381–398.
- Dechter, R. (1990a). Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41, 273–312.
- Dechter, R. (1990b). On the expressiveness of networks with hidden variables. In *Proceedings of the*

- Eighth National Conference on Artificial Intelligence (AAAI-90)*, pp. 379–385, Boston. MIT Press.
- Dechter, R. (1992). Constraint networks. In Shapiro, S. (Ed.), *Encyclopedia of Artificial Intelligence* (2nd edition), pp. 276–285. Wiley and Sons, New York.
- Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113, 41–85.
- Dechter, R. e Frost, D. (1999). Backtracking algorithms for constraint satisfaction problems. Tech. rep., Department of Information and Computer Science, University of California, Irvine.
- Dechter, R. e Pearl, J. (1985). Generalized best-first search strategies and the optimality of A\*. *Journal of the Association for Computing Machinery*, 32(3), 505–536.
- Dechter, R. e Pearl, J. (1987). Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1), 1–38.
- Dedekind, R. (1888). *Was sind und was sollen die Zahlen*. Braunschweig, Germany.
- Deng, X. e Papadimitriou, C. H. (1990). Exploring an unknown graph. In *Proceedings 31st Annual Symposium on Foundations of Computer Science*, pp. 355–361, St. Louis. IEEE Computer Society Press.
- Dennett, D. C. (1971). Intentional systems. *The Journal of Philosophy*, 68(4), 87–106.
- Dennett, D. C. (1978). Why you can't make a computer that feels pain. *Synthese*, 38(3).
- Deo, N. e Pang, C.-Y. (1984). Shortest path algorithms: Taxonomy and annotation. *Networks*, 14(2), 275–323.
- Descotte, Y. e Latombe, J.-C. (1985). Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, 27, 183–217.
- DiGioia, A. M., Kanade, T. e Wells, P. (1996). Final report of the second international workshop on robotics and computer assisted medical interventions. *Computer Aided Surgery*, 2, 69–101.
- Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 269–271.
- Do, M. B. e Kambhampati, S. (2001). Sapa: A domain-independent heuristic metric temporal planner. In *Proceedings of the European Conference on Planning*, Toledo, Spain. Springer-Verlag.
- Doran, J. e Michie, D. (1966). Experiments with the graph traverser program. *Proceedings of the Royal Society of London*, 294, Series A, 235–259.
- Dorf, R. C. e Bishop, R. H. (1999). *Modern Control Systems*. Addison-Wesley, Reading, Massachusetts.
- Dowling, W. F. e Gallier, J. H. (1984). Linear-time algorithms for testing the satisfiability of propositional Horn formulas. *Journal of Logic Programming*, 1, 267–284.
- Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12(3), 231–272.
- Doyle, J. (1983). What is rational psychology? Toward a modern mental philosophy. *AI Magazine*, 4 (3), 50–53.
- Doyle, J. e Patil, R. (1991). Two theses of knowledge representation: Language restrictions, taxonomic classification e the utility of representation services. *Artificial Intelligence*, 48(3), 261–297.
- Drabble, B. (1990). Mission scheduling for spacecraft: Diaries of T-SCHED. In *Expert Planning Systems*, pp. 76–81, Brighton, UK. Institute of Electrical Engineers.
- Draper, D., Hanks, S. e Weld, D. S. (1994). Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on AI Planning Systems*, pp. 31–36, Chicago. Morgan Kaufmann.
- Dreyfus, H. L. (1972). *What Computers Can't Do: A Critique of Artificial Reason*. Harper and Row, New York.
- Dreyfus, S. E. (1969). An appraisal of some shortestpaths algorithms. *Operations Research*, 17, 395–412.
- Du, D., Gu, J. e Pardalos, P. M. (Eds.). (1999). *Optimization methods for logical inference*. American Mathematical Society, Providence, Rhode Island.
- Durfee, E. H. e Lesser, V. R. (1989). Negotiating task decomposition and allocation using partial global planning. In Huhns, M. and Gasser, L. (Eds.), *Distributed AI*, Vol. 2. Morgan Kaufmann, San Mateo, California.
- Dyer, M. (1983). *In-Depth Understanding*. MIT Press, Cambridge, Massachusetts.
- Ebeling, C. (1987). *All the Right Moves*. MIT Press, Cambridge, Massachusetts.
- Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17, 449–467.

- Edwards, P. (Ed.). (1967). *The Encyclopedia of Philosophy*. Macmillan, London.**
- Eiter, T., Leone, N., Mateis, C., Pfeifer, G. e Scarcello, F. (1998). The KR system dlv: Progress report, comparisons and benchmarks. In Cohn, A., Schubert, L. e Shapiro, S. (Eds.), *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 406–417, Trento, Italy.**
- Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press, New York.**
- Erdmann, M. A. e Mason, M. (1988). An exploration of sensorless manipulation. *IEEE Journal of Robotics and Automation*, 4 (4), 369–379.**
- Ernst, M., Millstein, T. e Weld, D. S. (1997). Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pp. 1169–1176, Nagoya, Japan. Morgan Kaufmann.**
- Erol, K., Hendler, J. e Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp. 1123–1128, Seattle. AAAI Press.**
- Erol, K., Hendler, J. e Nau, D. S. (1996). Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1), 69–93.**
- Etzioni, O. e Weld, D. S. (1994). A softbot-based interface to the Internet. *Communications of the Association for Computing Machinery*, 37(7), 72–76.**
- Etzioni, O., Hanks, S., Weld, D. S., Draper, D., Lesh, N. e Williamson, M. (1992). An approach to planning with incomplete information. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, Massachusetts.**
- Evans, T. G. (1968). A program for the solution of a class of geometric-analogy intelligence-test questions. In Minsky, M. L. (Ed.), *Semantic Information Processing*, pp. 271–353. MIT Press, Cambridge, Massachusetts.**
- Fagin, R., Halpern, J. Y., Moses, Y. e Vardi, M. Y. (1995). *Reasoning about Knowledge*. MIT Press, Cambridge, Massachusetts.**
- Fahlman, S. E. (1974). A planning system for robot construction tasks. *Artificial Intelligence*, 5 (1), 1–49.**
- Fahlman, S. E. (1979). *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, Cambridge, Massachusetts.**
- Feigenbaum, E. A. e Feldman, J. (Eds.). (1963). *Computers and Thought*. McGraw-Hill, New York.**
- Feigenbaum, E. A., Buchanan, B. G. e Lederberg, J. (1971). On generality and problem solving: A case study using the DENDRAL program. In Meltzer, B. e Michie, D. (Eds.), *Machine Intelligence 6*, pp. 165–190. Edinburgh University Press, Edinburgh, Scotland.**
- Feller, W. (1971). *An Introductioon to Probability Theory and its Applications*, Vol. 2. John Wiley.**
- Ferraris, P. e Giunchiglia, E. (2000). Planning as satisability in nondeterministic domains. In *Proceedings of Seventeenth National Conference on Artificial Intelligence*, pp. 748–753. AAAI Press.**
- Fikes, R. E. e Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2 (3–4), 189–208.**
- Fikes, R. E. e Nilsson, N. J. (1993). STRIPS, a retrospective. *Artificial Intelligence*, 59(1–2), 227–232.**
- Fikes, R. E., Hart, P. E. e Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3 (4), 251–288.**
- Findlay, J. N. (1941). Time: A treatment of some puzzles. *Australasian Journal of Psychology and Philosophy*, 19(3), 216–235.**
- Firby, R. J. (1996). Modularity issues in reactive planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pp. 78–85, Edinburgh, Scotland. AAAI Press.**
- Fischer, M. J. e Ladner, R. E. (1977). Propositional modal logic of programs. In *Proceedings of the 9th ACM Symposium on the Theory of Computing*, pp. 286–294, New York. ACM Press.**
- Fogel, D. B. (2000). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, New Jersey.**
- Fogel, L. J., Owens, A. J. e Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. Wiley, New York.**
- Forbus, K. D. (1985). The role of qualitative dynamics in naïve physics. In Hobbs, J. R. e Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 5, pp. 185–226. Ablex, Norwood, New Jersey.**

- Forbus, K. D. e de Kleer, J.** (1993). *Building Problem Solvers*. MIT Press, Cambridge, Massachusetts.
- Ford, K. M. e Hayes, P. J.** (1995). Turing Test considered harmful. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 972–977, Montreal. Morgan Kaufmann.
- Forgy, C.** (1981). OPS5 user's manual. Technical report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh.
- Forgy, C.** (1982). A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1), 17–37.
- Fourier, J.** (1827). Analyse des travaux de l'Académie Royale des Sciences, pendant l'année 1824; partie mathématique. *Histoire de l'Académie Royale des Sciences de France*, 7, xlviil–l
- Fox, M. S. e Long, D.** (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9, 367–421.
- Fox, M. S.** (1990). Constraint-guided scheduling: A short history of research at CMU. *Computers in Industry*, 14(1–3), 79–88.
- Fox, M. S., Allen, B. e Strohm, G.** (1982). Job shop scheduling: An investigation in constraint-directed reasoning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-82)*, pp. 155–158, Pittsburgh, Pennsylvania. Morgan Kaufmann.
- Franco, J. e Paull, M.** (1983). Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5, 77–87.
- Frege, G.** (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle, Berlin. English translation appears in van Heijenoort (1967).
- Freuder, E. C.** (1985). A sufficient condition for backtrack-bounded search. *Journal of the Association for Computing Machinery*, 32(4), 755–761.
- Freuder, E. C. e Mackworth, A. K. (Eds.)**. (1994). *Constraint-based reasoning*. MIT Press, Cambridge, Massachusetts.
- Friedberg, R. M.** (1958). A learning machine: Part I. *IBM Journal*, 2, 2–13.
- Friedberg, R. M., Dunham, B. e North, T.** (1959). A learning machine: Part II. *IBM Journal of Research and Development*, 3 (3), 282–287.
- Friedman, G. J.** (1959). Digital simulation of an evolutionary process. *General Systems Yearbook*, 4, 171–184.
- Fuchs, J. J., Gasquet, A., Olalainy, B. e Currie, K. W.** (1990). PlanERS-1: An expert planning system for generating spacecraft mission plans. In *First International Conference on Expert Planning Systems*, pp. 70–75, Brighton, UK. Institute of Electrical Engineers.
- Fukunaga, A. S., Rabideau, G., Chien, S. e Yan, D.** (1997). ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *Proceedings of the International Symposium on AI, Robotics and Automation in Space*, pp. 181–187, Tokyo.
- Gallaire, H. e Minker, J. (Eds.)**. (1978). *Logic and Databases*. Plenum, New Y.
- Gallier, J. H.** (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper and Row, New York.
- Gallo, G. e Pallottino, S.** (1988). Shortest path algorithms. *Annals of Operations Research*, 13, 3–79.
- Gardner, M.** (1968). *Logic Machines, Diagrams and Boolean Algebra*. Dover, New York.
- Garey, M. R. e Johnson, D. S.** (1979). *Computers and Intractability*. W. H. Freeman, New York.
- Gaschnig, J.** (1977). A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, p. 457, Cambridge, Massachusetts. IJCAII.
- Gaschnig, J.** (1979). Performance measurement and analysis of certain search algorithms. Technical report CMU-CS-79-124, Computer Science Department, Carnegie-Mellon University.
- Gasser, R.** (1995). *Efficiently harnessing computational resources for exhaustive search*. Ph.D. thesis, ETH Zürich, Switzerland.
- Gasser, R.** (1998). Solving nine men's morris. In Nowakowski, R. (Ed.), *Games of No Chance*. Cambridge University Press, Cambridge, UK.
- Gauss, K. F.** (1829). Beiträge zur theorie der algebraischen gleichungen. Collected in *Werke*, Vol. 3, pages 71–102. K. Gesellschaft Wissenschaft, Göttingen, Germany, 1876.
- Gelernter, H.** (1959). Realization of a geometrytheorem proving machine. In *Proceedings of an International Conference on Information Processing*, pp. 273–282, Paris. UNESCO House.

- Gelfond, M. e Lifschitz, V.** (1988). Compiling circumscriptive theories into logic programs. In Reinfrank, M., de Kleer, J., Ginsberg, M. L. e Sandewall, E. (Eds.), *Non-Monotonic Reasoning: 2nd International Workshop Proceedings*, pp. 74–99, Grassau, Germany. Springer-Verlag.
- Genesereth, M. R.** (1984). The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24(1–3), 411–436.
- Genesereth, M. R. e Nilsson, N. J.** (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, San Mateo, California.
- Genesereth, M. R. e Nourbakhsh, I.** (1993). Timesaving tips for problem solving with incomplete information. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 724–730, Washington, DC. AAAI Press.
- Gentzen, G.** (1934). Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39, 176–210, 405–431.
- Georgeff, M. P. e Lansky, A. L.** (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pp. 677–682, Seattle. Morgan Kaufmann.
- Gerevini, A. e Schubert, L. K.** (1996). Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5, 95–137.
- Gerevini, A. e Serina, I.** (2002). LPG: A planner based on planning graphs with action costs. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling*, pp. 281–290, Menlo Park, California. AAAI Press.
- Ghallab, M., Howe, A., Knoblock, C. A. e McDermott, D.** (1998). PDDL—the planning domain definition language. Tech. rep. DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut.
- Ghallab, M. e Laruelle, H.** (1994). Representation and control in IxTeT, a temporal planner. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pp. 61–67, Chicago. AAAI Press.
- Giacomo, G. D., Lespérance, Y. e Levesque, H. J.** (2000). ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121, 109–169.
- Gilmore, P. C.** (1960). A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, 4, 28–35.
- Ginsberg, M. L.** (1989). Universal planning: An (almost) universally bad idea. *AI Magazine*, 10(4), 40–44.
- Ginsberg, M. L.** (1993). *Essentials of Artificial Intelligence*. Morgan Kaufmann, San Mateo, California.
- Ginsberg, M. L.** (1999). GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 584–589, Stockholm. Morgan Kaufmann.
- Ginsberg, M. L., Frank, M., Halpin, M. P. e Torrance, M. C.** (1990). Search lessons learned from crossword puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, Vol. 1, pp. 210–215, Boston. MIT Press.
- Glover, F.** (1989). Tabu search: 1. *ORSA Journal on Computing*, 1 (3), 190–206.
- Glover, F. e Laguna, M. (Eds.)**. (1997). *Tabu search*. Kluwer, Dordrecht, Netherlands.
- Gödel, K.** (1930). *Über die Vollständigkeit des Logikkalküls*. Ph.D. thesis, University of Vienna.
- Golden, K.** (1998). Leap before you look: Information gathering in the PUCCINI planner. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pp. 70–77, Pittsburgh, Pennsylvania. AAAI Press.
- Goldman, R. e Boddy, M.** (1996). Expressive planning and explicit knowledge. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pp. 110–117, Edinburgh, Scotland. AAAI Press.
- Gomes, C., Selman, B. e Kautz, H.** (1998). Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 431–437, Madison, Wisconsin. AAAI Press.
- Good, I. J.** (1950). Contribution to the discussion of Eliot Slater's "Statistics for the chess computer and the factor of mobility". In *Symposium on Information Theory*, p. 199, London. Ministry of Supply.
- Goodman, D. e Keene, R.** (1997). *Man versus Machine: Kasparov versus Deep Blue*. H3 Publications, Cambridge, Massachusetts.

- Goodman, N.** (1977). *The Structure of Appearance* (3rd edition). D. Reidel, Dordrecht, Netherlands.
- Gordon, M. J., Milner, A. J. e Wadsworth, C. P.** (1979). *Edinburgh LCF*. Springer-Verlag, Berlin.
- Gottlob, G., Leone, N. e Scarcello, F.** (1999a). A comparison of structural CSP decomposition methods. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 394–399, Stockholm. Morgan Kaufmann.
- Gottlob, G., Leone, N. e Scarcello, F.** (1999b). Hypertree decompositions and tractable queries. In *Proceedings of the 18th ACM International Symposium on Principles of Database Systems*, pp. 21–32, Philadelphia. Association for Computing Machinery.
- Grassmann, H.** (1861). *Lehrbuch der Arithmetik*. Th. Chr. Fr. Enslin, Berlin.
- Green, C.** (1969a). Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-69)*, pp. 219–239, Washington, DC. IJCAI.
- Green, C.** (1969b). Theorem-proving by resolution as a basis for question-answering systems. In Meltzer, B., Michie, D. e Swann, M. (Eds.), *Machine Intelligence 4*, pp. 183–205. Edinburgh University Press, Edinburgh, Scotland.
- Green, C. e Raphael, B.** (1968). The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 23rd ACM National Conference*, Washington, DC. ACM Press.
- Greenblatt, R. D., Eastlake, D. E. e Crocker, S. D.** (1967). The Greenblatt chess program. In *Proceedings of the Fall Joint Computer Conference*, pp. 801–810. American Federation of Information Processing Societies (AFIPS).
- Gu, J.** (1989). *Parallel Algorithms and Architectures for Very Fast AI Search*. Ph.D. thesis, University of Utah.
- Guard, J., Oglesby, F., Bennett, J. e Settle, L.** (1969). Semi-automated mathematics. *Journal of the Association for Computing Machinery*, 16, 49–62.
- Haas, A.** (1986). A syntactic theory of belief and action. *Artificial Intelligence*, 28(3), 245–292.
- Hammond, K.** (1989). *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press, New York.
- Hamscher, W., Console, L. e Kleer, J. D.** (1992). *Readings in Model-based Diagnosis*. Morgan Kaufmann, San Mateo, California.
- Hansen, E. e Zilberstein, S.** (2001). LAO\*: a heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1–2), 35–62.
- Hansen, P. e Jaumard, B.** (1990). Algorithms for the maximum satisfiability problem. *Computing*, 44(4), 279–303.
- Hanski, I. e Cambefort, Y. (Eds.)**. (1991). *Dung Beetle Ecology*. Princeton University Press, Princeton, New Jersey.
- Hansson, O., Mayer, A. e Yung, M.** (1992). Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3), 207–227.
- Haralick, R. M. e Elliot, G. L.** (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3), 263–313.
- Harel, D.** (1984). Dynamic logic. In Gabbay, D. e Guenther, F. (Eds.), *Handbook of Philosophical Logic*, Vol. 2, pp. 497–604. D. Reidel, Dordrecht, Netherlands.
- Harman, G. H.** (1983). *Change in View: Principles of Reasoning*. MIT Press, Cambridge, Massachusetts.
- Hart, P. E., Nilsson, N. J. e Raphael, B.** (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2), 100–107.
- Hart, P. E., Nilsson, N. J. e Raphael, B.** (1972). Correction to “A formal basis for the heuristic determination of minimum cost paths”. *SIGART Newsletter*, 37, 28–29.
- Hart, T. P. e Edwards, D. J.** (1961). The tree prune (TP) algorithm. Artificial intelligence project memo 30, Massachusetts Institute of Technology, Cambridge, Massachusetts.
- Haslum, P. e Geffner, H.** (2001). Heuristic planning with time and resources. In *Proceedings of the IJCAI-01 Workshop on Planning with Resources*, Seattle.
- Haugeland, J. (Ed.)**. (1981). *Mind Design*. MIT Press, Cambridge, Massachusetts.
- Haugeland, J. (Ed.)**. (1985). *Artificial Intelligence: The Very Idea*. MIT Press, Cambridge, Massachusetts.

- Hayes, P. J.** (1978). The naive physics manifesto. In Michie, D. (Ed.), *Expert Systems in the Microelectronic Age*. Edinburgh University Press, Edinburgh, Scotland.
- Hayes, P. J.** (1979). The logic of frames. In Metzing, D. (Ed.), *Frame Conceptions and Text Understanding*, pp. 46–61. de Gruyter, Berlin.
- Hayes, P. J.** (1985a). Naive physics I: Ontology for liquids. In Hobbs, J. R. e Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 3, pp. 71–107. Ablex, Norwood, New Jersey.
- Hayes, P. J.** (1985b). The second naive physics manifesto. In Hobbs, J. R. e Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 1, pp. 1–36. Ablex, Norwood, New Jersey.
- Hebb, D. O.** (1949). *The Organization of Behavior*. Wiley, New York.
- Heckerman, D.** (1991). *Probabilistic Similarity Networks*. MIT Press, Cambridge, Massachusetts.
- Heinz, E. A.** (2000). *Scalable search in computer chess*. Vieweg, Braunschweig, Germany.
- Held, M. e Karp, R. M.** (1970). The traveling salesman problem and minimum spanning trees. *Operations Research*, 18, 1138–1162.
- Helmut, M.** (2001). On the complexity of planning in transportation domains. In Cesta, A. e Barrajo, D. (Eds.), *Sixth European Conference on Planning (ECP-01)*, Toledo, Spain. Springer-Verlag.
- Henzinger, T. A. e Sastry, S.** (Eds.). (1998). *Hybrid systems: Computation and control*. Springer-Verlag, Berlin.
- Herbrand, J.** (1930). *Recherches sur la Théorie de la Démonstration*. Ph.D. thesis, University of Paris.
- Hewitt, C.** (1969). PLANNER: a language for proving theorems in robots. In *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-69)*, pp. 295–301, Washington, DC. IJCAII.
- Hierholzer, C.** (1873). Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6, 30–32.
- Hintikka, J.** (1962). *Knowledge and Belief*. Cornell University Press, Ithaca, New York.
- Hinton, G. E. e Nowlan, S. J.** (1987). How learning can guide evolution. *Complex Systems*, 1 (3), 495–502.
- Hobbs, J. R. e Moore, R. C.** (Eds.). (1985). *Formal Theories of the Commonsense World*. Ablex, Norwood, New Jersey.
- Hoffmann, J.** (2000). A heuristic for domain independent planning and its use in an enforced hillclimbing algorithm. In *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems*, pp. 216–227, Charlotte, North Carolina. Springer-Verlag.
- Holland, J. H.** (1975). *Adaption in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan.
- Holland, J. H.** (1995). *Hidden order: How adaptation builds complexity*. Addison-Wesley, Reading, Massachusetts.
- Holldobler, S. e Schneeberger, J.** (1990). Anew deductive approach to planning. *New Generation Computing*, 8 (3), 225–244.
- Hood, A.** (1824). Case 4th—28 July 1824 (Mr. Hood's cases of injuries of the brain). *The Phrenological Journal and Miscellany*, 2, 82–94.
- Hopfield, J. J.** (1982). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences of the United States of America*, 79, 2554–2558.
- Horn, A.** (1951). On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16, 14–21.
- Horowitz, E. e Sahni, S.** (1978). *Fundamentals of computer algorithms*. Computer Science Press, Rockville, Maryland.
- Horvitz, E. J. e Heckerman, D.** (1986). The inconsistent use of measures of certainty in artificial intelligence research. In Kanal, L. N. e Lemmer, J. F. (Eds.), *Uncertainty in Artificial Intelligence*, pp. 137–151. Elsevier/North-Holland, Amsterdam, London, New York.
- Hsu, F.-H.** (1999). IBM's Deep Blue chess grandmaster chips. *IEEE Micro*, 19(2), 70–80.
- Hsu, F.-H., Anantharaman, T. S., Campbell, M. S. e Nowatzky, A.** (1990). A grandmaster chess machine. *Scientific American*, 263(4), 44–50.
- Huffman, D. A.** (1971). Impossible objects as nonsense sentences. In Meltzer, B. e Michie, D. (Eds.), *Machine Intelligence 6*, pp. 295–324. Edinburgh University Press, Edinburgh, Scotland.

- Huhns, M. N. e Singh, M. P. (Eds.). (1998). *Readings in agents*.** Morgan Kaufmann, San Mateo, California.
- Hume, D. (1739). *Trattato sulla natura umana*.**
- Hwang, C. H. e Schubert, L. K. (1993). EL: A formal, yet natural, comprehensive knowledge representation.** In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 676–682, Washington, DC. AAAI Press.
- Jaffar, J. e Lassez, J.-L. (1987). Constraint logic programming.** In *Proceedings of the Fourteenth ACM Conference on Principles of Programming Languages*, pp. 111–119, Munich. Association for Computing Machinery.
- Jaffar, J., Michaylov, S., Stuckey, P. J. e Yap, R. H. C. (1992a). The CLP(R) language and system.** *ACM Transactions on Programming Languages and Systems*, 14(3), 339–395.
- Jaffar, J., Stuckey, P. J., Michaylov, S. e Yap, R. H. C. (1992b). An abstract machine for CLP(R).** *SIGPLAN Notices*, 27(7), 128–139.
- Jaskowski, S. (1934). On the rules of suppositions in formal logic.** *Studia Logica*, 1.
- Jennings, H. S. (1906). *Behavior of the lower organisms*.** Columbia University Press, New York.
- Jeffreys, W. S. (1874). *The Principles of Science*.** Routledge/Thoemmes Press, London.
- Jimenez, P. e Torras, C. (2000). An efficient algorithm for searching implicit AND/OR graphs with cycles.** *Artificial Intelligence*, 124(1), 1–30.
- Johnson, W. W. e Story, W. E. (1879). Notes on the “15” puzzle.** *American Journal of Mathematics*, 2, 397–404.
- Johnson-Laird, P. N. (1988). *The Computer and the Mind: An Introduction to Cognitive Science*.** Harvard University Press, Cambridge, Massachusetts.
- Johnston, M. D. e Adorf, H.-M. (1992). Scheduling with neural networks: The case of the Hubble space telescope.** *Computers & Operations Research*, 19(3–4), 209–240.
- Jones, R., Laird, J. E. e Nielsen, P. E. (1998). Automated intelligent pilots for combat flight simulation.** In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 1047–54, Madison, Wisconsin. AAAI Press.
- Jonsson, A., Morris, P., Muscettola, N., Rajan, K. e Smith, B. (2000). Planning in interplanetary space: Theory and practice.** In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, pp. 177–186, Breckenridge, Colorado. AAAI Press.
- Joslin, D. e Pollack, M. E. (1994). Least-cost flaw repair: A plan refinement strategy for partial-order planning.** In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, p. 1506, Seattle. AAAI Press.
- Jouannaud, J.-P. e Kirchner, C. (1991). Solving equations in abstract algebras: A rule-based survey of unification.** In Lassez, J.-L. e Plotkin, G. (Eds.), *Computational Logic*, pp. 257–321. MIT Press, Cambridge, Massachusetts.
- Juels, A. e Wattneberg, M. (1996). Stochastic hillclimbing as a baseline method for evaluating genetic algorithms.** In Touretzky, D. S., Mozer, M. C. e Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 8, pp. 430–6. MIT Press, Cambridge, Massachusetts.
- Jurafsky, D. e Martin, J. H. (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*.** Prentice-Hall, Upper Saddle River, New Jersey.
- Kaelbling, L. P. e Rosenschein, S. J. (1990). Action and planning in embedded agents.** *Robotics and Autonomous Systems*, 6 (1–2), 35–48.
- Kahneman, D., Slovic, P. e Tversky, A. (Eds.) (1982). *Judgment under Uncertainty: Heuristics and Biases*.** Cambridge University Press, Cambridge, UK.
- Kaindl, H. e Khorsand, A. (1994). Memorybounded bidirectional search.** In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp. 1359–1364, Seattle. AAAI Press.
- Kambhampati, S. (1994). Exploiting causal structure to control retrieval and refitting during plan reuse.** *Computational Intelligence*, 10, 213–244.
- Kambhampati, S., Mali, A. D. e Srivastava, B. (1998). Hybrid planning for partially hierarchical domains.** In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 882–888, Madison, Wisconsin. AAAI Press.
- Kanal, L. N. e Kumar, V. (1988). *Search in Artificial Intelligence*.** Springer-Verlag, Berlin.
- Kaplan, D. e Montague, R. (1960). A paradox regained.** *Notre Dame Journal of Formal Logic*, 1 (3), 79–90.

- Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, 4, 373–395.
- Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. e Thatcher, J. W. (Eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum, New York.
- Kaufmann, M., Manolios, P. e Moore, J. S. (2000). *Computer-Aided Reasoning: An Approach*. Kluwer, Dordrecht, Netherlands.
- Kautz, H. e Selman, B. (1992). Planning as satisfiability. In *ECAI 92: 10th European Conference on Artificial Intelligence Proceedings*, pp. 359–363, Vienna. Wiley.
- Kautz, H. e Selman, B. (1998). BLACKBOX: A new approach to the application of theorem proving to problem solving. Working Notes of the AIPS-98 Workshop on Planning as Combinatorial Search.
- Kautz, H., McAllester, D. A. e Selman, B. (1996). Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 374–384, Cambridge, Massachusetts. Morgan Kaufmann.
- Kaye, R. (2000). Minesweeper is NP-complete! *Mathematical Intelligencer*, 5 (22), 9–15.
- Kern, C. e Greenstreet, M. R. (1999). Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4 (2), 123–193.
- Kirkpatrick, S. e Selman, B. (1994). Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163), 1297–1301.
- Kirkpatrick, S., Gelatt, C. D. e Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220, 671–680.
- Knoblock, C. A. (1990). Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, Vol. 2, pp. 923–928, Boston. MIT Press.
- Knuth, D. E. (1973). *The Art of Computer Programming* (second edition), Vol. 2: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts.
- Knuth, D. E. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, 6 (4), 293–326.
- Knuth, D. E. e Bendix, P. B. (1970). Simple word problems in universal algebras. In Leech, J. (Ed.), *Computational Problems in Abstract Algebra*, pp. 263–267. Pergamon, Oxford, UK.
- Koehler, J., Nebel, B., Hoffman, J. e Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *Proceedings of the Fourth European Conference on Planning*, pp. 273–285, Toulouse, France. Springer-Verlag.
- Koenig, S. e Simmons, R. (1998). Solving robot navigation problems with initial pose uncertainty using real-time heuristic search. In *aips98*. AAAI Press, Menlo Park, California.
- Kohn, W. (1991). Declarative control architecture. *Communications of the Association for Computing Machinery*, 34(8), 65–79.
- Kolodner, J. (1983). Reconstructive memory: A computer model. *Cognitive Science*, 7, 281–328.
- Kondrak, G. e van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89, 365–387.
- Konolige, K. (1982). A first order formalization of knowledge and action for a multi-agent planning system. In Hayes, J. E., Michie, D. e Pao, Y.-H. (Eds.), *Machine Intelligence 10*. Ellis Horwood, Chichester, England.
- Korf, R. E. (1985a). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1), 97–109.
- Korf, R. E. (1985b). Iterative-deepening A\*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 1034–1036, Los Angeles. Morgan Kaufmann.
- Korf, R. E. (1987). Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1), 65–88.
- Korf, R. E. (1988). Optimal path finding algorithms. In Kanal, L. N. e Kumar, V. (Eds.), *Search in Artificial Intelligence*, chap. 7, pp. 223–267. Springer-Verlag, Berlin.
- Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, 42(3), 189–212.
- Korf, R. E. (1991). Best-first search with limited memory. UCLA Computer Science Annual.
- Korf, R. E. (1993). Linear-space best-first search. *Artificial Intelligence*, 62(1), 41–78.

- Korf, R. E. (1995). Space-efficient search algorithms. *ACM Computing Surveys*, 27(3), 337–339.
- Korf, R. E. e Chickering, D. M. (1996). Best-first minimax search. *Artificial Intelligence*, 84(1–2), 299–337.
- Korf, R. E. e Zhang, W. (2000). Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pp. 910–916, Cambridge, Massachusetts. MIT Press.
- Kotok, A. (1962). A chess playing program for the IBM 7090. Ai project memo 41, MIT Computation Center, Cambridge, Massachusetts.
- Koutsoupias, E. e Papadimitriou, C. H. (1992). On the greedy algorithm for satisfiability. *Information Processing Letters*, 43(1), 53–55.
- Kowalski, R. (1974). Predicate logic as a programming language. In *Proceedings of the IFIP-74 Congress*, pp. 569–574. Elsevier/North-Holland.
- Kowalski, R. (1979a). Algorithm = logic + control. *Communications of the Association for Computing Machinery*, 22, 424–436.
- Kowalski, R. (1979b). *Logic for Problem Solving*. Elsevier/North-Holland, Amsterdam, London, New York.
- Kowalski, R. (1988). The early years of logic programming. *Communications of the Association for Computing Machinery*, 31, 38–43.
- Kowalski, R. e Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence*, 2 (3–4), 227–260.
- Kowalski, R. e Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, 4 (1), 67–95.
- Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts.
- Koza, J. R., Bennett, F. H., Andre, D. e Keane, M. A. (1999). *Genetic Programming III: Darwinian invention and problem solving*. Morgan Kaufmann, San Mateo, California.
- Kraus, S., Ephrati, E. e Lehmann, D. (1991). Negotiation in a non-cooperative environment. *Journal of Experimental and Theoretical Artificial Intelligence*, 3 (4), 255–281.
- Kripke, S. A. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16, 83–94.
- Kumar, P. R. e Varaiya, P. (1986). *Stochastic systems: Estimation, identification, and adaptive control*. Prentice-Hall, Upper Saddle River, New Jersey.
- Kumar, V. (1992). Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1), 32–44.
- Kumar, V. e Kanal, L. N. (1983). A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, 21, 179–198.
- Kumar, V., Nau, D. S. e Kanal, L. N. (1988). A general branch-and-bound formulation for AND/OR graph and game tree search. In Kanal, L. N. e Kumar, V. (Eds.), *Search in Artificial Intelligence*, chap. 3, pp. 91–130. Springer-Verlag, Berlin.
- Kuper, G. M. e Vardi, M. Y. (1993). On the complexity of queries in the logical data model. *Theoretical Computer Science*, 116(1), 33–57.
- Kurzweil, R. (1990). *The Age of Intelligent Machines*. MIT Press, Cambridge, Massachusetts.
- La Mettrie, J. O. (1748). *L'homme machine*. E. Luzac, Leyde, France.
- Ladkin, P. (1986a). Primitives and units for time specification. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Vol. 1, pp. 354–359, Philadelphia. Morgan Kaufmann.
- Ladkin, P. (1986b). Time representation: a taxonomy of interval relations. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Vol. 1, pp. 360–366, Philadelphia. Morgan Kaufmann.
- Laird, J. E., Rosenbloom, P. S. e Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1, 11–46.
- Laird, J. E., Newell, A. e Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1–64.
- Lakoff, G. (1987). *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. University of Chicago Press, Chicago.
- Langton, C. (Ed.). (1995). *Artificial life*. MIT Press, Cambridge, Massachusetts.
- Laplace, P. (1816). *Essai philosophique sur les probabilités* (3rd edition). Courcier Imprimeur, Paris.
- Larrañaga, P., Kuijpers, C., Murga, R., Inza, I. e Dizdarevic, S. (1999). Genetic algorithms for

- the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13, 129–170.
- Lawler, E. L.** (1985). *The traveling salesman problem: A guided tour of combinatorial optimization*. Wiley, New York.
- Lawler, E. L.**, Lenstra, J. K., Kan, A. e Shmoys, D. B. (1992). *The Travelling Salesman Problem*. Wiley Interscience.
- Lefkowitz, D.** (1960). A strategic pattern recognition program for the game Go. Technical note 60-243, Wright Air Development Division, University of Pennsylvania, Moore School of Electrical Engineering.
- Lenat, D. B.** (1995). Cy: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11).
- Lenat, D. B.** e Guha, R. V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*. Addison-Wesley, Reading, Massachusetts.
- Leonard, H. S.** e Goodman, N. (1940). The calculus of individuals and its uses. *Journal of Symbolic Logic*, 5 (2), 45–55.
- Leśniewski, S.** (1916). Podstawy ogólnej teorii mnogości. Moscow.
- Letz, R.**, Schumann, J., Bayerl, S. e Bibel, W. (1992). SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8 (2), 183–212.
- Levesque, H. J.** e Brachman, R. J. (1987). Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3 (2), 78–93.
- Levesque, H. J.**, Reiter, R., Lespérance, Y., Lin, F. e Scherl, R. (1997a). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31, 59–84.
- Levitt, G. M.** (2000). *The Turk, Chess Automaton*. McFarland and Company.
- Lewis, D. K.** (1966). An argument for the identity theory. *The Journal of Philosophy*, 63(1), 17–25.
- Lewis, D. K.** (1980). Mad pain and Martian pain. In Block, N. (Ed.), *Readings in Philosophy of Psychology*, Vol. 1, pp. 216–222. Harvard University Press, Cambridge, Massachussets.
- Li, C. M.** e Anbulagan (1997). Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pp. 366–371, Nagoya, Japan. Morgan Kaufmann.
- Lifschitz, V.** (1986). On the semantics of STRIPS. In Georgeff, M. P. e Lansky, A. L. (Eds.), *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pp. 1–9, Timberline, Oregon. Morgan Kaufmann.
- Lifschitz, V.** (2001). Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2), 39–54.
- Lighthill, J.** (1973). Artificial intelligence: A general survey. In Lighthill, J., Sutherland, N. S., Needham, R. M., Longuet-Higgins, H. C. e Michie, D. (Eds.), *Artificial Intelligence: A Paper Symposium*. Science Research Council of Great Britain, London.
- Lin, F.** e Reiter, R. (1997). How to progress a database. *Artificial Intelligence*, 92(1–2), 131–167.
- Lin, S.** (1965). Computer solutions of the travelling salesman problem. *Bell Systems Technical Journal*, 44(10), 2245–2269.
- Lin, S.** e Kernighan, B. W. (1973). An effective heuristic algorithm for the travelling-salesman problem. *Operations Research*, 21(2), 498–516.
- Linden, T. A.** (1991). Representing software designs as partially developed plans. In Lowry, M. R. e Mc-Cartney, R. D. (Eds.), *Automating Software Design*, pp. 603–625. MIT Press, Cambridge, Massachusetts.
- Lindsay, R. K.**, Buchanan, B. G., Feigenbaum, E. A. e Lederman, J. (1980). *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project*. McGraw-Hill, New York.
- Littman, M. L.**, Keim, G. A. e Shazeer, N. M. (1999). Solving crosswords with PROVERB. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pp. 914–915, Orlando, Florida. AAAI Press.
- Lloyd, J. W.** (1987). *Foundations of Logic Programming*. Springer-Verlag, Berlin.
- Locke, J.** (1690). *Saggio sull'intelletto umano*.
- Lohn, J. D.**, Kraus, W. F. e Colombano, S. P. (2001). Evolutionary optimization of yagi-uda antennas. In *Proceedings of the Fourth International Conference on Evolvable Systems*, pp. 236–243.
- Loveland, D.** (1968). Mechanical theorem proving by model elimination. *Journal of the Association for Computing Machinery*, 15(2), 236–251.

- Loveland, D.** (1970). A linear format for resolution. In *Proceedings of the IRIA Symposium on Automatic Demonstration*, pp. 147–162, Berlin. Springer-Verlag.
- Loveland, D.** (1984). Automated theorem-proving: A quarter-century review. *Contemporary Mathematics*, 29, 1–45.
- Löwenheim, L.** (1915). Über möglichkeiten im Relativkalkül. *Mathematische Annalen*, 76, 447–470.
- Lowerre, B. T.** (1976). *The HARPY Speech Recognition System*. Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania.
- Lowry, M. R. e McCartney, R. D.** (1991). *Automating Software Design*. MIT Press, Cambridge, Massachusetts.
- Loyd, S.** (1959). *Mathematical Puzzles of Sam Loyd: Selected and Edited by Martin Gardner*. Dover, New York.
- Luce, D. R. e Raiffa, H.** (1957). *Games and Decisions*. Wiley, New York.
- Luger, G. F. (Ed.)**. (1995). *Computation and intelligence: Collected readings*. AAAI Press, Menlo Park, California.
- Mackworth, A. K.** (1977). Consistency in networks of relations. *Artificial Intelligence*, 8 (1), 99–118.
- Mackworth, A. K.** (1992). Constraint satisfaction. In Shapiro, S. (Ed.), *Encyclopedia of Artificial Intelligence* (2nd edition), Vol. 1, pp. 285–293. Wiley, New York.
- Mahanti, A. e Daniels, C. J.** (1993). A SIMD approach to parallel heuristic search. *Artificial Intelligence*, 60(2), 243–282.
- Majercik, S. M. e Littman, M. L.** (1999). Planning under uncertainty via stochastic satisfiability. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pp. 549–556.
- Manna, Z. e Waldinger, R.** (1971). Toward automatic program synthesis. *Communications of the Association for Computing Machinery*, 14(3), 151–165.
- Manna, Z. e Waldinger, R.** (1985). *The Logical Basis for Computer Programming: Volume 1: Deductive Reasoning*. Addison-Wesley, Reading, Massachusetts.
- Manna, Z. e Waldinger, R.** (1986). Special relations in automated deduction. *Journal of the Association for Computing Machinery*, 33(1), 1–59.
- Manna, Z. e Waldinger, R.** (1992). Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8), 674–704.
- Marriott, K. e Stuckey, P. J.** (1998). *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachusetts.
- Martelli, A. e Montanari, U.** (1976). Unification in linear time and space: A structured presentation. Internal report B 76-16, Istituto di Elaborazione della Informazione, Pisa, Italy.
- Martelli, A. e Montanari, U.** (1978). Optimizing decision trees through heuristically guided search. *Communications of the Association for Computing Machinery*, 21, 1025–1039.
- Martin, P. e Shmoys, D. B.** (1996). A new approach to computing optimal schedules for the jobshop scheduling problem. In *Proceedings of the 5th International IPCO Conference*, pp. 389–403. Springer-Verlag.
- Maslov, S. Y.** (1964). An inverse method for establishing deducibility in classical predicate calculus. *Doklady Akademii nauk SSSR*, 159, 17–20.
- Maslov, S. Y.** (1967). An inverse method for establishing deducibility of nonprenex formulas of the predicate calculus. *Doklady Akademii nauk SSSR*, 172, 22–25.
- Mason, M.** (1993). Kicking the sensing habit. *AI Magazine*, 14(1), 58–59.
- Mates, B.** (1953). *Stoic Logic*. University of California Press, Berkeley and Los Angeles.
- McAllester, D. A.** (1980). An outlook on truth maintenance. Ai memo 551, MIT AI Laboratory, Cambridge, Massachusetts.
- McAllester, D. A.** (1988). Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3), 287–310.
- McAllester, D. A.** (1998). What is the most pressing issue facing ai and the aaai today? Candidate statement, election for Councilor of the American Association for Artificial Intelligence.
- McAllester, D. A. e Rosenblitt, D.** (1991). Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, Vol. 2, pp. 634–639, Anaheim, California. AAAI Press.

- McCarthy, J. (1958). Programs with common sense. In *Proceedings of the Symposium on Mechanisation of Thought Processes*, Vol. 1, pp. 77–84, London. Her Majesty's Stationery Office.
- McCarthy, J. (1963). Situations, actions, and causal laws. Memo 2, Stanford University Artificial Intelligence Project, Stanford, California.
- McCarthy, J. (1968). Programs with common sense. In Minsky, M. L. (Ed.), *Semantic Information Processing*, pp. 403–418. MIT Press, Cambridge, Massachusetts.
- McCarthy, J. (1980). Circumscription: A form of non-monotonic reasoning. *Artificial Intelligence*, 13(1–2), 27–39.
- McCarthy, J. e Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., Michie, D. e Swann, M. (Eds.), *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press, Edinburgh, Scotland.
- McCulloch, W. S. e Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5, 115–137.
- McCune, W. (1992). Automated discovery of new axiomatizations of the left group and right group calculi. *Journal of Automated Reasoning*, 9(1), 1–24.
- McDermott, D. (1978a). Planning and acting. *Cognitive Science*, 2 (2), 71–109.
- McDermott, D. (1978b). Tarskian semantics, or, no notation without denotation!. *Cognitive Science*, 2 (3).
- McDermott, D. (1987). A critique of pure reason. *Computational Intelligence*, 3 (3), 151–237.
- McDermott, D. (1996). A heuristic estimator for means-ends analysis in planning. In *Proceedings of the Third International Conference on AI Planning Systems*, pp. 142–149, Edinburgh, Scotland. AAAI Press.
- McDermott, D. e Doyle, J. (1980). Non-monotonic logic: i. *Artificial Intelligence*, 13(1–2), 41–72.
- McDermott, J. (1982). R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1), 39–88.
- McGregor, J. J. (1979). Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3), 229–250.
- McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer, Dordrecht, Netherlands.
- Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A. e Teller, E. (1953). Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21, 1087–1091.
- Michie, D. (1966). Game-playing and game-learning automata. In Fox, L. (Ed.), *Advances in Programming and Non-Numerical Computation*, pp. 183–200. Pergamon, Oxford, UK.
- Michie, D. (1972). Machine intelligence at Edinburgh. *Management Informatics*, 2 (1), 7–12.
- Michie, D. (1974). Machine intelligence at Edinburgh. In *On Intelligence*, pp. 143–155. Edinburgh University Press.
- Michie, D. e Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E. e Michie, D. (Eds.), *Machine Intelligence 2*, pp. 125–133. Elsevier/North-Holland, Amsterdam, London, New York.
- Minsky, M. L. (1975). A framework for representing knowledge. In Winston, P. H. (Ed.), *The Psychology of Computer Vision*, pp. 211–277. McGraw-Hill, New York. Originally an MIT AI Laboratory memo; the 1975 version is abridged, but is the most widely cited.
- Minsky, M. L. (Ed.), *Semantic Information Processing*, pp. 271–353. MIT Press, Cambridge, Massachusetts.
- Minsky, M. L. e Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry* (first edition). MIT Press, Cambridge, Massachusetts.
- Minton, S., Johnston, M. D., Philips, A. B. e Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1–3), 161–205.
- Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, Massachusetts.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York.
- Mohr, R. e Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28(2), 225–233.
- Montague, R. (1973). The proper treatment of quantification in ordinary English. In Hintikka, K. J. J., Moravcsik, J. M. E. e Suppes, P. (Eds.), *Approaches to Natural Language*. D. Reidel, Dordrecht, Netherlands.

- Montanari, U.** (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7 (2), 95–132.
- Moore, E. F.** (1959). The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching, Part II*, pp. 285–292. Harvard University Press, Cambridge, Massachusetts.
- Moore, J. S. e Newell, A.** (1973). How can Merlin understand? In Gregg, L. (Ed.), *Knowledge and Cognition*. Lawrence Erlbaum Associates, Potomac, Maryland.
- Moore, R. C.** (1980). Reasoning about knowledge and action. Artificial intelligence center technical note 191, SRI International, Menlo Park, California.
- Moore, R. C.** (1985). A formal theory of knowledge and action. In Hobbs, J. R. e Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, pp. 319–358. Ablex, Norwood, New Jersey.
- Morgenstern, L.** (1987). Knowledge preconditions for actions and plans. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 867–874, Milan. Morgan Kaufmann.
- Morgenstern, L.** (1998). Inheritance comes of age: Applying nonmonotonic techniques to problems in industry. *Artificial Intelligence*, 103, 237–271.
- Morrison, P. e Morrison, E.** (Eds.). (1961). *Charles Babbage and His Calculating Engines: Selected Writings by Charles Babbage and Others*. Dover, New York.
- Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. e Malik, S.** (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 530–535. ACM Press.
- Mostow, J. e Priedtis, A. E.** (1989). Discovering admissible heuristics by abstracting and optimizing: A transformational approach. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Vol. 1, pp. 701–707, Detroit. Morgan Kaufmann.
- Mousouris, J., Holloway, J. e Greenblatt, R. D.** (1979). CHEOPS: A chess-oriented processing system. In Hayes, J. E., Michie, D. e Mikulich, L. I. (Eds.), *Machine Intelligence 9*, pp. 351–360. Ellis Horwood, Chichester, England.
- Müller, M.** (2002). Computer Go. *Artificial Intelligence*, 134(1–2), 145–179.
- Muscettola, N., Nayak, P., Pell, B. e Williams, B.** (1998). Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103, 5–48.
- Nagel, T.** (1974). What is it like to be a bat?. *Philosophical Review*, 83, 435–450.
- Nalwa, V. S.** (1993). *A Guided Tour of Computer Vision*. Addison-Wesley, Reading, Massachusetts.
- Nau, D. S.** (1980). Pathology on game trees: A summary of results. In *Proceedings of the First Annual National Conference on Artificial Intelligence (AAAI-80)*, pp. 102–104, Stanford, California. AAAI.
- Nau, D. S.** (1983). Pathology on game trees revisited, and an alternative to minimaxing. *Artificial Intelligence*, 21(1–2), 221–244.
- Nau, D. S., Kumar, V. e Kanal, L. N.** (1984). General branch and bound, and its relation to A\* and AO\*. *Artificial Intelligence*, 23, 29–58.
- Nayak, P. e Williams, B.** (1997). Fast context switching in real-time propositional reasoning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pp. 50–56, Providence, Rhode Island. AAAI Press.
- Nebel, B.** (2000). On the compilability and expressive power of propositional planning formalisms. *Journal of AI Research*, 12, 271–315.
- Nelson, G. e Oppen, D. C.** (1979). Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1 (2), 245–257.
- Netto, E.** (1901). *Lehrbuch der Combinatorik*. B. G. Teubner, Leipzig.
- Nevins, A. J.** (1975). Plane geometry theorem proving using forward chaining. *Artificial Intelligence*, 6 (1), 1–23.
- Newell, A.** (1982). The knowledge level. *Artificial Intelligence*, 18(1), 82–127.
- Newell, A.** (1990). *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts.
- Newell, A. e Simon, H. A.** (1972). *Human Problem Solving*. Prentice-Hall, Upper Saddle River, New Jersey.
- Newell, A. e Simon, H. A.** (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the Association for Computing Machinery*, 19, 113–126.

- Newell, A. e Ernst, G. (1965). The search for generality. In Kalenich, W. A. (Ed.), *Information Processing 1965: Proceedings of IFIP Congress 1965*, Vol. 1, pp. 17–24, Chicago. Spartan.
- Newell, A. e Simon, H. A. (1961). GPS, a program that simulates human thought. In Billing, H. (Ed.), *Lernende Automaten*, pp. 109–124, R. Oldenbourg, Munich.
- Newell, A., Shaw, J. C. e Simon, H. A. (1957). Empirical explorations with the logic theory machine. *Proceedings of the Western Joint Computer Conference*, 15, 218–239. Reprinted in Feigenbaum and Feldman (1963).
- Newell, A., Shaw, J. C. e Simon, H. A. (1958). Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, 4 (2), 320–335.
- Newton, I. (1664–1671). Methodus fluxionum et serierum infinitarum. Unpublished notes.
- Nguyen, X. e Kambhampati, S. (2001). Reviving partial order planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pp. 459–466, Seattle. Morgan Kaufmann.
- Nguyen, X., Kambhampati, S. e Nigenda, R. S. (2001). Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. Tech. rep., Computer Science and Engineering Department, Arizona State University.
- Niemelä, I., Simons, P. e Syrjänen, T. (2000). Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*.
- Nilsson, N. J. (1991). Logic and artificial intelligence. *Artificial Intelligence*, 47(1–3), 31–56.
- Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York.
- Nilsson, N. J. (1984). Shakey the robot. Technical note 323, SRI International, Menlo Park, California.
- Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, San Mateo, California.
- Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, California.
- Nussbaum, M. C. (1978). *Aristotle's De Motu Animalium*. Princeton University Press, Princeton, New Jersey.
- O'Reilly, U.-M. e Oppacher, F. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Davidor, Y., Schwefel, H.-P. e Manner, R. (Eds.), *Proceedings of the Third Conference on Parallel Problem Solving from Nature*, pp. 397–406, Jerusalem, Israel. Springer-Verlag.
- Ogawa, S., Lee, T.-M., Kay, A. R. e Tank, D. W. (1990). Brain magnetic resonance imaging with contrast dependent on blood oxygenation. *Proceedings of the National Academy of Sciences of the United States of America*, 87, 9868–9872.
- Olawsky, D. e Gini, M. (1990). Deferred planning and sensor use. In Sycara, K. P. (Ed.), *Proceedings, DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, San Diego, California. Defense Advanced Research Projects Agency (DARPA), Morgan Kaufmann.
- Palay, A. J. (1985). *Searching with Probabilities*. Pitman, London.
- Papadimitriou, C. H. (1994). *Computational Complexity*. Addison Wesley.
- Papadimitriou, C. H. e Yannakakis, M. (1991). Shortest paths without a map. *Theoretical Computer Science*, 84(1), 127–150.
- Paterson, M. S. e Wegman, M. N. (1978). Linear unification. *Journal of Computer and System Sciences*, 16, 158–167.
- Patrick, B. G., Almulla, M. e Newborn, M. M. (1992). An upper bound on the time complexity of iterative-deepening-A\*. *Annals of Mathematics and Artificial Intelligence*, 5 (2–4), 265–278.
- Peano, G. (1889). *Arithmetices principia, nova methodo exposita*. Fratres Bocca, Turin.
- Pearl, J. (1982a). Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-82)*, pp. 133–136, Pittsburgh, Pennsylvania. Morgan Kaufmann.
- Pearl, J. (1982b). The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the Association for Computing Machinery*, 25(8), 559–564.

- Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, Massachusetts.
- Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, California.
- Pearson, J. e Jeavons, P. (1997). A survey of tractable constraint satisfaction problems. Technical report CSD-TR-97-15, Royal Holloway College, U. of London.
- Pednault, E. P. D. (1986). Formulating multiagent, dynamic-world problems in the classical planning framework. In Georgeff, M. P. e Lansky, A. L. (Eds.), *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pp. 47–82, Timberline, Oregon. Morgan Kaufmann.
- Peirce, C. S. (1870). Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic. *Memoirs of the American Academy of Arts and Sciences*, 9, 317–378.
- Peirce, C. S. (1883). A theory of probable inference. Note B. The logic of relatives. In *Studies in Logic by Members of the Johns Hopkins University*, pp. 187–203, Boston.
- Peirce, C. S. (1909). Existential graphs. Unpublished manuscript; reprinted in (Buchler 1955).
- Pelikan, M., Goldberg, D. E. e Cantu-Paz, E. (1999). BOA: The Bayesian optimization algorithm. In *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 525–532, Orlando, Florida. Morgan Kaufmann.
- Pemberton, J. C. e Korf, R. E. (1992). Incremental planning on graphs with cycles. In Hendler, J. (Ed.), *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, pp. 525–532, College Park, Maryland. Morgan Kaufmann.
- Penberthy, J. S. e Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92*, pp. 103–114. Morgan Kaufmann.
- Peot, M. e Smith, D. E. (1992). Conditional nonlinear planning. In Hendler, J. (Ed.), *Proceedings of the First International Conference on AI Planning Systems*, pp. 189–197, College Park, Maryland. Morgan Kaufmann.
- Plaat, A., Schaeffer, J., Pijls, W. e de Bruin, A. (1996). Best-first fixed-depth minimax algorithms. *Artificial Intelligence Journal*, 87(1–2), 255–293.
- Place, U. T. (1956). Is consciousness a brain process? *British Journal of Psychology*, 47, 44–50.
- Plotkin, G. (1972). Building-in equational theories. In Meltzer, B. e Michie, D. (Eds.), *Machine Intelligence 7*, pp. 73–90. Edinburgh University Press, Edinburgh, Scotland.
- Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pp. 46–57, Providence, Rhode Island. IEEE, IEEE Computer Society Press.
- Pohl, I. (1969). Bi-directional and heuristic search in path problems. Tech. rep. 104, SLAC (Stanford Linear Accelerator Center, Stanford, California).
- Pohl, I. (1970). First results on the effect of error in heuristic search. In Meltzer, B. e Michie, D. (Eds.), *Machine Intelligence 5*, pp. 219–236. Elsevier/North-Holland, Amsterdam, London, New York.
- Pohl, I. (1971). Bi-directional search. In Meltzer, B. e Michie, D. (Eds.), *Machine Intelligence 6*, pp. 127–140. Edinburgh University Press, Edinburgh, Scotland.
- Pohl, I. (1977). Practical and theoretical considerations in heuristic search algorithms. In Elcock, E. W. e Michie, D. (Eds.), *Machine Intelligence 8*, pp. 55–72. Ellis Horwood, Chichester, England.
- Poole, D., Mackworth, A. K. e Goebel, R. (1998). *Computational intelligence: A logical approach*. Oxford University Press, Oxford, UK.
- Post, E. L. (1921). Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43, 163–185.
- Pratt, V. R. (1976). Semantical considerations on Floyd-Hoare logic. In *Proceedings of the 17th IEEE Symposium on the Foundations of Computer Science*, pp. 109–121. IEEE Computer Society Press.
- Prawitz, D. (1960). An improved proof procedure. *Theoria*, 26, 102–139.
- Prawitz, D. (1965). *Natural Deduction: A Proof Theoretical Study*. Almqvist and Wiksell, Stockholm.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T. e Flannery, B. P. (2002). *Numerical Recipes in C++*.

- The Art of Scientific Computing* (Second edition). Cambridge University Press, Cambridge, UK.
- Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine Learning*, 12(1–3), 117–141.
- Prinz, D. G. (1952). Robot chess. *Research*, 5, 261–266.
- Prior, A. N. (1967). *Past, Present, and Future*. Oxford University Press, Oxford, UK.
- Prosser, P. (1993). Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, 9, 268–299.
- Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York.
- Putnam, H. (1960). Minds and machines. In Hook, S. (Ed.), *Dimensions of Mind*, pp. 138–164. Macmillan, London.
- Putnam, H. (1967). The nature of mental states. In Capitan, W. H. e Merrill, D. D. (Eds.), *Art, Mind, and Religion*, pp. 37–48. University of Pittsburgh Press, Pittsburgh.
- Pylyshyn, Z. W. (1974). Minds, machines and phenomenology: Some reflections on Dreyfus' "What Computers Can't Do". *International Journal of Cognitive Psychology*, 3 (1), 57–77.
- Pylyshyn, Z. W. (1984). *Computation and Cognition: Toward a Foundation for Cognitive Science*. MIT Press, Cambridge, Massachusetts.
- Quillian, M. R. (1961). A design for an understanding machine. Paper presented at a colloquium: Semantic Problems in Natural Language, King's College, Cambridge, England.
- Quine, W. V. (1953). Two dogmas of empiricism. In *From a Logical Point of View*, pp. 20–46. Harper and Row, New York.
- Quine, W. V. (1960). *Word and Object*. MIT Press, Cambridge, Massachusetts.
- Quine, W. V. (1982). *Methods of Logic* (Fourth edition). Harvard University Press, Cambridge, Massachusetts.
- Rabani, Y., Rabinovich, Y. e Sinclair, A. (1998). A computational view of population genetics. *Random Structures and Algorithms*, 12(4), 313–334.
- Ramakrishnan, R. e Ullman, J. D. (1995). A survey of research in deductive database systems. *Journal of Logic Programming*, 23(2), 125–149.
- Ramsey, F. P. (1931). Truth and probability. In Braithwaite, R. B. (Ed.), *The Foundations of Mathematics and Other Logical Essays*. Harcourt Brace Jovanovich, New York.
- Raphson, J. (1690). *Analysis aequationum universalis*. Apud Abelem Swalle, London.
- Ratner, D. e Warmuth, M. (1986). Finding a shortest solution for the  $n \times n$  extension of the 15-puzzle is intractable. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Vol. 1, pp. 168–172, Philadelphia. Morgan Kaufmann.
- Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. Library translation 1122, Royal Aircraft Establishment.
- Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, Germany.
- Regin, J. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp. 362–367, Seattle. AAAI Press.
- Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13(1–2), 81–132.
- Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V. (Ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 359–380. Academic Press, New York.
- Reiter, R. (2001a). On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic*, 2 (4), 433–457.
- Reiter, R. (2001b). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, Massachusetts.
- Reitman, W. e Wilcox, B. (1979). The structure and performance of the INTERIM.2 Go program. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence (IJCAI-79)*, pp. 711–719, Tokyo. IJCAII.
- Remus, H. (1962). Simulation of a learning machine for playing Go. In *Proceedings IFIP Congress*, pp. 428–432, Amsterdam, London, New York. Elsevier/North-Holland.

- Rescher, N. e Urquhart, A. (1971). *Temporal Logic*. Springer-Verlag, Berlin.
- Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21, 25–34. SIGGRAPH '87 Conference Proceedings.
- Rich, E. e Knight, K. (1991). *Artificial Intelligence* (second edition). McGraw-Hill, New York.
- Rieger, C. (1976). An organization of knowledge for problem solving and language comprehension. *Artificial Intelligence*, 7, 89–127.
- Ringle, M. (1979). *Philosophical Perspectives in Artificial Intelligence*. Humanities Press, Atlantic Highlands, New Jersey.
- Rintanen, J. (1999). Improvements to the evaluation of quantified boolean formulae. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 1192–1197, Stockholm. Morgan Kaufmann.
- Robertson, N. e Seymour, P. D. (1986). Graph minors. ii. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7 (3), 309–322.
- Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, 23–41.
- Rorty, R. (1965). Mind-body identity, privacy, and categories. *Review of Metaphysics*, 19(1), 24–54.
- Rosenblueth, A., Wiener, N. e Bigelow, J. (1943). Behavior, purpose, and teleology. *Philosophy of Science*, 10, 18–24.
- Rosenschein, S. J. (1985). Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3 (4), 345–357.
- Rosenthal, D. M. (Ed.). (1971). *Materialism and the Mind-Body Problem*. Prentice-Hall, Upper Saddle River, New Jersey.
- Ross, S. M. (1988). *A First Course in Probability* (third edition). Macmillan, London.
- Roussel, P. (1975). Prolog: Manual de référence et d'utilisation. Tech. rep., Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Rumelhart, D. E. e McClelland, J. L. (Eds.). (1986). *Parallel Distributed Processing*. MIT Press, Cambridge, Massachusetts.
- Rumelhart, D. E., Hinton, G. E. e Williams, R. J. (1986b). Learning representations by backpropagating errors. *Nature*, 323, 533–536.
- Russell, S. J. (1992). Efficient memory-bounded search methods. In *ECAI 92: 10th European Conference on Artificial Intelligence Proceedings*, pp. 1–5, Vienna. Wiley.
- Russell, S. J. e Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, New Jersey.
- Russell, S. J. e Wefald, E. H. (1989). On optimal game-tree search using rational meta-reasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pp. 334–340, Detroit. Morgan Kaufmann.
- Russell, S. J. e Wefald, E. H. (1991). *Do the Right Thing: Studies in Limited Rationality*. MIT Press, Cambridge, Massachusetts.
- Ryder, J. L. (1971). Heuristic analysis of large trees as generated in the game of Go. Memo AIM-155, Stanford Artificial Intelligence Project, Computer Science Department, Stanford University, Stanford, California.
- Sabin, D. e Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *ECAI 94: 11th European Conference on Artificial Intelligence. Proceedings*, pp. 125–129, Amsterdam. Wiley.
- Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5 (2), 115–135.
- Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75)*, pp. 206–214, Tbilisi, Georgia. IJCAII.
- Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*. Elsevier/North-Holland, Amsterdam, London, New York.
- Sacerdoti, E. D., Fikes, R. E., Reboh, R., Sagalowicz, D., Waldinger, R. e Wilber, B. M. (1976). QLISP—a language for the interactive development of complex systems. In *Proceedings of the AFIPS National Computer Conference*, pp. 349–356.
- Sacks, E. e Joskowicz, L. (1993). Automated modeling and kinematic simulation of mechanisms. *Computer Aided Design*, 25(2), 106–118.
- Sadri, F. e Kowalski, R. (1995). Variants of the event calculus. In *International Conference on Logic Programming*, pp. 67–81.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3 (3), 210–229.

- Samuel, A. L.** (1967). Some studies in machine learning using the game of checkers II—Recent progress. *IBM Journal of Research and Development*, 11(6), 601–617.
- Schaeffer, J.** (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, Berlin.
- Schank, R. C. e Abelson, R. P.** (1977). *Scripts, Plans, Goals, and Understanding*. Lawrence Erlbaum Associates, Potomac, Maryland.
- Schank, R. C. e Riesbeck, C.** (1981). *Inside Computer Understanding: Five Programs Plus Miniatures*. Lawrence Erlbaum Associates, Potomac, Maryland.
- Schmolze, J. G. e Lipkis, T. A.** (1983). Classification in the KL-ONE representation system. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*, pp. 330–332, Karlsruhe, Germany. Morgan Kaufmann.
- Schofield, P. D. A.** (1967). Complete solution of the eight puzzle. In Dale, E. e Michie, D. (Eds.), *Machine Intelligence 2*, pp. 125–133. Elsevier/North-Holland, Amsterdam, London, New York.
- Schöning, T.** (1999). A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40<sup>th</sup> Annual Symposium on Foundations of Computer Science*, pp. 410–414, New York. IEEE Computer Society Press.
- Schoppers, M. J.** (1987). Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 1039–1046, Milan. Morgan Kaufmann.
- Schoppers, M. J.** (1989). In defense of reaction plans as caches. *AI Magazine*, 10(4), 51–60.
- Schröder, E.** (1877). *Der Operationskreis des Logikkalküls*. B. G. Teubner, Leipzig.
- Schwartz, S. P. (Ed.)**. (1977). *Naming, Necessity, and Natural Kinds*. Cornell University Press, Ithaca, New York.
- Scriven, M.** (1953). The mechanical concept of mind. *Mind*, 62, 230–240.
- Searle, J. R.** (1980). Minds, brains, and programs. *Behavioral and Brain Sciences*, 3, 417–457.
- Searle, J. R.** (1984). *Minds, Brains and Science*. Harvard University Press, Cambridge, Massachusetts.
- Searle, J. R.** (1992). *The Rediscovery of the Mind*. MIT Press, Cambridge, Massachusetts.
- Selman, B. e Levesque, H. J.** (1993). The complexity of path-based defeasible inheritance. *Artificial Intelligence*, 62(2), 303–339.
- Selman, B., Kautz, H. e Cohen, B.** (1996). Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 26*, pp. 521–532. American Mathematical Society, Providence, Rhode Island.
- Selman, B., Levesque, H. J. e Mitchell, D.** (1992). A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 440–446, San Jose. AAAI Press.
- Shahookar, K. e Mazumder, P.** (1991). VLSI cell placement techniques. *Computing Surveys*, 23(2), 143–220.
- Shanahan, M.** (1997). *Solving the Frame Problem*. MIT Press, Cambridge, Massachusetts.
- Shanahan, M.** (1999). The event calculus explained. In Wooldridge, M. J. e Veloso, M. (Eds.), *Artificial Intelligence Today*, pp. 409–430. Springer-Verlag, Berlin.
- Shannon, C. E.** (1950). Programming a computer for playing chess. *Philosophical Magazine*, 41(4), 256–275.
- Shapiro, S. C. (Ed.)**. (1992). *Encyclopedia of Artificial Intelligence* (second edition). Wiley, New York.
- Shoham, Y.** (1987). Temporal logics in AI: Semantical and ontological considerations. *Artificial Intelligence*, 33(1), 89–104.
- Shoham, Y.** (1993). Agent-oriented programming. *Artificial Intelligence*, 60(1), 51–92.
- Shoham, Y.** (1994). *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann, San Mateo, California.
- Siekmann, J. e Wrightson, G. (Eds.)**. (1983). *Automation of Reasoning*. Springer-Verlag, Berlin.
- Siklossy, L. e Dreussi, J.** (1973). An efficient robot planner which generates its own procedures. In *Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI-73)*, pp. 423–430, Stanford, California. IJCAII.
- Simon, H. A.** (1947). *Administrative behavior*. Macmillan, New York.

- Simon, H. A.** (1957). *Models of Man: Social and Rational*. John Wiley, New York.
- Simon, H. A.** (1963). Experiments with a heuristic compiler. *Journal of the Association for Computing Machinery*, 10, 493–506.
- Simon, H. A.** (1981). *The Sciences of the Artificial* (second edition). MIT Press, Cambridge, Massachusetts.
- Simon, H. A. e Newell, A.** (1958). Heuristic problem solving: The next advance in operations research. *Operations Research*, 6, 1–10.
- Simon, J. C. e Dubois, O.** (1989). Number of solutions to satisfiability instances—applications to knowledge bases. *Int. J. Pattern Recognition and Artificial Intelligence*, 3, 53–65.
- Skinner, B. F.** (1953). *Science and Human Behavior*. Macmillan, London.
- Skolem, T.** (1920). Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über die dichte Mengen. *Videnskapselskapets skrifter, I. Matematisk-naturvidenskabelig klasse*, 4.
- Skolem, T.** (1928). Über die mathematische Logik. *Norsk matematisk tidsskrift*, 10, 125–142.
- Slagle, J. R.** (1963a). A heuristic program that solves symbolic integration problems in freshman calculus. *Journal of the Association for Computing Machinery*, 10(4).
- Slagle, J. R.** (1963b). Game trees,  $m \& n$  minimaxing e the  $m \& n$  alpha-beta procedure. Artificial intelligence group report 3, University of California, Lawrence Radiation Laboratory, Livermore, California.
- Slagle, J. R. e Dixon, J. K.** (1969). Experiments with some programs that search game trees. *Journal of the Association for Computing Machinery*, 16(2), 189–207.
- Slate, D. J. e Atkin, L. R.** (1977). CHESS 4.5—Northwestern University chess program. In Frey, P. W. (Ed.), *Chess Skill in Man and Machine*, pp. 82–118. Springer-Verlag, Berlin.
- Slater, E.** (1950). Statistics for the chess computer and the factor of mobility. In *Symposium on Information Theory*, pp. 150–152, London. Ministry of Supply.
- Sloman, A.** (1978). *The Computer Revolution in Philosophy*. Harvester Press, Hassocks, Sussex, UK.
- Smith, D. E. e Weld, D. S.** (1998). Conformant Graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, p. 888–896, Madison, Wisconsin. AAAI Press.
- Smith, D. R.** (1990). KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9), 1024–1043.
- Smith, D. R.** (1996). Machine support for software development. In *Proceedings of the 18th International Conference on Software Engineering*, pp. 167–168, Berlin. IEEE Computer Society Press.
- Smith, J. M. e Szathmáry, E.** (1999). *The Origins of Life: From the Birth of Life to the Origin of Language*. Oxford University Press, Oxford, UK.
- Smith, R. C. e Cheeseman, P.** (1986). On the representation and estimation of spatial uncertainty. *International Journal of Robotics Research*, 5 (4), 56–68.
- Smolensky, P.** (1988). On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 2, 1–74.
- Soderland, S. e Weld, D. S.** (1991). Evaluating nonlinear planning. Technical report TR-91-02-03, University of Washington Department of Computer Science and Engineering, Seattle, Washington.
- Susic, R. e Gu, J.** (1994). Efficient local search with conflict minimization: A case study of the n-queens problem. *IEEE Transactions on Knowledge and Data Engineering*, 6 (5), 661–668.
- Sowa, J.** (1999). *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Blackwell, Oxford, UK.
- Stallman, R. M. e Sussman, G. J.** (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9 (2), 135–196.
- Stefik, M.** (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann, San Mateo, California.
- Stickel, M. E.** (1985). Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1 (4), 333–355.
- Stickel, M. E.** (1988). A Prolog Technology Theorem Prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4, 353–380.
- Stiller, L. B.** (1992). KQNKR. *ICCA Journal*, 15(1), 16–18.

- Stillings, N. A., Weisler, S., Feinstein, M. H., Garfield, J. L. e Rissland, E. L. (1995).** *Cognitive Science: An Introduction* (second edition). MIT Press, Cambridge, Massachusetts.
- Stockman, G. (1979).** A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2), 179–196.
- Stone, P. (2000).** *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer*. MIT Press, Cambridge, Massachusetts.
- Strachey, C. (1952).** Logical or non-mathematical programmes. In *Proceedings of the Association for Computing Machinery (ACM)*, pp. 46–49, Toronto, Canada.
- Subramanian, D. (1993).** Artificial intelligence and conceptual design. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pp. 800–809, Chambery, France. Morgan Kaufmann.
- Subramanian, D. e Wang, E. (1994).** Constraintbased kinematic synthesis. In *Proceedings of the International Conference on Qualitative Reasoning*, pp. 228–239. AAAI Press.
- Sussman, G. J. (1975).** *A Computer Model of Skill Acquisition*. Elsevier/North-Holland, Amsterdam, London, New York.
- Sussman, G. J. e Winograd, T. (1970).** MICROPLANNER Reference Manual. Ai memo 203, MIT AI Lab, Cambridge, Massachusetts.
- Sutherland, I. (1963).** Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pp. 329–346. IFIPS.
- Swade, D. D. (1993).** Redeeming Charles Babbage's mechanical computer. *Scientific American*, 268(2), 86–91.
- Swift, T. e Warren, D. S. (1994).** Analysis of SLG-WAM evaluation of definite programs. In *Logic Programming. Proceedings of the 1994 International Symposium*, pp. 219–235, Ithaca, NY. MIT Press.
- Syrjänen, T. (2000).** Lparse 1.0 user's manual. <http://saturn.tcs.hut.fi/Software/smodels>.
- Tait, P. G. (1880).** Note on the theory of the "15 puzzle". *Proceedings of the Royal Society of Edinburgh*, 10, 664–665.
- Tamaki, H. e Sato, T. (1986).** OLD resolution with tabulation. In *Third International Conference on Logic Programming*, pp. 84–98, London. Springer-Verlag.
- Tarjan, R. E. (1983).** *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM (Society for Industrial and Applied Mathematics, Philadelphia).
- Tarski, A. (1935).** Die Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1, 261–405.
- Tarski, A. (1956).** *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*. Oxford University Press, Oxford, UK.
- Tate, A. (1975a).** Interacting goals and their use. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75)*, pp. 215–218, Tbilisi, Georgia. IJCAI.
- Tate, A. (1975b).** *Using Goal Structure to Direct Search in a Problem Solver*. Ph.D. thesis, University of Edinburgh, Edinburgh, Scotland.
- Tate, A. (1977).** Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, pp. 888–893, Cambridge, Massachusetts. IJCAI.
- Tate, A. e Whiter, A. M. (1984).** Planning with multiple resource constraints and an application to a naval planning problem. In *Proceedings of the First Conference on AI Applications*, pp. 410–416, Denver, Colorado.
- Tesauro, G. (1989).** Neurogammon wins computer olympiad. *Neural Computation*, 1 (3), 321–323.
- Tesauro, G. (1992).** Practical issues in temporal difference learning. *Machine Learning*, 8 (3–4), 257–277.
- Tesauro, G. (1995).** Temporal difference learning and TD-Gammon. *Communications of the Association for Computing Machinery*, 38(3), 58–68.
- Thagard, P. (1996).** *Mind: Introduction to Cognitive Science*. MIT Press, Cambridge, Massachusetts.
- Thielscher, M. (1999).** From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2), 277–299.
- Touretzky, D. S. (1986).** *The Mathematics of Inheritance Systems*. Pitman e Morgan Kaufmann, London e San Mateo, California.
- Tsang, E. (1993).** *Foundations of Constraint Satisfaction*. Academic Press, New York.

- Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, 2nd series*, 42, 230–265.
- Turing, A. (1948). Intelligent machinery. Tech. rep., National Physical Laboratory. reprinted in (Ince, 1992).
- Turing, A. (1950). Computing machinery and intelligence. *Mind*, 59, 433–460.
- Turing, A., Strachey, C., Bates, M. A. e Bowden, B. V. (1953). Digital computers applied to games. In Bowden, B. V. (Ed.), *Faster than Thought*, pp. 286–310. Pitman, London.
- Ullman, J. D. (1985). Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3), 289–321.
- Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Maryland.
- Vaessens, R. J. M., Aarts, E. H. I. e Lenstra, J. K. (1996). Job shop scheduling by local search. *INFORMS J. on Computing*, 8, 302–117.
- van Benthem, J. (1983). *The Logic of Time*. D. Reidel, Dordrecht, Netherlands.
- Van Emden, M. H. e Kowalski, R. (1976). The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4), 733–742.
- van Heijenoort, J. (Ed.). (1967). *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, Massachusetts.
- Van Hentenryck, P., Saraswat, V. e Deville, Y. (1998). Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37(1–3), 139–164.
- Van Roy, P. L. (1990). Can logic programming execute as fast as imperative programming? Report UCB/CSD 90/600, Computer Science Division, University of California, Berkeley, California.
- Vere, S. A. (1983). Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 5, 246–267.
- von Neumann, J. e Morgenstern, O. (1944). *Theory of Games and Economic Behavior* (first edition). Princeton University Press, Princeton, New Jersey.
- Vossen, T., Ball, M., Lotem, A. e Nau, D. S. (2001). Applying integer programming to ai planning. *Knowledge Engineering Review*, 16, 85–100.
- Waldinger, R. (1975). Achieving several goals simultaneously. In Elcock, E. W. e Michie, D. (Eds.), *Machine Intelligence 8*, pp. 94–138. Ellis Horwood, Chichester, England.
- Waltz, D. (1975). Understanding line drawings of scenes with shadows. In Winston, P. H. (Ed.), *The Psychology of Computer Vision*. McGraw-Hill, New York.
- Warren, D. H. D. (1974). WARPLAN: A System for Generating Plans. Department of Computational Logic Memo 76, University of Edinburgh, Edinburgh, Scotland.
- Warren, D. H. D. (1976). Generating conditional plans and programs. In *Proceedings of the AISB Summer Conference*, pp. 344–354.
- Warren, D. H. D. (1983). An abstract Prolog instruction set. Technical note 309, SRI International, Menlo Park, California.
- Watson, J. D. e Crick, F. H. C. (1953). A structure for deoxyribose nucleic acid. *Nature*, 171, 737.
- Webber, B. L. e Nilsson, N. J. (Eds.). (1981). *Readings in Artificial Intelligence*. Morgan Kaufmann, San Mateo, California.
- Weidenbach, C. (2001). SPASS: Combining superposition, sorts and splitting. In Robinson, A. and Voronkov, A. (Eds.), *Handbook of Automated Reasoning*. mit, mit-ad.
- Weiss, G. (1999). *Multiagent systems*. MIT Press, Cambridge, Massachusetts.
- Weizenbaum, J. (1976). *Computer Power and Human Reason*. W. H. Freeman, New York.
- Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4), 27–61.
- Weld, D. S. (1999). Recent advances in ai planning. *AI Magazine*, 20(2), 93–122.
- Weld, D. S. e de Kleer, J. (1990). *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann, San Mateo, California.
- Weld, D. S., Anderson, C. R. e Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 897–904, Madison, Wisconsin. AAAI Press.

- Weld, D. S., Anderson, C. R. e Smith, D. E.** (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 897–904, Madison, Wisconsin. AAAI Press.
- Wellman, M. P.** (1995). The economic approach to artificial intelligence. *ACM Computing Surveys*, 27(3), 360–362.
- Whitehead, A. N. e Russell, B.** (1910). *Principia Mathematica*. Cambridge University Press, Cambridge, UK.
- Widrow, B.** (1962). Generalization and information storage in networks of adaline “neurons”. In Yovits, M. C., Jacobi, G. T. e Goldstein, G. D. (Eds.), *Self-Organizing Systems 1962*, pp. 435–461, Chicago, Illinois. Spartan.
- Widrow, B. e Hoff, M. E.** (1960). Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, pp. 96–104, New York.
- Wiener, N.** (1942). The extrapolation, interpolation, and smoothing of stationary time series. Osrd 370, Report to the Services 19, Research Project DIC-6037, MIT, Cambridge, Massachusetts.
- Wiener, N.** (1948). *Cybernetics*. Wiley, New York.
- Wilensky, R.** (1978). *Understanding goal-based stories*. Ph.D. thesis, Yale University, New Haven, Connecticut.
- Wilensky, R.** (1983). *Planning and Understanding*. Addison-Wesley, Reading, Massachusetts.
- Wilkins, D. E.** (1988). *Practical Planning: Extending the AI Planning Paradigm*. Morgan Kaufmann, San Mateo, California.
- Wilkins, D. E.** (1990). Can AI planners solve practical problems?. *Computational Intelligence*, 6 (4), 232–246.
- Wilkins, D. E., Myers, K. L., Lowrance, J. D. e Wesley, L. P.** (1995). Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7 (1), 197–227.
- Wilson, R. A. e Keil, F. C. (Eds.).** (1999). *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press, Cambridge, Massachusetts.
- Winograd, S. e Cowan, J. D.** (1963). *Reliable Computation in the Presence of Noise*. MIT Press, Cambridge, Massachusetts.
- Winograd, T.** (1972). Understanding natural language. *Cognitive Psychology*, 3 (1), 1–191.
- Winston, P. H.** (1970). Learning structural descriptions from examples. Technical report MAC-TR-76, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.
- Winston, P. H.** (1992). *Artificial Intelligence* (Third edition). Addison-Wesley, Reading, Massachusetts.
- Wittgenstein, L.** (1953). *Ricerche filosofiche*.
- Wittgenstein, L.** (1922). *Tractatus Logico-Philosophicus*.
- Wood, M. K. e Dantzig, G. B.** (1949). Programming of interdependent activities. i. general discussion. *Econometrica*, 17, 193–199.
- Woods, W. A.** (1975). What's in a link? Foundations for semantic networks. In Bobrow, D. G. e Collins, A. M. (Eds.), *Representation and Understanding: Studies in Cognitive Science*, pp. 35–82. Academic Press, New York.
- Wooldridge, M. e Rao, A. (Eds.).** (1999). *Foundations of rational agency*. Kluwer, Dordrecht, Netherlands.
- Wos, L. e Robinson, G.** (1968). Paramodulation and set of support. In *Proceedings of the IRIA Symposium on Automatic Demonstration*, pp. 276–310. Springer-Verlag.
- Wos, L. e Winker, S.** (1983). Open questions solved with the assistance of AURA. In Bledsoe, W. W. e Loveland, D. (Eds.), *Automated Theorem Proving: After 25 Years: Proceedings of the Special Session of the 89th Annual Meeting of the American Mathematical Society*, pp. 71–88, Denver, Colorado. American Mathematical Society.
- Wos, L., Carson, D. e Robinson, G.** (1964). The unit preference strategy in theorem proving. In *Proceedings of the Fall Joint Computer Conference*, pp. 615–621.
- Wos, L., Carson, D. e Robinson, G.** (1965). Efficiency and completeness of the set-of-support strategy in theorem proving. *Journal of the Association for Computing Machinery*, 12, 536–541.
- Wos, L., Overbeek, R., Lusk, E. e Boyle, J.** (1992). *Automated Reasoning: Introduction and Applications* (second edition). McGraw-Hill, New York.
- Wos, L., Robinson, G., Carson, D. e Shalla, L.** (1967). The concept of demodulation in theorem proving. *Journal of the Association for Computing Machinery*, 14, 698–704.

- Wright, S. (1931). Evolution in Mendelian populations. *Genetics*, 16, 97–159.
- Wygant, R. M. (1989). CLIPS—a powerful development and delivery expert system tool. *Computers and Industrial Engineering*, 17, 546–549.
- Yang, Q. (1990). Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6, 12–24.
- Yang, Q. (1997). *Intelligent planning: A decomposition and abstraction based approach*. Springer-Verlag, Berlin.
- Yip, K. M.-K. (1991). *KAM: A System for Intelligently Guiding Numerical Experimentation by Computer*. MIT Press, Cambridge, Massachusetts.
- Yob, G. (1975). Hunt the wumpus! *Creative Computing*, Sep/Oct.
- Young, R. M., Pollack, M. E. e Moore, J. D. (1994). Decomposition and causality in partial order planning. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pp.188–193, Chicago.
- Zhou, R. e Hansen, E. (2002). Memory-bounded A\* graph search. In *Proceedings of the 15th International Flairs Conference*.
- Zobrist, A. L. (1970). *Feature Extraction and Representation for Pattern Recognition and the Game of Go*. Ph.D. thesis, University of Wisconsin.
- Zuse, K. (1945). The Plankalkül. Report 175, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Germany.

# Indice analitico

## Simboli

---

↪ (prima di), 494  
 $A \xrightarrow{p} B$  (raggiunge), 494  
 $\lambda$ , espressione, 317  
 $\Rightarrow$  (condizionale), 264  
 $\wedge$  (and), 264  
 $\Leftrightarrow$  (se e solo se), 264  
 $\neg$  (not), 264  
 $\vee$  (or), 264  
 $\Box p$  (sempre nel futuro), 462  
 $\Diamond p$  (in qualche istante nel futuro), 462  
 $\models$  (implicazione), 260  
 $\vdash$  (derivazione), 261  
 $\exists$  (esiste), 320  
 $\forall$  (per ogni), 318

## 0-9

---

3SAT, 182

## A

---

AAAI (American Association for AI), 42  
Aarts, E. H.I., 577  
Aarup, M., 577  
ABC, computer, 21  
Abelson, R.P., 34  
ABO (Asymptotic Bounded Optimality), v. ottimalità limitata, asintotica  
Abramson, B., 117

ABSOLVER, 142  
ABSTRIPS, 577  
AC-3, 190, 203  
AC-4, 203  
Acharya, A., 171  
aciclicità, 296  
Ackley, D.H., 173  
ACT, 365  
Ada, linguaggio di programmazione, 22  
adaline, 29  
Adelson-Velsky, G.M., 241  
ADIACENTI, 190  
ADL (Action Description Language), 481, 519  
Adorf, H.-M., 577  
Advice Taker, 27, 28, 33  
agente, 8, 45, 73  
architettura, 38, 616-618  
aspirapolvere, 47, 76-77  
autonomo, 254  
basato su circuito, 293-297  
basato su conoscenza, 20, 251-255, 307, 614  
basato su modello, 66-68  
basato su obiettivi, 68-69, 73-74  
basato sull'utilità, 69  
capace di apprendere, 70-73  
componenti, 614-616  
di pianificazione continua, 564  
di ripianificazione, 576  
funzione, 46  
guidatore di taxi, 72, 617  
intelligente, 39, 606, 614  
logico, 253, 289-298, 343  
nel mondo del wumpus, 255,

programma, 46, 61, 74  
razionale, 8-9, 46, 48, 75  
razionale limitato, 435  
reattivo, 63-67, 74  
reattivo semplice, 63  
risolutore di problemi, 82-87  
situato, 594  
software, 56

AGENTE-CON-TABELLA, 62  
AGENTE-ONLINE-DFS, 164, 166  
AGENTE-POP-CONTINUO, 569, 579  
AGENTE-REATTIVO-ASPIRAPOLVERE, 63  
AGENTE-REATTIVO-CON-STATO, 67  
AGENTE-RIPIANIFICAZIONE, 560  
AGGIORNA-STATO, 67, 84  
aggregazione, 534  
Agre, P.E., 580  
AISB (Society for Artificial Intelligence and Simulation of Behaviour), 42  
Ait-Kaci, H., 399  
al-Khowarazmi, 13  
Aldiss, B., 608  
Aldous, D., 172  
alfa-beta, v. ricerca, alfa-beta  
algoritmo, 13  
anytime, 617  
Metropolis, 172  
minimax, 214, 551

ALGORITMO-GENETICO, 155  
ALICE, 589  
Allen, B., 577

Allen, J.F., 463, 522  
 Almulla, M., 171  
 Alterman, R., 577  
 Alvey report, 34  
 Amarel, S., 117, 461  
 ambiente, 46, 53-60  
     ad agente singolo, 58  
     artificiale, 54  
     classe di, 60  
     competitivo, 59  
     continuo, 59  
     cooperativo, 59  
     deterministico, 57  
     dinamico, 58  
     discreto, 58  
     episodico, 57  
     generatore di, 60  
     multiagente, 58, 433, 569  
     osservabile, 56  
     proprietà, 56  
     semidinamico, 58  
     sequenziale, 57  
     sequenziale, 57  
     statico, 58  
     stocastico, 57  
     strategico, 57  
 ambiguità, 309, 441  
 Ambros-Ingerson, J., 577  
 ampiezza, ricerca in, 99, 116  
 analisi asintotica, 623-626  
 analisi degli algoritmi, 623  
 analisi di complessità, 100, 625  
 ANALOGY, 28, 29, 43  
 Analytical Engine, 22  
 Anbulagan, 302  
 AND-parallelismo, 373  
 Anderson, A.R., 610  
 Anderson, C.R., 521, 578  
 Anderson, J.A., 41  
 Anderson, J.R., 21, 365  
 Andre, D., 173  
 Anshelevich, V.A., 244  
 antecedente, 264  
 Appel, K., 202  
 apprendimento, 52, 73, 255  
     dama, 27  
     e teoria della computazione, 179  
     elemento di, 71  
     euristiche, 144  
     nel mondo dei blocchi, 28  
     per la ricerca, 138

approccio procedurale, 254  
 approfondimento iterativo,  
     v. ricerca, approfondimento  
     iterativo  
 Apt, K.R., 204  
 Arbuckle, T., 239  
 architettura, 61  
     agente, 38  
     basata su regole, 365  
     cognitiva, 365  
     di controllo in tempo reale, 400  
     di sussunzione, 580  
     parallela, 174  
     per il riconoscimento vocale, 36  
 Arentoft, M.M., 577  
 argomentazione derivante  
     dall'incapacità, 589-590  
 argomentazione derivante  
     dall'informalità, 592-594  
 Aristotele, 7, 10, 11, 16, 74,  
     300, 341, 396, 461, 465  
 arità, 316, 360  
 Armstrong, D.M., 610  
 Arnauld, A., 12  
 Arora, S., 117  
 artificiale  
     inseminazione, 595  
     urea, 595  
     volo, 6  
 ASK, 253, 343, 354  
 aspirapolvere  
     “doppio Murphy”, 550, 555  
     “triplo Murphy”, 553  
 assegnamento(in un CSP), 180  
 asserzioni (in logica), 323  
 assiomi, 325  
     dei nomi unici, 423-424  
     della stanza cinese, 603  
     della teoria degli insiemi, 327  
     delle azioni distinte, 423  
     di effetto, 420  
     di esclusione tra azioni, 513  
     di frame, 421  
     di Peano, 327, 341, 360  
     di possibilità, 420  
     di precondizione, 513  
     di stato successore, 303, 422,  
         463, 512  
     di teoria dei numeri, 327  
     nel calcolo delle situazioni, 420  
     nel mondo del wumpus, 330  
 specifico di un dominio, 409  
 STRIPS, 521  
 utilizzabile (in OTTER), 390  
 assone, 17  
 assunto di STRIPS, 481  
 astrazione, 87  
 Atanasoff, J., 21  
 Atkin, L.R., 118  
 ATMS, v. sistemi di  
     mantenimento della verità,  
     basato sugli assunti  
 attitudini proposizionali, 433  
 attuatori, 46, 54  
 Audi, R., 611  
 AURA, 393, 394, 400  
 automi, 605, 610  
 Auton, L.D., 302  
 autonomia, 51, 478  
 AZIONE, 95, 98  
 azioni  
     applicabili, 480  
     condizionali, 578  
     congiunte, 571  
     di percezione, 546, 557, 577  
     di persistenza, 502  
     MONITORAGGIO DELLE, 559,  
         560  
     PRIMITIVE, 536  
     RAZIONALI, 11, 38  
     RILEVANTI, 488  
     SCHEMA DI, 480  
     SCOMPOSIZIONE, 535, 537-539  
     SCOMPOSIZIONE (RICORSIVA),  
         543  
 AZIONI, 161, 166

**B**

---

*b*\* (fattore di ramificazione), 140  
*B*\*, ricerca, 242  
 Babbage, C., 22, 240  
 Bacchus, F., 186, 203  
 Bachmann, P. G.H., 630  
 backgammon, 227-228, 235  
 backjumping, 193, 204  
 backtracking  
     cronologico, 192  
     dinamico, 204  
     intelligente, 192-194  
 BACKTRACKING-RICORSIVO, 185

- Bacone, F., 11  
 Baker, C.L., 41  
 Baldwin, J.M., 157  
 Ball, M., 521  
 Ballard, B.W., 243  
 Baluja, S., 173  
 Bancilhon, F., 399  
 Barrett, A., 577  
 Bartak, R., 205  
 Bartlett, F., 20  
 Barto, A.G., 174  
 Barwise, J., 305  
 base di Herbrand, 384  
 Bates, M.A., 21, 240  
 Baum, E., 154, 242  
 Bayardo, R.J., 302  
 Bayerl, S., 400  
 Bayes, regola di, 14  
 Bayesiane, reti, 37  
 BDD, 556  
 Beal, D.F., 243  
 beam search stocastica, 152  
 Beckert, B., 401  
 Beeri, C., 204  
 behaviorismo, 19, 24, 75  
 Bell, C., 545, 576  
 Bell, J.L., 342  
 Bellman, R.E., 4, 16, 117, 118,  
     170  
 Belsky, M.S., 239  
 benchmark, 623  
 Bendix, P.B., 400  
 BENINQ, 466  
 Bennett, B., 467  
 Bennett, F.H., 173  
 Bennett, J., 394, 400  
 Berger, H., 18  
 Berlekamp, E.R., 118  
 Berleur, J., 604  
 Berliner, H.J., 242, 243  
 Berners-Lee, T., 462  
 Bernoulli, J., 14  
 Bernstein, A., 239  
 Berry, C., 21  
 Bertoli, P., 577  
 BESM, 240  
 Bezzel, M., 117  
 Bibel, W., 397  
 Bickford, M., 394  
 bicondizionale, 264  
 Bigelow, J., 23  
 Biggs, N.L., 202  
 Biro, J.I., 611  
 Birtwistle, G., 465  
 Bishop, C.M., 172  
 Bishop, R.H., 75  
 Bistarelli, S., 203  
 Bitman, A.R., 241  
 Bitner, J.R., 203  
 BKG (programma per  
     backgammon), 243  
 black box, 179  
 BLACKBOX, 517, 521  
 blind search, v. ricerca, non  
     informata  
 Block, N., 610  
 Blum, A.L., 521  
 BO, v. ottimalità limitata  
 Bobrow, D.G., 28  
 Boddy, M., 578, 617  
 Boden, M.A., 300, 612  
 boid, 573, 580  
 Boneh, D., 154  
 Bonet, B., 522, 578  
 Boole, G., 12-13, 301, 341  
 Booleana, logica, v. logica,  
     proposizionale  
 Boolos, G.S., 401  
 Borgida, A., 448, 466  
 Boutilier, C., 463, 580  
 Bowden, B.V., 21, 240  
 Box, G. E.P., 172  
 Boyan, J.A., 172  
 Boyer, R.S., 394, 400  
 Boyle, J., 401  
 Brachman, R.J., 448, 465, 468  
 Bradtke, S.J., 174  
 Brafman, R.I., 580  
 Brahmagupta, 203  
 Brandeis, L., 606  
 Bratko, I., 171, 400  
 Bratman, M.E., 75, 611  
 Breese, J.S., 75  
 Brelaz, D., 203  
 Brent, R.P., 172  
 Brewka, G., 466  
 Bridge Baron, 238  
 Briggs, R., 461  
 Broadbent, D.E., 20  
 Broca, P., 16  
 Brooks, R.A., 75, 300, 303, 580  
 Brown, J.S., 467  
 Brownston, L., 398  
 Brudno, A.L., 241  
 Brunnstein, K., 604  
 Bryant, R.E., 556, 578  
 Bryson, A.E., 32  
 Buchanan, B.G., 32, 75, 461  
 BUILD, 467  
 Bundy, A., 401  
 Bunt, H.C., 464  
 Buro, M., 235  
 Burstall, R.M., 401, 462  
 Burstein, J., 590  
 Bylander, T., 492, 519
- 
- C**
- C-BURIDAN, 579  
 Cajal, S., 17  
 calcolo dei fluenti, 424, 463  
 calcolo dei predicati, v. logica,  
     del primo ordine  
 calcolo delle situazioni, 418,  
     418-426, 462  
 calcolo proposizionale, v. logica,  
     proposizionale  
 Call, 372  
 Calvanese, D., 466  
 Cambefort, Y., 76  
 Cambridge, 20  
 cammino, 85, 116  
 Campbell, M.S., 234, 242  
 campo minato, 302  
 Cantu-Paz, E., 173  
 caratteristica (di uno stato), 144  
 Carbonell, J.G., 577, 578  
 Cardano, G., 14  
 Carnap, R., 11  
 Carnegie Mellon, Università, 25  
 Carpenter, M., 577  
 Carson, D., 400

- Cartesio, R., 10  
 Cassandra, 579  
 casualità, 46, 66  
 categoria, 410-417  
 Ceri, S., 398  
 cervello  
     e computer, 20  
     e supercervelloni elettronici, 14  
     nella vasca, 598  
     protesi, 632  
     stati, 597  
 CGP, 579  
 CHAFF, 285, 289, 302  
 Chakrabarti, P.P., 171  
 Chandra, A.K., 398  
 Chang, C.-L., 401  
 Chapman, D., 521, 580  
 Charniak, E., 4, 33, 398  
 chatbot, 589  
 Cheeseman, P., 14, 37, 204, 302  
 CHESS 4.5, 118  
 CHESS 4.6, 241  
 Chickering, D.M., 242  
 Chien, S., 577  
 Chierchia, G., 41  
 Chinook, 235, 243  
 Chomsky, N., 21, 24  
 Church, A., 14, 351, 397  
 Churchland, P.M., 610  
 Churchland, P.S., 601, 610, 611  
 cibernetica, 22  
 ciclo aperto, 84  
 Cimatti, A., 522, 578  
 circoscrizione, 455, 461, 466  
     con priorità, 456  
 Clark, Forma Normale di, 451  
 Clark, K.L., 466  
 Clarke, E., 401, 522  
 Clarke, M. R.B., 243  
 CLASSIC, 448, 449  
 classificazione (con logiche descrittive), 448  
 clausole, 275  
     definite, 280, 357-358  
     di Horn, 280, 390  
     unitarie, 275, 388  
     unitarie, preferenza per le, 388  
 CLIPS, 398  
 Clocksin, W.F., 400  
 CLP(R), 399  
 CLP, v. programmazione logica, con vincoli  
 CMU, 39, 234, 242  
 CNLP, 579  
 co-NP-completezza, 269,  
     301-302, 626  
 Cobham, A., 14  
 coda, 96  
     con priorità, 126  
     FIFO, 99  
     LIFO, 102  
 codificare direttamente (open-code), 371  
 coercizione, 546, 577  
 cognitiva  
     architettura, 365  
     psicologia, 20  
     scienza, 6-7, 41  
 Cohen, B., 302  
 Cohen, J., 399  
 Cohen, P.R., 36, 580  
 Cohn, A.G., 467  
 COLLEGAMENTI, 538  
 collegamenti causali, 494  
 collegamenti inversi, 446  
 collegamento procedurale, 442,  
     447  
 Collins, G., 578  
 Colmerauer, A., 342, 399  
 Colombano, S.P., 173  
 commutatività (nei problemi di ricerca), 184  
 competizione, 574  
 compilazione, 372  
 complementari, letterali, 275  
 complessità, 623-626  
     spaziale, 97, 116  
     temporale, 96, 116  
 completamento di Clark, 451,  
     466  
 completamento (di un database), 451  
 completezza  
     dell'inferenza, 299  
     della risoluzione, 383-386  
     di un algoritmo di ricerca, 96,  
         116  
 di una procedura di dimostrazione, 262, 298,  
     376  
 per la refutazione, 276, 383  
 teorema di, 376  
 comportamento emergente, 573  
 COMPOSE, 367  
 composizionalità, 308  
 composizione (delle sostituzioni), 367  
 computabilità, 13  
 concatenazione all'indietro, 281,  
     367-376, 395  
 concatenazione in avanti,  
     280-281, 302, 357-366  
 conclusione (di un'implicazione), 264  
 concorrenza, azioni, 572  
 condivisione delle sottoattività,  
     540  
 Condon, J.H., 242  
 configurazione VLSI, 92, 152  
 confini (nello spazio degli stati), 132  
 congiunti, 264  
 congiuntiva, forma normale,  
     277, 377-379  
 congiunzione (logica), 264  
 connessionismo, 35  
 connettivo logico, 24, 264, 318  
*ConosceCosa*, 436  
 conoscenza  
     acquisizione, 334  
     base di, 253, 299, 342  
     comune, 28  
     e azione, 11, 437  
     effetto sulla, 437  
     ingegneria della, 333-340, 407  
     iniziale, 253, 381  
     linguaggio di rappresentazione,  
         253, 299, 307  
     livello, 254, 299  
     mappe, v. reti Bayesiane  
     precondizioni di, 437  
     pregressa, 52  
     proposizioni di, 295  
     rappresentazione, 6, 24, 28, 34,  
         307-313, 407-474  
     sistemi basati su, 32-34  
*ConosceSe*, 436

*Consecutivi* (relazione tra intervalli), 431

conseguente, 264

consistenza, 448

condizionale, 170

d'arco, 189

delle azioni, 489

di cammino, 203

di un'eustistica, 130

Console, L., 75

conspiracy number search, 242

contenuto allargato, 598

contenuto stretto, 598

continuazioni, 372

controllo di occorrenza, 354, 370

controllore, 74

convenzione, 573

conversione in forma normale, 377-379

Conway, J.H., 118

Cook, P.J., 605

Cook, S.A., 14, 302, 630

cooperazione, 570

coordinamento, 573-574

Copeland, J., 463, 612

Copernico, 605

Cormen, T.H., 630

corpo (di una clausola), 367

correttezza (di un'inferenza), 262

coscienza, 16, 594, 595-600, 604

costante di Skolem, 348, 396

costo di cammino, 85, 116, 181

costo di passo, 86

costo totale, 97

**COSTRUISCI-FORMULA-AZIONE**, 253

**COSTRUISCI-FORMULA-PERCEZIONE**, 253

**COSTRUISCI-INTERROGAZIONE-AZIONE**, 253

Cowan, J.D., 29

CP-AGENTE-WUMPUS, 292, 297

CP-CA-IMPLICA?, 282

CP-RISOLUZIONE, 279

CP-VERO?, 304

Craik, K.J., 20

Crawford, J.M., 302

credenze, 433-436

aggiornamento delle, 457

reti di, v. reti Bayesiane

revisione delle, 457

stati-, 553

Crick, F. H.C., 157

criptazione a chiave pubblica (RSA), 394

criptoaritmetica, 183

critico, elemento di

apprendimento, 71

Crocker, S.D., 241

Crockett, L., 463

Cross, S.E., 39

crossover, 154

cruciverba, 39, 58

CSP booleani, 182

CSP, v. soddisfacimento di vincoli, problema di

Ctesibio, 22

cubo di Rubik, 117, 142

Culberson, J., 171

Cullingford, R.E., 34

culto del computazionalismo, 588

Currie, K.W., 577

CYC, 461-463

CYPRESS, 579

## D

da Vinci, L., 11

Dahl, O.-J., 465

dama, 27, 76, 234-235, 243

Daniels, C.J., 175

Dantzig, G.B., 172

DARKTHOUGHT, 242

Darlington, J., 401

DARPA, 39

DART, 39

Dartmouth, workshop presso, 25

Darwin, C., 157, 605

Dasgupta, P., 174

data mining, 37

database deduttivi, 395

database di pattern, 143, 171

disgiunti, 144

Datalog, 358, 395, 398

Daun, B., 577

Davidson, D., 463

Davis, E., 465, 467, 468

Davis, G., 577

Davis, M., 284, 301, 383, 396

Davis-Putnam, algoritmo di, 284

de Kleer, J., 204, 398, 467

De Morgan, A., 341

De Morgan, regole di, 323

Deacon, T.W., 35

Deale, M., 577

Dean, M.E., 394

Dean, T., 576

debugging, 334

Dechter, R., 170, 203-205

decisione, problema della, 13

DECISIONE-MINIMAX, 215, 217

Dedekind, R., 341

deduzione naturale, 396

deduzione, v. inferenza

Deep Blue, 38, 233, 242

Deep Space One, 75, 517, 577

Deep Thought, 233, 242

default, logica di, 456, 461, 466

default, ragionamento con informazione di, 450-457

default, valori di, 447

definizioni (logica), 325

delay, v. linea di ritardo

demodulatore, 390, 403

demodulazione, 386, 400

*Den* (denotazione), 434

DENDRAL, 32, 461

dendrite, 17

Deng, X., 173

Dennett, D.C., 592, 610

Deo, N., 118

Descotte, Y., 576

Deville, Y., 204

Deviser, 576

Dewey, sistema decimale, 411

diacronico, 330

diagnosi medica, 33, 39, 332

diagramma binario di

decisione, 556

diametro di un grafo, 104

dichiarativo, approccio, 254

difetto (di un piano), 567

DiGioia, A.M., 39

- Digital Equipment Corporation (DEC), 34, 365  
 Dijkstra, E.W., 118, 170, 610  
 Dill, D.L., 394  
 dimostratore di teoremi, 390-395  
     come aiutante, 393  
     di Boyer-Moore, 394, 400  
 dimostrazione, 273  
     di teoremi, 519  
     procedura, 13  
 dimostrazione di teoremi matematici, 30, 400  
 diofantine, equazioni, 202  
 Diplomacy, gioco da tavolo, 216  
 discesa di gradiente, 151  
 discretizzazione, 158  
 disgiunte, categorie, 412  
 disgiunti, 264  
 disgiunzione, 264  
 distanza Manhattan, 140  
 distanze in linea d'aria, 127  
 distribuzione a coda pesante, 172  
 divisione dei simboli, 515  
 Dix, J., 466  
 Dixon, J.K., 241  
 Dizdarevic, S., 173  
 DLV, 466  
 Do, M.B., 576  
 dominanza (tra euristiche), 140  
 dominio  
     dei circuiti elettronici, 335-340  
     di parentela, 325-326  
     elemento di un, 313  
     finito, 375  
     in logica del primo ordine, 313  
     in un CSP, 180  
     indipendenza, 478  
     nella rappresentazione della conoscenza, 323  
 DOMINIO, 190  
 Dono dei Magi, il problema del, 542  
 Donskoy, M.V., 241  
 Doran, J., 170, 171  
 Dorf, R.C., 75  
 Dowling, W.F., 302  
 Doyle, J., 75, 204, 466, 467  
 DPPLL, 284, 285, 288-289, 297, 302, 516  
 DPPLL-SODDISFACIBILE?, 286  
 Drabble, B., 576  
 Draper, D., 578  
 Drebbel, C., 23  
 Dreussi, J., 577  
 Dreyfus, H.L., 463, 592, 618  
 Dreyfus, S.E., 118  
 Du, D., 303  
 dualismo, 10, 597, 609  
 Dubois, O., 302  
 Durfee, E.H., 580  
 Dyer, M., 34
- 
- E**
- Eastlake, D.E., 241  
 Ebeling, C., 241  
 eccezioni, 408  
 Eckert, J., 21  
 economia, 15-16, 75  
 Edmonds, D., 25  
 Edmonds, J., 14  
 educata convenzione, 595  
 Edwards, D.J., 241  
 Edwards, P., 611  
**EFFETTI**, 569  
 effetto, 480  
     assiomi di, 420  
     condizionale, 549  
     disgiuntivo, 549  
     esterno, 537  
     implicito, 423  
     interno, 539  
     orizzonte, 225  
     positivo, 489  
     primario, 537  
     secondario, 537  
**EFFETTO**, 549  
 Einstein, A., 3  
 Eiter, T., 466  
 elaborazione parallela distribuita,  
     v. reti neurali  
 elemento esecutivo, 71, 72  
 eliminazione degli and, 272  
 ELIZA, 589  
 Elliot, G.L., 203  
 empirismo, 11  
 Enderton, H.B., 342, 396  
 ENIAC, 21
- Entscheidungsproblem, 13  
 enunciati osservativi, 11  
 Ephrati, E., 580  
 epifenomeno, 600  
 epistemologico, impegno, 311, 340  
**EQP**, 394  
 equivalenza inferenziale, 349  
 equivalenza logica, 270  
 Erdmann, M.A., 118  
 ereditarietà, 411, 445  
     multipla, 445  
 Ernst, G., 170  
 Ernst, M., 521  
 Erol, K., 577  
 esecuzione, 46, 83, 115  
**ESPANDI-GRAFO**, 507  
 espansione (di stati), 94  
 espansione iterativa, 171  
 esplorazione, 52, 160-168  
 esplosione combinatoria, 31  
 estensione (di un collegamento causale), 566  
 estensione (di una teoria di default), 456  
 estensioni singole, 225  
**ESTRAI-SOLUZIONE**, 507, 510, 511  
 estrinseche, proprietà, 417  
 Etchemendy, J., 305  
 etica, 604-609  
 Etzioni, O., 76, 579, 606, 620  
 EU, v. utilità, attesa  
 Euclide, 13  
 euristica  
     della lista di cancellazione vuota, 491  
     di grado, 187, 203  
     funzione, 126, 139-144, 478  
     indipendente dal dominio, 478  
     Manhattan, 140  
     min-conflicts, 195  
     per la pianificazione, 490-492  
 euristiche, 168  
     ammissibile, 129  
     composita, 143  
     del valore meno vincolante, 187  
     della distanza in linea d'aria, 126  
 European Space Agency, 577  
**EVAL**, 221, 224

Evans, T.G., 28, 41  
 eventi, 407, 426-433  
     calcolo degli, 425-426, 463  
     discreti, 428  
     generalizzati, 426  
     liquidi, 429  
 evoluzione, 157, 312  
     automatica, 31  
     delle macchine, 31  
 expectiminimax  
     complessità di, 230  
     valore, 228, 243

**F**

Fagin, R., 204, 464  
 Fahlman, S.E., 28, 466, 467  
 Farrell, R., 398  
 FASTFORWARD, 522  
 fatto, 280  
 fattore di certezza, 33  
 fattore di ramificazione, 97  
     effettivo, 140, 170, 219  
 fattorizzazione, 276, 379  
 Feigenbaum, E.A., 32, 41, 461  
 Feinstein, M.H., 41  
 Feldman, J., 41  
 Feller, W., 630  
 Felner, A., 171  
 fenomenologia, 594  
 Fermat, P., 14  
 Ferraris, P., 578  
 FETCH, 354, 402  
 FF (pianificatore  
     FASTFORWARD), 522  
 Fikes, R.E., 75, 342, 400, 519,  
     577, 578  
 film  
     A.I., 608  
     Matrix, 607  
     Terminator, 607  
 Filone di Megara, 301  
 filosofia, 10-12, 74, 587-612  
 filtraggio, 457  
 Findlay, J.N., 462  
 Firby, R.J., 576, 580  
 Fischer, M.J., 462  
 fisica qualitativa, 416  
 Flannery, B.P., 172

fluenti, 418, 429  
 fMRI, 18  
 Fogel, D.B., 173  
 Fogel, L.J., 173  
 FOL-BC-ASK, 366, 367, 371  
 FOL-FC-ASK, 359, 360, 367  
 FOPC, v. calcolo dei predicati  
 FORBIN, 576  
 Forbus, K.D., 398, 467  
 Ford, K.M., 41, 610  
 Forgy, C., 398  
 Forma Backus-Naur, v. BNF  
 forma di Horn, 280, 301  
 FORMULA-OBIETTIVO, 84  
 FORMULA-PROBLEMA, 84  
 formule  
     atomiche, 264, 317, 323, 341  
     complesse, 264, 318, 341  
     configurazioni fisiche, 263  
     derivate, 263  
     in una KB, 253, 299  
     quantificate, 340  
 Forrest, S., 173  
 Fortescue, M.D., 312  
 Fourier, J., 203  
 Fox, M.S., 521, 576  
 frame  
     assioma di, 421  
     problema del, 421, 462, 463,  
         481  
     rappresentazione, 34, 464  
 Franco, J., 302  
 Frank, M., 205  
 Frank, R.H., 605  
 Frankenstein, 607  
 FREDDY, 93, 578  
 Fredkin, premio, 242  
 Frege, G., 12, 301, 341, 396  
 Freuder, E.C., 204  
 Friedberg, R.M., 31, 174  
 Friedman, G.J., 172  
 FRITZ, 234  
 frontiera, 96  
 Frost, D., 205  
 Fuchs, J.J., 577  
 Fukunaga, A.S., 577  
 funzionalismo, 75, 596, 601, 610  
 funzione, 310

densità di probabilità  
     cumulativa, 629  
     di errore, 630  
     di unità, 414  
     di valutazione, 126, 211,  
         221-223  
     di valutazione lineare, 145  
     lineare pesata, 223  
     obiettivo, 145, 180  
     successore, 85, 181, 211  
     totale, 314  
     utilità, 211  
 Furst, M., 521

**G**

Galileo, G., 3, 72  
 Gallaire, H., 398  
 Gallier, J.H., 302, 342, 396  
 Gallo, G., 118  
 Gardner, M., 301  
 Garey, M.R., 630  
 Garey, M.R., 630  
 Garfield, J.L., 41  
 GARI, 576  
 Garrett, C., 154  
 Gaschnig, J., 170, 176, 203  
 Gasquet, A., 576  
 Gasser, R., 171, 235  
 Gauss, C.F., 117  
 Gauss, K.F., 202  
 Gauss-Jordan, eliminazione di,  
     627  
 gaussiana, distribuzione, 629  
 Gawande, A., 606  
 Geffner, H., 522, 576, 578  
 Gelatt, C.D., 172, 204  
 Gelernter, H., 27, 400  
 Gelfond, M., 466  
 General Problem Solver, 6, 519  
 Genesereth, M.R., 75, 118, 255,  
     342, 376, 383, 399, 400  
 genetici, algoritmi, 31, 153-  
     156, 173  
 Gentzen, G., 396  
 Geometry Theorem Prover, 27  
 Georgeff, M.P., 580  
 Gerevini, A., 521  
 Ghallab, M., 519, 521, 577

Ghose, S., 171  
 Giacomo, G.D., 463  
 GIB, 236  
 Gilmore, P.C., 396  
 Gini, M., 578  
 Ginsberg, M.L., 204, 236, 399,  
     580  
 giochi, 15, 209-210, 239  
     bridge, 231  
     campo minato, 302  
     con i dadi, 231  
     contro la natura, 550  
     d'azzardo, 14  
     di carte, 231  
     di fortuna, 227-233  
     Go, 235-236  
     multiplayer, 214-217  
     Othello, 210, 235  
     programmi, 233-236  
     Qubic, 235  
     teoria dei, 15  
 Giunchiglia, E., 578, 578  
 giustificazione, 453  
 Glover, F., 172  
 Go-Moku, 235  
 Go4++, 236  
 Gödel, K., 13, 301, 376, 396,  
     590  
 Goebel, R., 38, 74  
 Goemate, 236  
 Goldberg, D.E., 173  
 Golden, K., 580  
 Goldman, R., 578  
 Golgi, C., 16  
 GOLOG, 463  
 Gomes, C., 172  
 Good, I.J., 240  
 Good Old-Fashioned AI  
     (GOFAI), 592, 593, 611  
 Goodman, D., 38  
 Goodman, N., 463  
 Gordon, M.J., 342  
 Gottlob, G., 205, 398, 399  
 grado di verità, 311  
 grafi esistenziali, 444  
 grafo AND-OR, 281  
 GRAPHPLAN, 501, 506-507,  
     510, 516-518, 579  
 Grassmann, H., 342  
 greedy, ricerca, 147

Green, C., 28, 342, 394, 398,  
     402, 462, 519  
 Greenblatt, R.D., 241  
 Greenstreet, M.R., 401  
 griglia rettangolare, 109  
 ground, termine, 318-322, 348  
 grounding, 262  
 Grove, W., 590  
 Grumberg, O., 401, 522  
 GSAT, 302  
 Gu, J., 204, 302  
 Guard, J., 394, 400  
 Guha, R.V., 461  
 Guy, R.K., 118

---

**H**

Haas, A., 464  
 HACKER, 520  
 Haken, W., 202  
 Halpern, J.Y., 464  
 Halpin, M.P., 205  
 Hammond, K., 577  
 Hamscher, W., 75  
 Hanks, S., 578  
 Hansen, E., 171, 579  
 Hansen, P., 302  
 Hanski, I., 76  
 Hansson, O., 171, 176  
 Haralick, R.M., 203  
 Harel, D., 398, 462  
 Harman, G.H., 611  
 HARPY, 172  
 Hart, P.E., 170, 577, 578  
 Hart, T.P., 241  
 Haslum, P., 576  
 Haugeland, J., 4, 41, 592, 612  
 Havelund, K., 394  
 Hayes, P.J., 41, 462, 463, 466,  
     467, 610  
 Heath Robinson, 21  
 Heawood, P., 591  
 Hebb, D.O., 24, 29  
 Heckerman, D., 37  
 Heinz, E.A., 242  
 Held, M., 171  
 Helmert, M., 521  
 Helmholtz, H., 19  
 Hempel, C., 11

**I**

IA, v. intelligenza artificiale  
 IBM, 25, 27, 38, 234  
 IBM 704, computer, 235  
 IDA\*, v. ricerca, ad approfondimento iterativo  
 IEEE, 462  
 IJCAI (International Joint Conference on AI), 42  
 impegno  
     epistemologico, 311, 340  
     ontologico, 311, 340  
 implicazione, 260, 264  
 incertezza, 33, 38, 408  
 incompletezza, teorema di, 13, 387, 590  
 incomputabilità, 14  
 indecidibilità, 14, 42  
 indeterminatezza, 546  
 indicizzazione, 354-355  
     dei predicati, 354  
 indipendenza dei sottobiettivi, 490  
 individuazione, 416  
 induzione, 11  
     matematica, 13  
 inferenza, 253, 347  
     procedura di, 334  
         regole di, 272, 299  
 information gathering, 51  
 ingegneria ontologica, 407-410  
 insieme di supporto, 389  
 insieme magico, v. magic set  
 insiemi di risposta,  
     programmazione con, 454  
 insiemi (nella logica del primo ordine), 328  
 intelligenza, 4, 46  
 intelligenza artificiale  
     applicazioni, 38-40  
     associazioni, 42  
     come progettazione di agenti razionali, 9  
     congressi, 42  
     definizione, 4  
     filosofia, 587-612  
     fondamenti, 9-24  
     futuro, 621-622  
     IA debole, 587, 588-594

IA forte, 587, 594-604  
 in tempo reale, 617  
 linguaggi di programmazione, 27  
 obiettivi, 618-621  
 possibilità, 587-594  
 riviste, 42  
 storia, 24-40  
 intenzionalità, 594, 611  
 intenzione congiunta, 574  
 interleaving (tra ricerca ed esecuzione), 115  
 Internet, ricerca su, 440  
 Internet, shopping su, 437-444  
 INTERPLAN, 520  
 interpretazione, 316  
 intervallo, 430-431  
 intervallo di protezione, v.  
     collegamento causale  
 intrattabilità, 14, 30, 42  
 intrinsiche, proprietà, 417  
 introduzione dell'esistenziale, 402  
 Inza, I., 173  
 IPFM, 579  
 IPL, 25  
 ipotesi  
     dei nomi unici, 423, 450  
     del mondo aperto, 511, 556  
     del mondo chiuso, 460, 479, 556  
 IPP, 521  
 ISIS, 576  
 istanze tipiche, 415  
 istanziazione esistenziale, 348  
 istanziazione universale, 348  
 ITEP, programma scacchistico, 241  
 IXTEL, 521

**J**

Jacquard, J., 22  
     telaio di, 22  
 Jaffar, J., 399  
 James, H., 20  
 James, W., 20  
 Jaskowski, S., 396  
 Jaumard, B., 302  
 Jeavons, P., 205

Jefferson, G., 594  
 Jeffrey, R.C., 401  
 Jevons, W.S., 301  
 Jimenez, P., 579  
 Johnson, C.R., 75  
 Johnson, D.S., 630  
 Johnson, D.S., 630  
 Johnson, W.W., 117  
 Johnson-Laird, P.N., 41  
 Johnston, M.D., 172, 204, 577  
 Jones, R., 398  
 Jonsson, A., 38, 74, 576  
 Joskowicz, L., 467  
 Joslin, D., 520  
 Jouannaud, J.-P., 400  
 JTMS, v. sistemi di  
     mantenimento della verità,  
     basati sulla giustificazione  
 Juels, A., 173  
 Jurafsky, D., 41

**K**

Kaelbling, L.P., 303  
 Kahneman, D., 5  
 Kaindl, H., 171  
 Kalah, 241  
 Kambhampati, S., 521, 576, 577  
 Kan, A., 117  
 Kanade, T., 39  
 Kanal, L.N., 171  
 Kanefsky, B., 15, 204, 302  
 Kanoui, H., 342, 399  
 Kant, E., 398  
 Kaplan, D., 464  
 Karmarkar, N., 172  
 Karp, R.M., 15, 118, 171, 630  
 Kasparov, G., 38, 233  
 Kaufmann, M., 401  
 Kautz, D., 577  
 Kautz, H., 172, 302, 521  
 Kaye, R., 302  
 KB, v. base di conoscenza  
 KB-AGENTE, 253  
 Keane, M.A., 173  
 Keene, R., 38  
 Keil, F.C., 6, 41, 611  
 Keim, G.A., 205

- Kempe, A.B., 591  
 Kent, C., 433  
 Kern, C., 401  
 Kernighan, B.W., 117  
 Khorsand, A., 171  
 KIDS, 401  
 Kirchner, C., 400  
 Kirkpatrick, S., 172, 204, 302  
 KL-ONE, 466  
 Kleer, J.D., 75  
 Knight, K., 4  
 Knoblock, C.A., 117, 519, 577  
 Knuth, D.E., 241, 400  
 Koehler, J., 521  
 Koenig, S., 118, 174  
 Kohn, W., 400  
 Kolodner, J., 34  
 Kondrak, G., 204  
 Konolige, K., 464, 580  
 Korf, R.E., 118, 171, 174, 242,  
     520  
 Kotok, A., 241  
 Koutsoupias, E., 172, 303  
 Kowalski, forma di, 377  
 Kowalski, R., 342, 365, 377,  
     399, 400, 462, 463, 466  
 Koza, J.R., 173  
 Kramnik, V., 234  
 Kraus, S., 580  
 Kraus, W.F., 173  
 Kripke, S.A., 464  
 KRYPTON, 466  
 Kuehner, D., 399  
 Kuijpers, C., 173  
 Kumar, P.R., 74  
 Kumar, V., 170, 205  
 Kuper, G.M., 399  
 Kurzweil, R., 4, 608
- L**
- La Mettrie, J.O., 605, 610  
 Ladkin, P., 463  
 Ladner, R.E., 462  
 Laguna, M., 172  
 Laird, J.E., 38, 365, 398, 577  
 Laird, P., 172, 204  
 Lakoff, G., 461
- Lamarck, J.B., 157  
 Langlotz, C.P., 37  
 Langton, C., 173  
 Lansky, A.L., 580  
 Laplace, P., 15  
 larghezza d'albero, 201  
 Larrañaga, P., 173, 177  
 Laruelle, H., 521, 576  
 Lassez, J.-L., 399  
 Lassila, O., 462  
 Latombe, J.-C., 576  
 Lawaly, 577  
 Lawler, E.L., 117, 170, 534  
 LCF, 342  
 Leśniewski, S., 464  
 LEANTAP, 401  
 Lederberg, J., 33, 461  
 Lee, R. C.-T., 401  
 Lefkowitz, D., 244  
 Legge di Moore, 608  
 leggi del pensiero, 7  
 leggi sociali, 573  
 Lehmann, D., 580  
 Leibniz, G.W., 10, 158, 301  
 Leiserson, C.E., 630  
 lemma di lifting, 384, 385  
 Lenat, D.B., 461, 472  
 Lenstra, J.K., 118, 577  
 Lenzerini, M., 466  
 Leonard, H.S., 463  
 Leone, N., 205, 399  
 Lesh, N., 579  
 Lesser, V.R., 580  
 letterale di risposta, 382  
 letterale (formula), 264  
 letterale negato, 392  
 Letz, R., 400  
 Levesque, H.J., 172, 302, 463,  
     466, 467, 580  
 Levitt, G.M., 239  
 Levy, D. N.L., 244  
 Lewis, D.K., 75  
 Li, C.M., 302  
 libero arbitrio, 10, 597  
 LIFE, 400  
 LIFO, coda, 102  
 Lifschitz, V., 466, 519
- lifting, 352, 352-356  
 Lighthill, J., 31  
 limite alla profondità, 224  
 Lin, S., 118, 170  
 Linden, T.A., 579  
 Lindsay, R.K., 461  
 line search, v. ricerca,  
     unidimensionale  
 linea di ritardo, 293  
 lineare  
     algebra, 626-628  
     risoluzione di input, 389, 392  
     vincolo, 181  
 linearizzazione, 493  
 linguaggio  
     comprensione, 28, 33  
     di programmazione, 307  
     e pensiero, 312  
     elaborazione, 24  
     naturale, 9, 308  
     traduzione, 30  
 linguistica, 24, 41  
     computazionale, 24  
 Linneo, 461  
 Lipkis, T.A., 466  
 Lisp, 27, 317  
 lista di aggiunte (in STRIPS), 480  
 lista di cancellazioni (in  
     STRIPS), 480  
 lista di legami, 324  
 liste, 329  
 Littman, M.L., 39, 173, 206, 579  
 livellato (un grafo di  
     pianificazione), 503  
 livelli (nei grafici di  
     pianificazione), 501  
 livello di implementazione, 254  
 Lloyd, E.K., 203  
 Lloyd, J.W., 399  
 località, 296  
 Locke, J., 11  
 locking, 392  
 Lodge, D., 621  
 Loebner, premio, 42  
 Logemann, G., 284, 302  
 Logic Theorist, 25, 301  
 logica, 7, 12, 259-263  
     del primo ordine, 307  
     descrittiva, 444, 448-450,  
         460, 466

di default, 456  
 di ordine superiore, 311  
 dinamica, 462  
 formule atomiche, 317  
 inferenza, 261, 347-406  
 interpretazioni, 315-316  
 modale, 434, 464  
 modelli, 313-315  
 non monotona, 455, 466  
 notazione, 7  
 onniscienza, 435  
 proposizionale, 252, 263-272, 309  
 quantificatore, 318-322  
 risoluzione, 274-280  
 temporale, 311, 462  
 termine, 317  
 uguaglianza, 323  
 variabile, 370

logicismo, 7  
 logico  
     agente, 253  
     connettivo, 24, 264, 318  
     piano, 301  
     positivismo, 11  
     ragionamento, 263-289

Logistello, 235  
 Lohn, J.D., 173  
 Long, D., 521  
 LOOKUP, 62  
 Lotem, A., 521  
 Lovelace, A., 22  
 Loveland, D., 284, 302, 399, 400, 401  
 Löwenheim, L., 342  
 Lowerre, B.T., 172  
 Lowrance, J.D., 580  
 Lowry, M., 394  
 Lowry, M.R., 401  
 Loyd, S., 118  
 LPG, 521  
 LRTA\*, 166, 174  
 LRTA\*-AGENTE, 167  
 Luby, M., 150  
 Lucas, J.R., 590, 591  
 Luce, D.R., 15  
 Luger, G.F., 42  
 Lullo, R., 10  
 Lusk, E., 401

**M**

MA\*, v. ricerca, A\* a memoria limitata  
 MAC, 189  
 MacHack 6, 241  
 Machover, M., 342  
 Mackworth, A.K., 38, 74, 190, 203, 205  
 macrop (macro operatore), 577  
 Madigan, C.F., 302  
 magic set, 366  
 Mahanti, A., 174  
 Maier, D., 204, 399  
 Majercik, S.M., 579  
 Mali, A.D., 577  
 Malik, S., 302  
 Manhattan, v. euristica, Manhattan  
 Manna, Z., 342, 400, 401  
 Manolios, P., 401  
 margine, 532  
 margine minimo, 535  
 Marriott, K., 204, 400  
 Marsland, A.T., 244  
 Martelli, A., 171, 400  
 Martin, J.H., 41  
 Martin, N., 398  
 Martin, P., 577  
 Maslov, S.Y., 398  
 Mason, M., 118, 577  
 Mateis, C., 466  
 matematica, 12-14, 27, 41  
 materialismo, 10, 597, 610  
     eliminativo, 610  
 Mates, B., 301  
 matrice, 627  
 Mauchly, J., 21  
 MAX, 215  
 Mayer, A., 171, 176  
 Mazumder, P., 118  
 McAllester, D.A., 36, 242, 393, 467, 521  
 MCC, 34  
 McCarthy, J., 25, 26, 27, 28, 74, 300, 342, 397, 433, 462, 466, 602  
 McCartney, R.D., 401  
 McClelland, J.L., 35  
 McConnell-Ginet, S., 41  
 McCulloch, W.S., 23, 29, 303  
 McCune, W., 390, 394, 400  
 McDermott, D., 398, 445, 465, 519, 522, 578, 579  
 McDermott, J., 4, 34, 365, 398  
 McDonald, R., 310  
 McGregor, J.J., 203  
 McMillan, K.L., 522  
 McNealy, S., 606  
 MDP, vedi processi decisionali di Markov  
 media sulla chiaroveggenza, 231  
 Meehan, J., 398  
 Meehl, P., 590  
 Mellish, C.S., 400  
 memoizzazione, 374, 395  
 memoria, 312  
     a breve e lungo termine, 365  
 MEMS, 614  
 Mendel, G., 157  
 mente  
     come sistema fisico, 10  
     filosofia della, 609, 611  
     teoria della, 6  
     visione dualistica, 610  
 mereologia, 463  
 MERLIN, 465  
 Merlin, P.M., 399  
 metalivello, spazio degli stati di, 138  
 metaragionamento, 238  
     in teoria delle decisioni, 618  
 metaregole, 376  
 metodi deboli, 32  
 metodo del cammino critico, 531  
 metodo di connessione, 397  
 metodo di Newton-Raphson, 159  
 metodo inverso, 398  
 Metropolis, N., 172  
 MGONZ, 589  
 MGSS\*, 242  
 MU (unificatore più generale), 354, 356, 385, 402  
 Michaylov, S., 399  
 Michie, D., 41, 93, 169, 171, 243, 578

- MICRO-PLANNER, 397  
 micromondi, 28  
 Mill, J.S., 12  
 Miller, D., 576  
 Millstein, T., 522  
 Milner, A.J., 342  
 MIN, 215  
 MIN-CONFLICTS, 196, 286  
 min-conflicts, v. euristica,  
     min-conflicts  
 MINIMAX, 220  
 minimax, v. ricerca, minimax  
 minimo albero di copertura,  
     171, 176  
 minimo impegno, 492  
 Minker, J., 398  
 Minsky, M.L., 28, 31, 34, 465  
 Minton, S., 172, 204, 577  
 missionari e cannibali, 117, 461  
 misura delle prestazioni, 49  
 misure, 414-416  
 MIT, 25-28  
 Mitchell, D., 172, 302  
 Mitchell, M., 174  
 Mitchell, T.M., 75  
 model checking, 261  
 modelli nascosti di Markov, 36  
 modello, 66, 259, 340  
     minimo, 452, 455  
     stabile, 453  
 Modus Ponens, 272, 301, 389,  
     395, 435  
     generalizzato, 352  
 Mohr, R., 190, 203  
 mondo chiuso, 450-453  
 mondo dei blocchi, 28, 33, 467,  
     484-486, 564  
 mondo dell'aspirapolvere, 47-48,  
     76, 550  
 mondo possibile, 259, 340, 464  
 monismo, 597  
 monitoraggio di esecuzione,  
     547, 558-564, 576, 579  
 monotonicità, 274, 455  
 monotonicità, condizione, 170  
 monotonicità di un'euristica, 130  
 montaggio automatico, 93  
 Montague, R., 464  
 Montanari, U., 170, 203, 400  
 Moore, A.W., 172  
 Moore, E.F., 118  
 Moore, J.D., 577  
 Moore, J.S., 394, 400, 401, 465  
 Moore, R.C., 464, 468  
 Morgan, J., 580  
 Morgenstern, L., 464, 467  
 Morgenstern, O., 15, 240  
 Morris, P., 38, 74, 576  
 Morrison, E., 240  
 Morrison, P., 240  
 Moskewicz, M.W., 302  
 Mostow, J., 171, 176  
 Moussouris, J., 241  
 MRS (sistemi di ragionamento  
     di metalivello), 376  
 MST (albero minimo di  
     copertura), 176  
 mucchio, 413  
 Müller, M., 244  
 Murakami, T., 235  
 Murga, R., 173, 177  
 Muscettola, N., 38, 75, 576, 577  
 mutazione, 155  
 MUTAZIONE, 155  
 mutex, 503  
 mutua esclusione, 503  
 MYCIN, 33  
 Myers, K.L., 580
- 
- N**
- Nagel, T., 610  
 Nalwa, V.S., 19  
 Nardi, D., 466  
 NASA, 38, 398, 467, 517, 577  
 NASL, 579  
 naturalismo biologico, 596  
 Nau, D.S., 170, 236, 243, 522,  
     577  
 navigazione dei robot, 93  
 NAVLAB, 39  
 Nayak, P., 75, 467, 577  
 Nealy, R., 235  
 Nebel, B., 519, 521  
 negazione, 264  
 negazione come fallimento, 370,  
     453-454  
 Nelson, G., 400  
 NESSUNA-SOLUZIONE-  
     POSSIBILE, 507  
 NETL, 466  
 Netto, E., 118  
 NEUROGAMMON, 243  
 neuroni, 16, 24, 599  
 neuroscienze, 16-19  
 Nevins, A.J., 398  
 Newborn, M.M., 171  
 Newell, A., 6, 25, 27, 38, 74,  
     117, 169, 240, 241, 300, 365,  
     465, 519, 577  
 Newton, I., 3, 62, 158, 159, 172  
 Nguyen, X., 520, 521  
 Nielsen, P.E., 398  
 Niemelä, I., 466  
 Nigenda, R.S., 521  
 Nilsson, N.J., 4, 38, 75, 117,  
     170, 176, 242, 300, 342, 383,  
     400, 519  
 Nixon, R., 456  
 NLP (natural language  
     processing), 5  
 NOAH, 520, 578  
 nodi di possibilità (in un albero  
     di gioco), 228  
 nodo foglia, 96  
 nodo padre, 95  
 nomenclatura binomiale, 461  
 nomi lunghi composti, 337  
 non monotonicità, 455  
 non-deterministico, v. ambiente,  
     stocastico  
 non-episodico, 56  
 NONLIN, 520  
 NONLIN+, 576  
 normale, distribuzione, 629  
 Norvig, P., 38, 398, 414  
 notazione  
     aritmetica, 7  
     logica, 7  
     O(), 625  
 Nourbakhsh, I., 118  
 Nowatzyk, A., 241  
 Nowick, S.M., 394

Nowlan, S.J., 173  
 NP (problemi difficili), 625-626  
 NP-completezza, 14, 90, 117,  
 182, 271, 302, 466  
 NSS, programma per scacchi, 241  
 numeri naturali, 327  
*Número Telefono*, 436  
 Nussbaum, M.C., 11, 609

---

**O**

*O()*, notazione, 625  
 O'Reilly, U.-M., 173  
 O-PLAN, 545, 576, 577  
 obiettivi  
     agente basato su, 68, 74  
     formulazione di, 82, 547  
     inferenziali, 324  
     ragionamento guidato dagli,  
       238, 284  
     serializzabili, 517  
 obiettivo, 68, 82, 115  
     test, 85, 116, 181, 487  
 Ogawa, S., 18  
 oggetti, 310  
 oggetti mentali, 433-437  
 oggetto, 317  
     composto, 413  
 Oglesby, F., 394, 400  
 Olawsky, D., 579  
 onniscienza, 51  
     logica, 435  
 ONTIC, 393  
 ontologia, 334, 338  
     generale, 437  
     superiore, 408  
 opacità, 434  
 opacità referenziale, 465  
 Open Mind Initiative, 462  
 operatori modali, 434, 462  
 Oppacher, F., 173  
 Oppen, D.C., 400  
 OPS-5, 365, 396, 398  
 OPTIMUM-AIV, 577  
 or esclusivo, 266  
 OR-parallelismo, 373  
 ORDINAMENTO, 538  
 ordinamento dei congiunti, 362  
 Organon, 300, 461

Oscar, 580  
 osservabile, 56  
 Othello, 210, 235  
 OTTER, 390, 391, 394, 401  
 ottimalità (di un algoritmo di  
     ricerca), 96, 116  
 ottimalità limitata, 619  
     asintotica (ABO), 620  
 ottimamente efficiente  
     (algoritmo), 133  
 Overbeek, R., 401  
 Owens, A.J., 173

---

**P**

Palay, A.J., 242  
 Pallottino, S., 118  
 Pang, C.-Y., 118  
 Panini, 24  
 Papadimitriou, C.H., 172, 173,  
 303  
 Papert, S., 31  
 PARADISE, 238  
 paradosso, 464  
 parallelismo  
     AND-, 373  
     OR-, 373  
 paramodulazione, 388, 400  
 Pardalos, P.M., 303  
 Park, S., 394  
 Parrod, Y., 576  
*ParteDi*, relazione, 412  
 partizione, 412  
 Pascal, B., 10, 14  
 Pasero, R., 342, 399  
 passi condizionali, 549  
 passo ridondante, 566  
 Paterson, M.S., 400  
 Patil, R., 466  
 Patrick, B.G., 171  
 pattern matching, 361  
 Paull, M., 303  
 PDDL, 482, 519  
 Peano, G., 342  
 Pearl, J., 37, 75, 127, 170, 171,  
 172, 205, 241, 243  
 Pearson, J., 205  
 PEAS, descrizione, 53, 55

Pecheur, C., 394  
 Pednault, E. P.D., 519, 580  
 Peirce, C.S., 203, 341, 444, 464  
 Peled, D., 401  
 Pelikan, M., 173  
 Pell, B., 75, 577  
 Pemberton, J.C., 174  
 Penberthy, J.S., 521  
 Pengi, 580  
 Penix, J., 394  
 Penrose, R., 590  
 pensare razionalmente, 7  
 Peot, M., 579  
 percettore, 29  
     potenza espressiva, 31  
     teorema di convergenza, 29  
 percezione, 43, 46, 330  
     attiva, 557  
     automatica, 557  
     azioni di, 546  
 Pereira, F., 369, 371  
 Pereira, L.M., 371  
 performance, v. misura delle  
     prestazioni  
 peso dinamico, 170  
 Pfeifer, G., 466  
 Philips, A.B., 172, 204  
 piani  
     condizionali, 575  
     congiunti, 570  
     consistenti, 495  
     continui, 565  
     difetti, 567  
     libreria di, 538  
     monitoraggio dei, 559-562  
     non lineari, 520  
     riconoscimento di, 574  
     universali, 580  
 PIANIFICATORE, 560-561  
 pianificazione, 68, 238, 418,  
 477-503  
     basata sui casi, 578  
     classica, 477, 545, 547  
     come soddisfacibilità, 510-516  
     con ordinamento parziale, 493  
     condizionale, 547-558, 575  
     conforme, 546, 576, 579  
     congiunta, 570  
     continua, 547, 564-569, 576,  
       580  
     di contingenza, 546

- di regressione, 489, 521  
di un cammino, 28  
e azione, 545-548  
ed esecuzione, 564-569  
euristiche per la, 500-501  
gerarchica, 535-545, 575  
grafi di, 501-510, 519  
in una fabbrica di birra, 579  
lineare, 520  
multiagente, 569-575  
multibody, 571  
nel mondo dei blocchi, 29  
nello spazio degli stati, 486-491  
progressione, 486  
reattiva, 580  
rete gerarchica di attività, 536  
senza interleaving, 525  
senza sensori, 546  
storia della, 519  
su una portaerei, 579
- Pijls, W., 243  
pila, vedi stack  
pilota automatico, 400  
pinguino, 580  
Pinker, S., 309  
Pitts, W., 23, 24, 29, 303  
Plaat, A., 243  
Place, U.T., 610  
PLAN-ERS1, 577  
PLANEX, 579  
Plankalkül, 21  
Platone, 300, 463, 609  
Plotkin, G., 400  
Pnueli, A., 462  
Podelski, A., 399  
Pohl, I., 118, 170  
politica, 580  
Pollack, M.E., 521, 577, 580  
Poole, D., 4, 38, 74  
POP, 494, 495, 496, 499, 518, 521, 539, 541, 567, 571-572  
Porfirio, 465  
porta logica, 337  
Posegga, J., 401  
positivismo logico, 11  
posizione intenzionale, 611  
Post, E.L., 301  
potatura, 133, 211  
alfa-beta, 217, 245  
di futilità, 234
- in avanti, 226  
nei problemi di contingenza, 230  
pozzo senza fondo, 254  
Pratt, V.R., 462  
Prawitz, D., 396  
precondizioni, 480  
aperte, 495, 568  
esterne, 537  
predecessori, 107, 488  
premessa, 264  
Press, W.H., 172  
Price Waterhouse, 576  
Prieditis, A.E., 142, 171, 176  
Princeton, 25  
*Principia Mathematica*, 26  
Prinz, D.G., 240  
Prior, A.N., 462  
probabilità, 14, 37  
distribuzione, 237  
teoria, 311  
problema, 85, 115  
a n regine, 288  
a un milione di regine, 195, 204  
del commesso viaggiatore, 92, 117, 171, 176  
del frame, 463  
della qualificazione, 546, 592  
della ramificazione, 423  
della terminazione, 351  
delle 8 regine, 90, 117  
di coloratura di mappe, 202, 591  
di contingenza, 113  
di esplorazione, 160  
di ricerca dell'itinerario, 91  
di viaggio, 92  
formulazione di un, 82  
generatore di, 72  
giocattolo, 88  
intrinsecamente difficile, 625-626  
mente-corpo, 597-598  
missionari e cannibali, 120  
nel mondo reale, 88  
rilassato, 141, 142, 490  
risoluzione, 31  
rompicapo a 8 tasselli, 139, 141, 170
- processi decisionali di Markov, 15  
processo, 429
- processo decisionale composito, 171  
PRODIGY, 577  
PROFONDITÀ, 95, 103  
programmazione  
con insiemi di risposta, 454  
dinamica, 170, 374  
funzionale, 400  
genetica, 173  
orientata agli oggetti, 23, 445  
programmazione logica, 342, 368-376, 399  
con tabelle, 375, 399  
con vincoli, 375, 399  
proiezione, 418  
Prolog, 34, 369, 395, 521  
parallelismo, 373  
Prolog Technology Theorem  
Prover (PTTP), 392, 401  
propagazione delle unità, 285  
proposizionale, simbolo, 264  
proposizionalizzazione, 350  
proprietà (relazione unaria), 308  
Prosser, P., 204  
protesi cerebrale, v. cervello,  
protesi  
PROVERB, 39  
PRS (Procedural Reasoning System), 580  
Pryor, L., 579  
pseudocodice, 632-633  
psicologia, 19-21  
sperimentale, 6, 19  
PSPACE, 626  
PSPACE-completezza, 508, 519  
Puccini, 580  
Pullum, G.K., 312  
PUNTATORE-GLOBALE-TRACCIA, 372  
punto critico, 288  
punto di scelta, 370  
punto fisso, 281, 360  
Puterman, M.L., 75  
Putnam, H., 75, 284, 302, 383, 396  
Pylyshyn, Z.W., 611

**Q**

- QA3, 342, 463, 519  
 QLISP, 400  
 qualia, 598, 599  
 quantificatore, 341  
     esistenziale, 320-321  
     in logica, 318-322  
     nidificato, 321-322  
     universale, 318  
 Qubic, 235  
 query (logica), 324  
 Quevedo, T., 240  
 quiescenza, 225, 240  
 Quillian, M.R., 465  
 Quine, W.V., 342, 415, 464  
 Quinta Generazione, progetto, 34

**R**

- R1, 34, 365, 398  
 Rabani, Y., 173  
 Rabinovich, Y., 173  
 ragionamento, 28  
     basato sui modelli, 332  
     di default, 450-457  
     guidato dagli obiettivi, 238  
     incerto, 37  
     proposizionale, 272-284  
     psicologico, 468  
     spaziale, 467  
 ragionatore socratico, 393  
 Raiffa, H., 15  
 Rajan, K., 38, 75  
 Ramakrishnan, R., 398  
 ramificazioni, 482  
 Ramsey, F.P., 15  
 Rao, A., 76  
 RAP, 580  
 Raphael, B., 170, 397  
 Raphson, J., 159, 172  
 rapporto Lighthill, 31, 35  
 Ratner, D., 118  
 razionale, pensiero, 7  
 razionalità, 4, 48-50  
     calcolativa, 619  
     limitata, 9, 619  
     perfetta, 9, 618  
 RBFS, 134-138  
 Reboh, R., 400

- Rechenberg, I., 172  
 refutazione, 272, 396  
 Regin, J., 204  
 REGOLA-AZIONE, 65, 67  
 REGOLA-CORRISPONDENTE, 65  
 regole  
     causalì, 332  
     condizione-azione, 63, 243  
     di default, 456  
     di implicazione, 264  
     diagnostiche, 331  
     if-then, 63, 264  
 reificazione, 410, 433  
 Reingold, E.M., 203  
 Reiss, M., 236  
 Reiter, R., 303, 463, 466  
 Reitman, W., 244  
 relazioni, 308  
 Remote Agent, 38, 75, 394,  
     517, 577  
 Remus, H., 244  
 renominazione, 359  
 REPOP, 521  
 requisiti di memoria, 100  
 Rescher, N., 462  
 RESET-TRACCIA, 372  
 rete, 365  
 rete gerarchica (HTN), 536  
 rete neurale, 24, 29, 35, 235  
     a livello singolo, v. percepitrone  
     apprendimento, 24  
     espressività, 24  
     hardware, 25  
 reti causali, v. Bayesiane, reti  
 reti di attività, v. task network  
 reti probabilistiche, v. reti  
     Bayesiane  
 reti semantiche, 444-448, 460,  
     465  
 retropropagazione, 31, 35  
 Reversi, 235  
 Reynolds, C.W., 580  
 riavvii casuali, 177  
 ricerca, 68, 83, 115  
     A\*, 129-134  
     A\* a memoria limitata, 137-  
         138, 171  
     A\* ad approfondimento  
         iterativo, 134, 171  
 a costo uniforme, 100, 116  
 a costo uniforme, 100-101  
 a profondità limitata, 104  
 ad approfondimento iterativo,  
     104-106, 116, 118, 224, 392  
 albero di, 94  
 alfa-beta, 217-221, 239  
 all'indietro, per la  
     pianificazione, 488-490  
 apprendere per la, 138  
 B\*, 242  
 beam, 152  
 beam stocastica, 152  
 best-first, 126, 168  
 best-first greedy, 127  
 best-first ricorsiva(RBFS), 134-  
     138, 171  
 bidirezionale, 116  
 con backtracking, 103, 185-187  
 con informazione parziale,  
     111, 116  
 dell'itinerario, 91  
 di quiescenza, 225  
 euristica, 99, 169  
 generale, 115  
 greedy, 147  
 hill-climbing, 146-150, 165  
 in ampiezza, 99, 116  
 in avanti, per la pianificazione,  
     486-488  
 in profondità, 101-103  
 in spazi continui, 156-160, 172  
 in tempo reale, 174, 221-227  
 in un CSP, 184-195  
 informata, 99, 125-138  
 local beam, 152  
 locale, 145-156, 172, 204,  
     286-287, 302  
 locale, per CSP, 195-196  
 minimax, 214-218, 237, 239  
 nodo, 94  
 non informata, 99, 116, 118  
 offline, 160  
 online, 161-168, 173  
 parallela, 174  
 ripetizione negli stati, 108-111  
 simulated annealing, 150-152  
 strategia di, 95  
 su Internet, 440  
 tabù, 172  
 tagliare la, 224-225  
 unidimensionale, 159  
 ricerca operativa, 15, 75, 117,  
     171, 577

- RICERCA-ALBERO, 95, 98, 110, 116, 126, 169  
 RICERCA-ALFA-BETA, 220, 224  
 RICERCA-AND, 552  
 RICERCA-APPROFONDIMENTO-ITERATIVO, 105, 141  
 RICERCA-BACKTRACKING, 185, 269, 284  
 RICERCA-BEST-FIRST, 126  
 RICERCA-GRAFO, 110, 111, 116, 126, 169, 174, 221, 583  
 RICERCA-GRAFO-AND-OR, 551-553, 583  
 RICERCA-OR, 552, 583  
 RICERCA-PROFOUNDITÀ-LIMITATA, 103, 104  
 Rich, E., 4  
 ricompense, 73  
 riconoscimento vocale, 36  
 ridotto, 453  
 Rieger, C., 34  
 Riesbeck, C., 34, 398  
 RIMUOVI-PRIMO, 96, 98, 111, 190  
 Ringle, M., 612  
 Rintanen, J., 579  
 ripetizione negli stati, 108  
 ripianificazione, 547, 558-564, 578  
 RIPRODUZIONE, 155  
 riscrittura dei termini, 400  
 riscritture, 390  
 risoluzione, 28, 31, 274-277, 341, 376-394  
     binaria, 379  
     chiusura della, 278, 385  
     completezza della, 383, 383-386  
     di input, 389  
     di input, 389  
     lineare, 389  
     strategie di, 388-389  
     unitaria, 276  
 risorse, 529-535  
     consumabili, 533  
     nella pianificazione, 532, 575  
     riutilizzabili, 532  
     vincoli sulle, 532-535  
 risposta, 20  
 Rissland, E.L., 41  
 roba, 416  
 Robbins, algebra di, 393  
 Roberts, M., 240  
 Robertson, N., 205  
 Robinson, A., 396  
 Robinson, G., 400  
 Robinson, J.A., 28, 302, 342, 376, 383, 397  
 robot  
     calcio per, 77, 210, 580  
     navigazione, 93  
 robotica, 6  
     cognitiva, 463  
 Rochester, N., 25, 27  
 rombo di Nixon, 456  
 rompicapi a tasselli mobili, 90  
 rompicapo a 8 tasselli, 88, 117, 138, 141, 170  
 Rorty, R., 610  
 Rosenblatt, F., 29  
 Rosenblitt, D., 521  
 Rosenbloom, P.S., 38, 365, 398, 577  
 Rosenblueth, A., 23  
 Rosenbluth, M., 172  
 Rosenfeld, E., 41  
 Rosenschein, S.J., 75, 303  
 Rosenthal, D.M., 611  
 Rossi, F., 203  
 Roussel, P., 342, 399  
 Roveri, M., 521, 579  
 RSA (Rivest, Shamir e Adelman), 394  
 RSA, v. criptazione a chiave pubblica  
 Rumelhart, D.E., 35  
 Russell, B., 11, 24, 26, 396  
 Russell, S.J., 38, 171, 242, 376, 414  
 Ryder, J.L., 244
- 
- S**
- Sabin, D., 204  
 Sacerdoti, E.D., 400, 520, 577  
 Sacks, E., 467  
 Sadri, F., 463  
 Sagalowicz, D., 400  
 Sagiv, Y., 399  
 Sahni, S., 118  
 SAINT, 28  
 Sam, 394, 400  
 Samuel, A.L., 25, 27, 74, 233, 240  
 Sanscrito, 461  
 SAPA, 576  
 Sapir-Whorf, ipotesi di, 312  
 Sastry, S., 75  
 Sato, T., 399  
 SATPLAN, 511, 517-518, 521, 579  
 saturazione, 384  
 Sayre, K., 588  
 scacchi, 21, 30, 58, 118, 211, 222-223, 233, 239  
     automa giocatore di, 239  
 scarabei, 312  
 scarabeo stercorario, 52, 76  
 Scarcello, F., 399  
 SCEGLI-VARIABILE-NON-ASSEGNATA, 188  
 Schaeffer, J., 171, 235, 243  
 Schank, R.C., 34  
 schedule, 531  
 scheduling, 529-535  
 scheduling, job shop, 530  
 schema di induzione  
     matematica, 387  
 schema (in un algoritmo genetico), 156  
 Scherl, R., 463  
 Schickard, W., 10  
 Schmolze, J.G., 466  
 Schneeberger, J., 463  
 Schnitzius, D., 577  
 Schofield, P. D.A., 117  
 Schönig, T., 303  
 Schoppers, M.J., 580  
 Schrag, R.C., 302  
 Schröder, E., 301  
 Schubert, L.K., 462, 520  
 Schumann, J., 400  
 Schwartz, S.P., 461  
 scomposizione ad albero, 200  
 scomposizione esaustiva, 412

- scomposizione gerarchica, 535  
 Scriven, M., 610  
 scuola di Megara, 301, 462  
 Searle, J.R., 18  
 Seconda Guerra Mondiale, 15  
 SELEZIONE-CASUALE, 155  
 Selfridge, O. G., 25  
 Selman, B., 172, 302, 466, 521  
 Semantic Web, 462  
 semantica, 41, 259  
     della logica, 299  
 semidecidibile, 395  
 semidecidibile, problema, 351  
 semidinamico, ambiente, 58  
 sensori, 46, 54  
 sequenza percettiva, 46  
 sequenziale, circuito, 293  
 serendipità, 562  
 Sergot, M., 463  
 Serina, I., 521  
 SETHEO, 401  
 Settle, L., 394, 400  
 Seymour, P.D., 205  
 SGP, 521, 579  
 Shahan, R.W., 611  
 Shahookar, K., 117  
 Shakey, 28, 75, 519, 526, 579  
 Shalla, L., 400  
 Shanahan, M., 463  
 Shankar, N., 394  
 Shannon, C.E., 25, 210, 240  
 Shapiro, S.C., 41  
 Shaw, J.C., 118, 239, 301  
 Shazeer, N.M., 39  
 Shelley, M., 606  
 Shmoys, D.B., 118, 577  
 Shoham, Y., 75, 400, 463  
 Shortliffe, E.H., 33  
 SHRDLU, 29, 33, 397, 484  
 Siekmann, J., 401  
 SIGART, 42  
 Siklossy, L., 578  
 sillogismi, 7, 395  
 simboli  
     di costante, 315, 317, 341  
     di funzione, 315, 317, 341  
     di predicato, 315, 341  
 simbolo iniziale, di una  
     grammatica, 631  
 simbolo puro, 285  
 Simmons, R., 118  
 Simon, H.A., 6, 16, 25, 27, 41,  
     74, 117, 169, 233, 240, 241,  
     301, 394, 401, 519  
 Simon, J.C., 302  
 simplesso, algoritmo del, 172  
 simulated annealing, 150, 172,  
     204  
 SIMULATED-ANNEALING, 151,  
     286  
 sinapsi, 17  
 Sinclair, A., 173  
 sincronica, conoscenza, 330  
 sincronizzazione, 571  
 Singh, M.P., 75  
 Singh, S.P., 174  
 singolarità tecnologica, 608  
 sinonimia, 441  
 sintassi, 34, 259  
     logica, 299  
 sintesi, 394  
     deduttiva, 394  
     di algoritmi, 394  
     di programmi, 579  
 SIPE, 576-578  
 sistemi di mantenimento della  
     verità, 204, 458-459, 467  
         basati su assunzioni, 459, 467  
         basati sulla giustificazione,  
             458-459  
 sistemi di produzioni, 347,  
     365, 395  
 sistemi esperti, 461  
     basati su Prolog, 369  
     commerciali, 365  
     con incertezza, 37  
     HPP (Heuristic Programming  
         Project), 33  
     medici, 39  
     primi, 33  
 situazioni, 418  
 SKETCHPAD, 203  
 Skinner, B.F., 23, 75  
 Skolem, funzioni di, 378, 396  
 Skolem, T., 342, 396  
 skolemizzazione, 348, 378  
 slack, vedi margine  
 Slagle, J.R., 28, 241  
 Slate, D.J., 118  
 Slater, E., 240  
 Sloman, A., 611  
 Smith, A., 15  
 Smith, B., 38, 75, 576  
 Smith, D.E., 376, 399, 521, 578  
 Smith, D.R., 401  
 Smith, J.M., 173  
 Smith, R.G., 75  
 Smith, S. J.J., 236  
 Smith, W.D., 242  
 SMODELS, 466  
 Smolensky, P., 35  
 Snarc, 25  
 SNLP, 521  
 SOAR, 38, 365, 398, 577  
 Socrate, 7  
 soddisfabilità, 271, 302, 510  
 soddisfacimento di vincoli, 28  
     problema di, 179, 180-184  
 soddisfazione, 16  
 Soderland, S., 520  
 softbot, 56  
 Solomonoff, R.J., 25  
 soluzione, 83, 116, 495  
     ciclica, 552  
     di giochi, 211-217  
     di un problema di  
         pianificazione, 481  
     ottima, 86  
 SOLUZIONE, 98, 103, 111  
 soma, 17  
 SOMMA, 623-624  
 Sosic, R., 204  
 sostanziali collettivi, 416  
 sostanziali contabili, 416  
 sostanze, 416  
     spaziali, 429  
     temporali, 429  
 sostituzione, 323, 348  
 sotto-oggettivi serializzabili, 517  
 sottoeventi, 426  
 Soutchanski, M., 463  
 sovrascrittura (di un valore di  
     default), 447  
 Sowa, J., 467, 468

- SPASS, 400  
 spettrometro di massa, 32  
 spiegazioni (fornite da sistemi di mantenimento della verità), 459  
 Spielberg, S., 608  
 SPIKE, 577  
 SPIN, 394  
 Springsteen, B., 605  
 SQL, 450  
 SRI, 28, 342, 519  
 Srivastava, M., 394  
 Srivastava, B., 577  
 SSS\*, algoritmo, 243  
 stack, 102  
 Stader, J., 576  
 STAGE, 172  
 Stallman, R.M., 204  
 STAN, 521  
 standardizzazione separata, 353, 402  
 Stanford University, 26, 28, 32, 32, 342  
 Stanhope, dimostratore di, 301  
 stanza cinese, 601-604  
 stati  
     insieme di, 553  
     spazio degli, 85, 116  
     spazio di metalivello, 138  
 stati mentali, 597  
 statiche, variabili 8, 171, 632  
 statico, ambiente, 58  
 stato  
     corrente, nella ricerca locale, 145  
     -credenza, 553  
     del mondo, 83  
     del processo, 429  
     iniziale, 85, 116, 181, 211  
     intenzionale, 598  
     interno, 66  
     terminale, 211  
     vincoli di, 483, 514  
 STATO, 95, 103, 111, 147  
 stato successore, assioma di, 422  
 Steel, S., 577, 579  
 Stefik, M., 468  
 Stein, L.A., 621  
 Stickel, M.E., 392, 400  
 Stiller, L.B., 235  
 Stillings, N.A., 41  
 stimolo, 20  
 Stockman, G., 243  
 stoica, scuola, 301, 462  
 Stokes, I., 576  
 Stone, P., 580  
 Story, W.E., 118  
 Strachey, C., 240, 243  
 strategico, ambiente, 57  
 strategie (in un gioco), 212  
 stringhe (in logica), 434  
 STRIPS, 479-483, 514, 518, 523-528, 529, 536, 539, 572, 577  
 Strohm, G., 576  
 Stuckey, P.J., 205, 399, 400  
 STUDENT, 28  
 Subramanian, D., 467  
 SUBST, 348, 367, 371, 539  
 SUCCESSORI, 215, 552  
 Sun Microsystems, 606  
 supporto inconsistente, 504  
 Sussman, anomalia di, 520, 525  
 Sussman, G.J., 204, 398, 520  
 sussunzione, 389  
     con logiche descrittive, 448  
     reticolo di, 356  
     risoluzione, 389  
 Sutherland, I., 203  
 Swift, T., 399  
 Syrjänen, T., 466  
 Szathmáry, E., 173
- 
- T**
- T-SCHED, 577  
 T4, 576  
 tabella delle trasposizioni, 221  
 tabella di hash, 355  
 tabelle triangolari, 579  
 tabù, ricerca, 172  
 TACAIR-SOAR, 398  
 Tait, P.G., 118  
 Tamaki, H., 399  
 Tanca, L., 398  
 Tarjan, R.E., 630  
 Tarski, A., 13, 342  
 task environment, 53  
 task network, 520  
 tassonomia, 411, 441, 461  
 tassonomica, gerarchia, 34, 411  
 Tate, A., 520, 522, 545, 576, 577  
 taxi, 53-54  
     automatizzato, 72  
 Taylor, W., 15, 204, 302  
 TD-GAMMON, 235, 243  
 TELL, 253-255, 283, 343, 354, 458  
 Teller, A., 172  
 Teller, E., 172  
 tempo, 462  
 tennis, 570, 574  
 teorema del limite centrale, 630  
 teorema di deduzione, 271  
 teorema di incompletezza, 13  
 teorema di risoluzione ground, 279, 384  
 teoremi, 326  
 teoria dei modelli, 342  
 teoria dei reticolii, 394  
     ottimo, 172  
 teoria del controllo, 23, 75, 172, 519  
 teoria dell'identità, 610  
 teoria della conferma, 11  
 teoria della specifica funzionale, 610  
 teoria delle decisioni, 15, 37  
 teoria sintattica (della conoscenza), 434  
 termine(in logica), 317  
 termini complessi, 340  
 Tesauro, G., 235, 244  
 test di terminazione, 211  
 test di Turing, 5, 8, 41  
     totale, 5  
 TEST-TERMINALE, 215, 224  
 testa (di una clausola), 367  
 Teukolsky, S.A., 172  
 Thagard, P., 41  
 Theo, 616  
 theory resolution, 400  
 Thielscher, M., 463  
 thingification, 433  
 Thompson, K., 241  
 Throop, T.A., 236  
 Thrun, S., 463

tic-tac-toe, 212, 240, 244

Tinsley, M., 235

tipi naturali, 415

TMS, v. sistemi di  
mantenimento della verità

Toffler, A., 605

Torras, C., 578

Touretzky, D.S., 466

traccia, 371

traduzione automatica, v.  
linguaggio, traduzione

transizione di fase, 302

transumanesimo, 608

trasparenza referenziale, 434

trattabilità dell'inferenza, 448

Traverso, P., 521, 579

Tsang, E., 205

TSP, v. problema del commesso  
viaggiatore

tuple, 313

Turco, automa, 239

Turing, A., 5, 13, 22, 24, 25, 42,  
70, 210, 240, 351, 396  
macchina di, 14  
test di, v. test di Turing

TV-IMPLICA?, 265, 266, 284, 292

TV-VERIFICA-TUTTO, 270

TWEAK, 520

## U

UCPOP, 521

uguaglianza (in logica), 323

Ullman, J.D., 398

ultraintelligenti, macchine, 607

unificatore più generale (MGU),  
354, 356, 385, 402

unificazione, 353-354, 395  
e uguaglianza, 385

Uniform Resource Locator  
(URL), 438

UNIFY, 353-354, 355, 367,  
372, 386

UNIFY-VAR, 355, 371

universo di Herbrand, 384, 396

UNPOP, 521

UOSAT-II, 577

URL, v. Uniform Resource

Locator

Urquhart, A., 462

Uskov, A.V., 241

UTILITÀ, 214, 215, 220

utilità, 69, 211

attesa, 76

funzione di, 69, 211

utilitarismo, 12

UWL, 579

## V

Vaessens, R. J.M., 577

validità, 270

VALORE, 147

valore atteso (in un albero di  
gioco), 228

valore meno vincolante, 187

VALORE-MAX, 215

VALORE-MIN, 215

van Beek, P., 203-205

van Benthem, J., 462

Van Emden, M.H., 399, 466

van Heijenoort, J., 401

Van Hentenryck, P., 204

Van Roy, P.L., 369, 372, 399

van Run, P., 186, 205

Varaiya, P., 74

Vardi, M.Y., 399, 465

variabile più vincolata, 362

variabili

ausiliarie (in un CSP), 183

in logica, 319

ordinamento, 187

Vazirani, U., 172

Vecchi, M.P., 172, 204

veicolo autonomo, 38

Vere, S.A., 576

verifica, 394

dei circuiti, 339

hardware, 339

in avanti, 188

verità, 259, 317

inferenza che preserva la, 262

tabella, 266, 301

vespa sphex, 52, 563

Vetterling, W.T., 172

vettore, 626

vincoli

binari, 183

di disuguaglianza, 500

di integrità, 280

di ordinamento, 494, 498

di preferenza, 183

di stato, 483, 514

grafo dei, 181, 199

memorizzazione dei, 204

non lineari, 181

propagazione dei, 188-192

speciali, 191

sulle risorse, 192

unari, 183

Vinge, V., 608

visione, 6, 28, 203

Visser, W., 394

VLSI, configurazione, 92, 118,

152

von Kempelen, W., 239

von Linne, C., 461

von Neumann, J., 15, 240

Vossen, T., 522

## W

Wadsworth, C.P., 342

Waldinger, R., 342, 400, 401,  
520

Walker, E., 39

WALKSAT, 195, 286-287, 288-  
290, 302, 454, 461, 516,  
518, 521

Walras, L., 15

Walsh, M.J., 173

Waltz, D., 28, 203

WAM, v. Warren Abstract  
Machine, 291, 399

Wang, E., 467

Wanner, E., 312

Warmuth, M., 118

WARPLAN, 520, 578

WARPLAN-C, 578

Warren Abstract Machine, 371,  
399

Warren, D. H.D., 369, 371,  
399, 520, 578

Warren, D.S., 399

Watson, J., 19

Watson, J.D., 157  
 Watt, J., 23  
 Wattenberg, M., 173  
 Webber, B.L., 41  
 Wefald, E.H., 171, 241  
 Wegman, M.N., 400  
 Weidenbach, C., 400  
 Weisler, S., 41  
 Weiss, G., 75, 580  
 Weizenbaum, J., 605  
 Weld, D.S., 75, 467, 521, 522,  
     577, 578, 606  
 Wellman, M.P., 16  
 Wells, P., 39  
 Wesley, L.P., 580  
 Westinghouse, 576  
 White, J.L., 394  
 Whitehead, A.N., 24, 396  
 Whiter, A.M., 576  
 Whorf, B., 312  
 Widrow, B., 29  
 Wiener, N., 23, 210, 240  
 Wilber, B.M., 400  
 Wilcox, B., 244  
 Wilensky, R., 33  
 Wilkins, D.E., 238, 576, 578  
 Williams, B., 75, 467, 577

Williamson, M., 579  
 Wilson, R.A., 7, 41, 611  
 Wilson, R.J., 203  
 Winker, S., 394, 400  
 Winograd, S., 29  
 Winograd, T., 29, 33, 397  
 Winston, P.H., 4, 28  
 Wittgenstein, L., 11, 262, 301,  
     415, 461  
 Wöhler, F., 595  
 Wojciechowski, W.S., 394  
 Wojcik, A.S., 394  
 Wood, D.E., 171  
 Wood, M.K., 172  
 Woods, W.A., 465  
 Wooldridge, M., 76  
 World Wide Web, 38, 437  
 Wos, L., 394, 400, 401  
 wrapper (per siti Internet), 442  
 Wright, O. and W., 6  
 Wright, S., 172-3  
 Wrightson, G., 401  
 wumpus, mondo del, 255-259,  
     269, 329-331, 409  
 Wundt, W., 19  
 Wygant, R.M., 397-8

**X**  
 Xcon, 365  
 XML, 462

**Y**  
 Yang, D., 577  
 Yang, Q., 522, 577  
 Yannakakis, M., 174, 203  
 Yap, R. H.C., 399  
 Yip, K. M.-K., 467  
 Yob, G., 303  
 Young, R.M., 577  
 Yvanovich, M., 577

**Z**  
 Z-3, 21  
 Zhang, L., 302  
 Zhang, W., 171  
 Zhao, Y., 302  
 Zhivotovsky, A.A., 241  
 Zhixing, C., 236  
 Zhou, R., 171  
 Zilberstein, S., 578, 579, 618  
 Zobrist, A.L., 244  
 zucchero sintattico, 327  
 Zuse, K., 21, 240  
 Zweben, M., 577