



INTELIGENCIA ARTIFICIAL

UN ENFOQUE MODERNO

Segunda edición

INTELIGÉNCIA ARTIFICIAL

UN ENFOQUE MODERNO

Segunda edición

Stuart J. Russell y Peter Norvig

Traducción:

Juan Manuel Corchado Rodríguez

Facultad de Ciencias

Universidad de Salamanca

**Fernando Martín Rubio, José Manuel Cadenas Figueredo,
Luis Daniel Hernández Molinero y Enrique Paniagua Arís**

Facultad de Informática

Universidad de Murcia

Raquel Fuentetaja Pinzán y Mónica Robledo de los Santos

Universidad Pontificia de Salamanca, campus Madrid

Ramón Rizo Aldeguer

Escuela Politécnica Superior

Universidad de Alicante

Revisión técnica:

Juan Manuel Corchado Rodríguez

Facultad de Ciencias

Universidad de Salamanca

Fernando Martín Rubio

Facultad de Informática

Universidad de Murcia

Andrés Castillo Sanz y María Luisa Díez Plata

Facultad de Informática

Universidad Pontificia de Salamanca, campus Madrid

Coordinación general de la traducción y revisión técnica:

Luis Joyanes Aguilar

Facultad de Informática

Universidad Pontificia de Salamanca, campus Madrid



Datos de catalogación bibliográfica

RUSSELL, S. J.; NORVIG, P.

INTELIGENCIA ARTIFICIAL. UN ENFOQUE MODERNO

Segunda edición

PEARSON EDUCACIÓN, S.A., Madrid, 2004

ISBN: 84-205-4003-X

Materia: Informática 681.3

Formato 195 × 250

Páginas: 1240

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual.

La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. Código Penal*).

DERECHOS RESERVADOS

© 2004 por PEARSON EDUCACIÓN, S.A.
Ribera del Loira, 28
28042 Madrid (España)

INTELIGENCIA ARTIFICIAL. UN ENFOQUE MODERNO. Segunda edición

RUSSELL, S. J.; NORVIG, P.

ISBN: 84-205-4003-X

Depósito Legal: M-26913-2004

PEARSON PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

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.

© 2003. All rights reserved.

No part or 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.

ISBN: 0-13-790395-2

Equipo editorial:

Editor: David Fayerman Aragón

Técnico editorial: Ana Isabel García Borro

Equipo de producción:

Director: José Antonio Clares

Técnico: José Antonio Hernán

Diseño de cubierta: Equipo de diseño de PEARSON EDUCACIÓN, S.A.

Composición: COPIBOOK, S.L.

Impreso por: Top Printer Plus

IMPRESO EN ESPAÑA - PRINTED IN SPAIN



Contenido

Prólogo

XIX

Sobre los autores

XXV

1 Introducción

1

1.1	¿Qué es la IA?	2
	Comportamiento humano: el enfoque de la Prueba de Turing	3
	Pensar como un humano: el enfoque del modelo cognitivo	3
	Pensamiento racional: el enfoque de las «leyes del pensamiento»	4
	Actuar de forma racional: el enfoque del agente racional	5
1.2	Los fundamentos de la inteligencia artificial	6
	Filosofía (desde el año 428 a.C. hasta el presente)	6
	Matemáticas (aproximadamente desde el año 800 al presente)	9
	Economía (desde el año 1776 hasta el presente)	11
	Neurociencia (desde el año 1861 hasta el presente)	12
	Psicología (desde el año 1879 hasta el presente)	14
	Ingeniería computacional (desde el año 1940 hasta el presente)	16
	Teoría de control y cibernetica (desde el año 1948 hasta el presente)	17
	Lingüística (desde el año 1957 hasta el presente)	18
1.3	Historia de la inteligencia artificial	19
	Génesis de la inteligencia artificial (1943-1955)	19
	Nacimiento de la inteligencia artificial (1956)	20
	Entusiasmo inicial, grandes esperanzas (1952-1969)	21
	Una dosis de realidad (1966-1973)	24
	Sistemas basados en el conocimiento: ¿clave del poder? (1969-1979)	26
	La IA se convierte en una industria (desde 1980 hasta el presente)	28
	Regreso de las redes neuronales (desde 1986 hasta el presente)	29
	IA se convierte en una ciencia (desde 1987 hasta el presente)	29
	Emergencia de los sistemas inteligentes (desde 1995 hasta el presente)	31
1.4	El estado del arte	32
1.5	Resumen	33
	Notas bibliográficas e históricas	34
	Ejercicios	35

2 Agentes inteligentes

37

2.1	Agentes y su entorno	37
2.2	Buen comportamiento: el concepto de racionalidad	40
	Medidas de rendimiento	40
	Racionalidad	41

Omnisciencia, aprendizaje y autonomía	42
2.3 La naturaleza del entorno	44
Especificación del entorno de trabajo	44
Propiedades de los entornos de trabajo	47
2.4 Estructura de los agentes	51
Programas de los agentes	51
Agentes reactivos simples	53
Agentes reactivos basados en modelos	55
Agentes basados en objetivos	57
Agentes basados en utilidad	58
Agentes que aprenden	59
2.5 Resumen	62
Notas bibliográficas e históricas	63
Ejercicios	65
3 Resolver problemas mediante búsqueda	67
3.1 Agentes resolventes-problemas	67
Problemas y soluciones bien definidos	70
Formular los problemas	71
3.2 Ejemplos de problemas	72
Problemas de juguete	73
Problemas del mundo real	76
3.3 Búsqueda de soluciones	78
Medir el rendimiento de la resolución del problema	80
3.4 Estrategias de búsqueda no informada	82
Búsqueda primero en anchura	82
Búsqueda de costo uniforme	84
Búsqueda primero en profundidad	85
Búsqueda de profundidad limitada	87
Búsqueda primero en profundidad con profundidad iterativa	87
Búsqueda bidireccional	89
Comparación de las estrategias de búsqueda no informada	91
3.5 Evitar estados repetidos	91
3.6 Búsqueda con información parcial	94
Problemas sin sensores	95
Problemas de contingencia	96
3.7 Resumen	97
Notas bibliográficas e históricas	98
Ejercicios	100
4 Búsqueda informada y exploración	107
4.1 Estrategias de búsqueda informada (heurísticas)	107
Búsqueda voraz primero el mejor	108
Búsqueda A*: minimizar el costo estimado total de la solución	110
Búsqueda heurística con memoria acotada	115
Aprender a buscar mejor	118
4.2 Funciones heurísticas	119
El efecto de la precisión heurística en el rendimiento	120
Inventar funciones heurísticas admisibles	121
Aprendizaje de heurísticas desde la experiencia	124
4.3 Algoritmos de búsqueda local y problemas de optimización	125

Búsqueda de ascensión de colinas	126
Búsqueda de temple simulado	129
Búsqueda por haz local	131
Algoritmos genéticos	131
4.4 Búsqueda local en espacios continuos	136
4.5 Agentes de búsqueda <i>online</i> y ambientes desconocidos	138
Problemas de búsqueda en línea (<i>online</i>)	138
Agentes de búsqueda en línea (<i>online</i>)	141
Búsqueda local en línea (<i>online</i>)	142
Aprendizaje en la búsqueda en línea (<i>online</i>)	144
4.6 Resumen	145
Notas bibliográficas e históricas	146
Ejercicios	151
5 Problemas de satisfacción de restricciones	155
5.1 Problemas de satisfacción de restricciones	155
5.2 Búsqueda con vuelta atrás para PSR	159
Variable y ordenamiento de valor	162
Propagación de la información a través de las restricciones	163
Comprobación hacia delante	163
Propagación de restricciones	164
Manejo de restricciones especiales	166
Vuelta atrás inteligente: mirando hacia atrás	167
5.3 Búsqueda local para problemas de satisfacción de restricciones	169
5.4 La estructura de los problemas	171
5.5 Resumen	175
Notas bibliográficas e históricas	176
Ejercicios	178
6 Búsqueda entre adversarios	181
6.1 Juegos	181
6.2 Decisiones óptimas en juegos	183
Estrategias óptimas	183
El algoritmo minimax	185
Decisiones óptimas en juegos multi-jugador	186
6.3 Poda alfa-beta	188
6.4 Decisiones en tiempo real imperfectas	191
Funciones de evaluación	192
Corte de la búsqueda	194
6.5 Juegos que incluyen un elemento de posibilidad	196
Evaluación de la posición en juegos con nodos de posibilidad	198
Complejidad del minimaxesperado	199
Juegos de cartas	200
6.6 Programas de juegos	202
6.7 Discusión	205
6.8 Resumen	207
Notas bibliográficas e históricas	208
Ejercicios	212
7 Agentes lógicos	217
7.1 Agentes basados en conocimiento	219
7.2 El mundo de <i>wumpus</i>	221

7.3	Lógica	224
7.4	Lógica proposicional: una lógica muy sencilla	229
	Sintaxis	229
	Semántica	230
	Una base de conocimiento sencilla	233
	Inferencia	233
	Equivalecia, validez y <i>satisfacibilidad</i>	235
7.5	Patrones de razonamiento en lógica proposicional	236
	Resolución	239
	Forma normal conjuntiva	241
	Un algoritmo de resolución	242
	Compleitud de la resolución	243
	Encadenamiento hacia delante y hacia atrás	244
7.6	Inferencia proposicional efectiva	248
	Un algoritmo completo con <i>backtracking</i> («vuelta atrás»)	248
	Algoritmos de búsqueda local	249
	Problemas duros de <i>satisfacibilidad</i>	251
7.7	Agentes basados en lógica proposicional	253
	Encontrar hoyos y <i>wumpus</i> utilizando la inferencia lógica	253
	Guardar la pista acerca de la localización y la orientación del agente	255
	Agentes basados en circuitos	256
	Una comparación	260
7.8	Resumen	261
	Notas bibliográficas e históricas	262
	Ejercicios	266
8	Lógica de primer orden	271
8.1	Revisión de la representación	271
8.2	Sintaxis y semántica de la lógica de primer orden	277
	Modelos en lógica de primer orden	277
	Símbolos e interpretaciones	278
	Términos	280
	Sentencias atómicas	281
	Sentencias compuestas	281
	Cuantificadores	281
	Cuantificador universal (\forall)	282
	Cuantificación existencial (\exists)	283
	Cuantificadores anidados	284
	Conexiones entre \forall y \exists	285
	Igualdad	286
8.3	Utilizar la lógica de primer orden	287
	Aserciones y peticiones en lógica de primer orden	287
	El dominio del parentesco	288
	Números, conjuntos y listas	290
	El mundo de <i>wumpus</i>	292
8.4	Ingeniería del conocimiento con lógica de primer orden	295
	El proceso de ingeniería del conocimiento	296
	El dominio de los circuitos electrónicos	297
	Identificar la tarea	298
	Recopilar el conocimiento relevante	298
	Decidir el vocabulario	299

Codificar el conocimiento general del dominio	300
Codificar la instancia del problema específico	300
Plantear peticiones al procedimiento de inferencia	301
Depurar la base de conocimiento	301
8.5 Resumen	302
Notas bibliográficas e históricas	303
Ejercicios	304
9 Inferencia en lógica de primer orden	309
9.1 Lógica proposicional vs. Lógica de primer orden	310
Reglas de inferencia para cuantificadores	310
Reducción a la inferencia proposicional	311
9.2 Unificación y sustitución	312
Una regla de inferencia de primer orden	313
Unificación	314
Almacenamiento y recuperación	315
9.3 Encadenamiento hacia delante	318
Cláusulas positivas de primer orden	318
Un algoritmo sencillo de encadenamiento hacia delante	320
Encadenamiento hacia delante eficiente	322
Emparejar reglas con los hechos conocidos	322
Encadenamiento hacia delante incremental	324
Hechos irrelevantes	326
9.4 Encadenamiento hacia atrás	326
Un algoritmo de encadenamiento hacia atrás	327
Programación lógica	328
Implementación eficiente de programas lógicos	330
Inferencia redundante y bucles infinitos	332
Programación lógica con restricciones	334
9.5 Resolución	335
Formas normales conjuntivas en lógica de primer orden	336
La regla de inferencia de resolución	338
Demostraciones de ejemplo	338
Completilud de la resolución	338
Manejar la igualdad	341
Estrategias de resolución	345
Resolución unitaria	346
Resolución mediante conjunto soporte	346
Resolución lineal	347
Subsunción	347
Demostradores de teoremas	348
Diseño de un demostrador de teoremas	348
Ampliar el Prolog	348
Demostradores de teoremas como asistentes	349
Usos prácticos de los demostradores de teoremas	350
9.6 Resumen	351
Notas bibliográficas e históricas	352
Ejercicios	353
10 Representación del conocimiento	363
10.1 Ingeniería ontológica	363
10.2 Categoría y objetos	366

Objetos compuestos	368
Medidas	369
Sustancias y objetos	371
10.3 Acciones, situaciones y eventos	373
La ontología del cálculo de situaciones	373
Descripción de acciones en el cálculo de situaciones	375
Resolver el problema de la representación del marco	377
Resolver el problema de la inferencia del marco	379
El tiempo y el cálculo de eventos	380
Eventos generalizados	381
Procesos	383
Intervalos	384
Flujos y objetos	386
10.4 Eventos mentales y objetos mentales	387
Una teoría formal de creencias	387
Conocimiento y creencia	389
Conocimiento, tiempo y acción	390
10.5 El mundo de la compra por Internet	391
Comparación de ofertas	395
10.6 Sistemas de razonamiento para categorías	397
Redes semánticas	397
Lógica descriptiva	401
10.7 Razonamiento con información por defecto	402
Mundos abiertos y cerrados	403
Negación como fallo y semánticas de modelado estables	405
Circunscripción y lógica por defecto	406
10.8 Sistemas de mantenimiento de verdad	409
10.9 Resumen	411
Notas bibliográficas e históricas	412
Ejercicios	419
11 Planificación	427
11.1 El problema de planificación	428
El lenguaje de los problemas de planificación	429
Expresividad y extensiones	431
Ejemplo: transporte de carga aéreo	433
Ejemplo: el problema de la rueda de recambio	434
Ejemplo: el mundo de los bloques	434
11.2 Planificación con búsquedas en espacios de estado	436
Búsquedas hacia-delante en el espacio de estados	436
Búsquedas hacia-atrás en el espacio de estados	438
Heurísticas para la búsqueda en el espacio de estados	439
11.3 Planificación ordenada parcialmente	441
Ejemplo de planificación de orden parcial	445
Planificación de orden parcial con variables independientes	448
Heurísticas para planificación de orden parcial	449
11.4 Grafos de planificación	450
Grafos de planificación para estimación de heurísticas	453
El algoritmo GRAPHPLAN	454
Interrupción de GRAPHPLAN	457
11.5 Planificación con lógica proposicional	458

Descripción de problemas de planificación en lógica proposicional	458
Complejidad de codificaciones proposicionales	462
11.6 Análisis de los enfoques de planificación	463
11.7 Resumen	465
Notas bibliográficas e históricas	466
Ejercicios	469
12 Planificación y acción en el mundo real	475
12.1 Tiempo, planificación y recursos	475
Programación con restricción de recursos	478
12.2 Redes de planificación jerárquica de tareas	481
Representación de descomposición de acciones	482
Modificación de planificadores para su descomposición	484
Discusión	487
12.3 Planificación y acción en dominios no deterministas	490
12.4 Planificación condicional	493
Planificación condicional en entornos completamente observables	493
Planificación condicional en entornos parcialmente observables	498
12.5 Vigilancia de ejecución y replanificación	502
12.6 Planificación continua	507
12.7 Planificación multiagente	512
Cooperación: planes y objetivos conjuntos	512
Planificación condicional en entornos parcialmente observables	514
Mecanismos de coordinación	515
Mecanismos de coordinación	517
12.8 Resumen	517
Notas bibliográficas e históricas	518
Ejercicios	522
13 Incertidumbre	527
13.1 Comportamiento bajo incertidumbre	527
Manipulación del conocimiento incierto	528
Incertidumbre y decisiones racionales	530
Diseño de un agente de decisión teórico	531
13.2 Notación básica con probabilidades	532
Proposiciones	532
Sucesos atómicos	534
Probabilidad priori	534
Probabilidad condicional	536
13.3 Los axiomas de la probabilidad	537
Utilización de los axiomas de probabilidad	539
Por qué los axiomas de la probabilidad son razonables	540
13.4 Inferencia usando las distribuciones conjuntas totales	541
13.5 Independencia	544
13.6 La Regla de Bayes y su uso	546
Aplicación de la regla de Bayes: el caso sencillo	547
Utilización de la regla de Bayes: combinación de evidencia	548
13.7 El mundo <i>wumpus</i> revisado	550
13.8 Resumen	554
Notas bibliográficas e históricas	555
Ejercicios	557

14 Razonamiento probabilista	561
14.1 La representación del conocimiento en un dominio incierto	561
14.2 La semántica de las redes bayesianas	564
La representación de la distribución conjunta completa	564
Un método para la construcción de redes bayesianas	565
Compactación y ordenación de nodos	566
Relaciones de independencia condicional en redes bayesianas	568
14.3 Representación eficiente de las distribuciones condicionales	569
Redes bayesianas con variables continuas	571
14.4 Inferencia exacta en redes bayesianas	574
Inferencia por enumeración	575
El algoritmo de eliminación de variables	577
La complejidad de la inferencia exacta	580
Algoritmos basados en grupos	580
14.5 Inferencia aproximada en redes bayesianas	581
Métodos de muestreo directo	582
Muestreo por rechazo en redes bayesianas	583
Ponderación de la verosimilitud	585
Inferencia por simulación en cadenas de Markov	587
14.6 Extensión de la probabilidad a representaciones de primer orden	590
14.7 Otros enfoques al razonamiento con incertidumbre	595
Métodos basados en reglas para razonamiento con incertidumbre	596
Representación de la ignorancia: teoría de Dempster-Shafer	598
Representación de la vaguedad: conjuntos difusos y lógica difusa	599
14.8 Resumen	601
Notas bibliográficas e históricas	601
Ejercicios	606
15 Razonamiento probabilista en el tiempo	611
15.1 El tiempo y la incertidumbre	611
Estados y observaciones	612
Procesos estacionarios e hipótesis de Markov	613
15.2 Inferencia en modelos temporales	616
Filtrado y predicción	617
Suavizado	619
Encontrar la secuencia más probable	622
15.3 Modelos ocultos de Markov	624
Algoritmos matriciales simplificados	624
15.4 Filtros de Kalman	627
Actualización de distribuciones gaussianas	628
Un ejemplo unidimensional sencillo	629
El caso general	632
Aplicabilidad del filtrado de Kalman	633
15.5 Redes bayesianas dinámicas	635
Construcción de RBDs	636
Inferencia exacta en RBDs	640
Inferencia aproximada en RBDs	641
15.6 Reconocimiento del habla	645
Sonidos del habla	647
Palabras	649
Oraciones	651

Construcción de un reconocedor del habla	654
15.7 Resumen	656
Notas bibliográficas e históricas	656
Ejercicios	659
16 Toma de decisiones sencillas	663
16.1 Combinación de creencias y deseos bajo condiciones de incertidumbre	664
16.2 Los fundamentos de la teoría de la utilidad	665
Restricciones sobre preferencias racionales	666
... y entonces apareció la utilidad	668
16.3 Funciones de utilidad	669
La utilidad del dinero	669
Escalas de utilidad y evaluación de la utilidad	671
16.4 Funciones de utilidad multiatributo	674
Predominio	674
Estructura de preferencia y utilidad multiatributo	677
Preferencias sin incertidumbre	677
Preferencias con incertidumbre	678
16.5 Redes de decisión	679
Representación de un problema de decisión mediante una red de decisión	679
Evaluación en redes de decisión	681
16.6 El valor de la información	682
Un ejemplo sencillo	682
Una fórmula general	683
Propiedades del valor de la información	685
Implementación de un agente recopilador de información	685
16.7 Sistemas expertos basados en la teoría de la decisión	686
16.8 Resumen	690
Notas bibliográficas e históricas	690
Ejercicios	692
17 Toma de decisiones complejas	697
17.1 Problemas de decisión secuenciales	698
Un ejemplo	698
Optimalidad en problemas de decisión secuenciales	701
17.2 Iteración de valores	704
Utilidades de los estados	704
El algoritmo de iteración de valores	705
Convergencia de la iteración de valores	707
17.3 Iteración de políticas	710
17.4 Procesos de decisión de Markov parcialmente observables	712
17.5 Agentes basados en la teoría de la decisión	716
17.6 Decisiones con varios agentes: teoría de juegos	719
17.7 Diseño de mecanismos	729
17.8 Resumen	732
Notas bibliográficas e históricas	733
Ejercicios	736
18 Aprendizaje de observaciones	739
18.1 Formas de aprendizaje	739
18.2 Aprendizaje inductivo	742

18.3	Aprender árboles de decisión	744
	Árboles de decisión como herramienta de desarrollo	744
	Expresividad de los árboles de decisión	745
	Inducir árboles de decisión a partir de ejemplos	746
	Elección de los atributos de test	750
	Valoración de la calidad del algoritmo de aprendizaje	752
	Ruido y sobreajuste	753
	Extensión de la aplicabilidad de los árboles de decisión	755
18.4	Aprendizaje de conjuntos de hipótesis	756
18.5	¿Por qué funciona el aprendizaje?: teoría computacional del aprendizaje	760
	¿Cuántos ejemplos se necesitan?	761
	Aprendizaje de listas de decisión	763
	Discusión	765
18.6	Resumen	766
	Notas bibliográficas e históricas	767
	Ejercicios	769
19	Conocimiento en el aprendizaje	773
19.1	Una formulación lógica del aprendizaje	773
	Ejemplos e hipótesis	774
	Búsqueda mejor-hipótesis-actual	776
	Búsqueda de mínimo compromiso	778
19.2	Conocimiento en el aprendizaje	782
	Algunos ejemplos sencillos	784
	Algunos esquemas generales	784
19.3	Aprendizaje basado en explicaciones	786
	Extraer reglas generales a partir de ejemplos	787
	Mejorar la eficiencia	789
19.4	Aprendizaje basado en información relevante	791
	Determinar el espacio de hipótesis	792
	Aprender y utilizar información relevante	792
19.5	Programación lógica inductiva	795
	Un ejemplo	795
	Métodos de aprendizaje inductivo de arriba a abajo (<i>Top-down</i>)	798
	Aprendizaje inductivo con deducción inversa	801
	Hacer descubrimientos con la programación lógica inductiva	803
18.6	Resumen	805
	Notas bibliográficas e históricas	806
	Ejercicios	809
20	Métodos estadísticos de aprendizaje	811
20.1	Aprendizaje estadístico	811
20.2	Aprendizaje con datos completos	815
	Aprendizaje del parámetro de máxima verosimilitud: modelos discretos	815
	Modelos de Bayes simples (Naive Bayes)	818
	Aprendizaje de parámetros de máxima verosimilitud: modelos continuos	819
	Aprendizaje de parámetros Bayésiano	821
	Aprendizaje de la estructura de las redes bayesianas	823
20.3	Aprendizaje con variables ocultas: el algoritmo EM	825
	Agrupamiento no supervisado: aprendizaje de mezclas de gaussianas	826
	Aprendizaje de redes bayesianas con variables ocultas	829

Aprendizaje de modelos de Markov ocultos	831
Forma general del algoritmo EM	832
Aprendizaje de la estructura de las redes de Bayes con variables ocultas	833
20.4 Aprendizaje basado en instancias	834
Modelos de vecinos más cercanos	835
Modelos núcleo	837
20.5 Redes neuronales	838
Unidades en redes neuronales	839
Estructuras de las redes	840
Redes neuronales de una sola capa con alimentación-hacia-delante (perceptrones) .	842
Redes neuronales multicapa con alimentación hacia delante	846
Aprendizaje de la estructura de las redes neuronales	851
20.6 Máquinas núcleo	851
20.7 Caso de estudio: reconocedor de dígitos escritos a mano	855
20.8 Resumen	857
Notas bibliográficas e históricas	859
Ejercicios	863
21 Aprendizaje por refuerzo	867
21.1 Introducción	867
21.2 Aprendizaje por refuerzo pasivo	869
Estimación directa de la utilidad	870
Programación dinámica adaptativa	871
Aprendizaje de diferencia temporal	872
21.3 Aprendizaje por refuerzo activo	876
Exploración	876
Aprendizaje de una Función Acción-Valor	880
21.4 Generalización en aprendizaje por refuerzo	882
Aplicaciones a juegos	885
Aplicación a control de robots	886
21.5 Búsqueda de la política	887
21.6 Resumen	890
Notas bibliográficas e históricas	891
Ejercicios	894
22 La comunicación	897
22.1 La comunicación como acción	898
Fundamentos del lenguaje	899
Etapas de la comunicación	900
22.2 Una gramática formal para un fragmento del español	903
El léxico de e0	904
La Gramática de e0	904
22.3 Análisis sintáctico	905
Análisis sintáctico eficiente	908
22.4 Gramáticas aumentadas	914
Subcategorización del verbo	916
Capacidad generativa de las gramáticas aumentadas	918
22.5 Interpretación semántica	919
La semántica de un fragmento en español	920
Tiempo y forma verbal	921
Cuantificación	922

Interpretación pragmática	925
Generación de lenguajes con DCGs	926
22.6 Ambigüedad y desambigüedad	927
Desambiguación	929
22.7 Comprensión del discurso	930
Resolución por referencia	931
La estructura de un discurso coherente	932
22.8 Inducción gramatical	934
22.9 Resumen	936
Notas bibliográficas e históricas	937
Ejercicios	941
23 Procesamiento probabilístico del lenguaje	945
23.1 Modelos probabilísticos del lenguaje	945
Gramáticas probabilísticas independientes del contexto	949
Aprendizaje de probabilidades para PCFGs	950
Aprendizaje de la estructura de las reglas para PCFGs	951
23.2 Recuperación de datos	952
Evaluación de los Sistemas de RD	955
Refinamientos RD	956
Presentación de los conjuntos de resultados	957
Implementar sistemas RD	959
23.3 Extracción de la información	961
23.4 Traducción automática	964
Sistemas de traducción automáticos	966
Traducción automática estadística	967
Probabilidades de aprendizaje para la traducción automática	970
23.5 Resumen	972
Notas bibliográficas e históricas	972
Ejercicios	975
24 Percepción	979
24.1 Introducción	979
24.2 Formación de la imagen	981
Imágenes sin lentes: la cámara de orificio o pinhole	982
Sistemas de lentes	983
Luz: la fotometría de la formación de imágenes	983
Color: la espectrofotometría de la formación de imágenes	985
24.3 Operaciones de procesamiento de imagen a bajo nivel	986
Detección de aristas	987
Segmentación de la imagen	990
24.4 Extracción de información tridimensional	991
Movimiento	993
Estereoscopía binocular	996
Gradientes de textura	997
Sombreado	999
Contorno	1000
24.5 Reconocimiento de objetos	1004
Reconocimiento basado en la intensidad	1007
Reconocimiento basado en las características	1008
Estimación de postura	1010

24.6 Empleo de la visión para la manipulación y navegación	1012
24.7 Resumen	1014
Notas bibliográficas e históricas	1015
Ejercicios	1018
25 Robótica	1023
25.1 Introducción	1023
25.2 <i>Hardware</i> robótico	1025
Sensores	1025
Efectores	1027
25.3 Percepción robótica	1029
Localización	1031
Generación de mapas	1036
Otros tipos de percepción	1039
25.4 Planear el movimiento	1039
Espacio de configuración	1040
Métodos de descomposición en celdas	1043
Métodos de esqueletización	1046
25.5 Planificar movimientos inciertos	1047
Métodos robustos	1048
25.6 Movimiento	1051
Dinámica y control	1051
Control del campo de potencial	1054
Control reactivo	1055
25.7 Arquitecturas <i>software</i> robóticas	1057
Arquitectura de subsumpción	1058
Arquitectura de tres capas	1059
Lenguajes de programación robóticos	1060
25.8 Dominios de aplicación	1061
25.9 Resumen	1064
Notas bibliográficas e históricas	1065
Ejercicios	1069
26 Fundamentos filosóficos	1075
26.1 IA débil: ¿pueden las máquinas actuar con inteligencia?	1075
El argumento de incapacidad	1077
La objeción matemática	1078
El argumento de la informalidad	1079
26.2 IA fuerte: ¿pueden las máquinas pensar de verdad?	1081
El problema de mente-cuerpo	1084
El experimento del «cerebro en una cubeta»	1085
El experimento de la prótesis cerebral	1086
La habitación china	1088
26.3 La ética y los riesgos de desarrollar la Inteligencia Artificial	1090
26.4 Resumen	1095
Notas bibliográficas e históricas	1095
Ejercicios	1098
27 IA: presente y futuro	1099
27.1 Componentes de los agentes	1099
27.2 Arquitecturas de agentes	1102

27.3	¿Estamos llevando la dirección adecuada?	1104
27.4	¿Qué ocurriría si la IA tuviera éxito?	1106
A	Fundamentos matemáticos	1109
A.1	Análisis de la complejidad y la notación O()	1109
	Análisis asintótico	1109
	Los problemas inherentemente difíciles y NP	1110
A.2	Vectores, matrices y álgebra lineal	1112
A.3	Distribuciones de probabilidades	1114
	Notas bibliográficas e históricas	1115
B	Notas sobre lenguajes y algoritmos	1117
B.1	Definición de lenguajes con Backus-Naur Form (BNF)	1117
B.2	Algoritmos de descripción en pseudocódigo	1118
B.3	Ayuda en línea	1119
	Bibliografía	1121
	Índice alfabético	1179

Prólogo

La Inteligencia Artificial (IA) es un campo grande (enorme), y este libro también. He intentado explorarlo con plena profundidad acompañándolo constantemente de lógica, probabilidad y matemáticas; de percepción, razonamiento, aprendizaje y acción, es decir, de todo lo que procede de los dispositivos microelectrónicos hasta los exploradores del planetario de la robótica. Otra razón para que este libro se pueda considerar espléndido es la profundidad en la presentación de los resultados, aunque nos hayamos esforzado por abarcar sólo las ideas más centrales en la parte principal de cada capítulo. En las notas bibliográficas al final de cada capítulo se proporcionan consejos para promover resultados.

El subtítulo de este libro es «Un Enfoque Moderno». La intención de esta frase bastante vacía es el hecho de que hemos intentado sintetizar lo que se conoce ahora dentro de un marco de trabajo común, en vez de intentar explicar cada uno de los subcampos de la IA dentro de su propio contexto histórico. Nos disculpamos ante aquellos cuyos subcampos son, como resultado, menos reconocibles de lo que podrían haber sido de cualquier otra forma.

El principal tema unificador es la idea del **agente inteligente**. Definimos la IA como el estudio de los agentes que reciben percepciones del entorno y llevan a cabo las acciones. Cada agente implementa una función la cual estructura las secuencias de las percepciones en acciones; también tratamos las diferentes formas de representar estas funciones, tales como sistemas de producción, agentes reactivos, planificadores condicionales en tiempo real, redes neurales y sistemas teóricos para las decisiones. Explicaremos el papel del aprendizaje cuando alcanza al diseñador y cómo se introduce en entornos desconocidos, mostrando también cómo ese papel limita el diseño del agente, favoreciendo así la representación y el razonamiento explícitos del conocimiento. Trataremos la robótica y su visión no como problemas con una definición independiente, sino como algo que ocurre para lograr los objetivos. Daremos importancia al entorno de las tareas al determinar el diseño apropiado de los agentes.

Nuestro objetivo principal es el de transmitir las *ideas* que han surgido durante los últimos 50 años de investigación en IA y trabajos afines durante los dos últimos milenios. Hemos intentado evitar una excesiva formalidad en la presentación de estas ideas a la vez que hemos intentado cuidar la precisión. Siempre que es necesario y adecuado, incluimos algoritmos en pseudo código para concretar las ideas, lo que se describe en el Apéndice B. Las implementaciones en varios lenguajes de programación están disponibles en el sitio Web de este libro, en la dirección de Internet aima.cs.berkeley.edu.

Este libro se ha pensado principalmente para utilizarse en un curso de diplomatura o en una serie de varios cursos. También se puede utilizar en un curso de licenciatura (quizás acompañado de algunas de las fuentes primarias, como las que se sugieren en las notas bibliográficas). Debida a la extensa cobertura y a la gran cantidad de algoritmos detallados, también es una herramienta útil como manual de referencia primario para alumnos de licenciatura superior y profesionales que deseen abarcar más allá de su propio subcampo. El único prerequisito es familiarizarse con los conceptos básicos de la ciencia de la informática (algoritmos, estructuras de datos, complejidad) a un nivel de aprendizaje de segundo año. El cálculo de Freshman es útil para entender las redes neuronales y el aprendizaje estadístico en detalle. En el Apéndice A se ofrecen los fundamentos matemáticos necesarios.

Visión general del libro

El libro se divide en ocho partes. La Parte I, **Inteligencia Artificial**, ofrece una visión de la empresa de IA basado en la idea de los agentes inteligentes, sistemas que pueden decidir qué hacer y entonces actuar. La Parte II, **Resolución de Problemas**, se concentra en los métodos para decidir qué hacer cuando se planean varios pasos con antelación, por ejemplo, navegar para cruzar un país o jugar al ajedrez. La Parte III, **Conocimiento y Razonamiento**, abarca las formas de representar el conocimiento sobre el mundo, es decir, cómo funciona, qué aspecto tiene actualmente y cómo podría actuar, y estudia también cómo razonar de forma lógica con ese conocimiento. La Parte IV, **Planificación**, abarca cómo utilizar esos métodos de razonamiento para decidir qué hacer, particularmente construyendo *planes*. La Parte V, **Conocimiento y Razonamiento Inciertos**, se asemeja a las Partes III y IV, pero se concentra en el razonamiento y en la toma de decisiones en presencia de la *incertidumbre* del mundo, la forma en que se podría enfrentar, por ejemplo, a un sistema de tratamiento y diagnosis médicos.

Las Partes II y V describen esa parte del agente inteligente responsable de alcanzar las decisiones. La Parte IV, **Aprendizaje**, describe los métodos para generar el conocimiento requerido por los componentes de la toma de decisiones. La Parte VII, **Comunicación, Percepción y Actuación**, describe las formas en que un agente inteligente puede percibir su entorno para saber lo que está ocurriendo, tanto si es mediante la visión, el tacto, el oído o el entendimiento del idioma; también describe cómo se pueden transformar los planes en acciones reales, o bien en movimientos de un robot, o bien en órdenes del lenguaje natural. Finalmente, la Parte VIII, **Conclusiones**, analiza el pasado y el futuro de la IA y sus repercusiones filosóficas y éticas.

Cambios de la primera edición

Desde que en 1995 se publicó la primera edición, la IA ha sufrido muchos cambios, por lo tanto esta edición también ha sufrido muchos cambios. Se han vuelto a escribir todos los capítulos elocuentemente para reflejar los últimos trabajos de este campo, reinterpretar los trabajos anteriores para que haya mayor coherencia con los actuales y mejorar la dirección pedagógica de las ideas. Los seguidores de la IA deberían estar alentados, ya que las técnicas actuales son mucho más prácticas que las del año 1995; por ejemplo, los algoritmos de planificación de la primera edición podrían generar planes sólo de unos cuantos pasos, mientras que los algoritmos de esta edición superan los miles de pasos. En la deducción probalística se han visto mejoras similares de órdenes de magnitud, procesamiento de lenguajes y otros subcampos. A continuación se muestran los cambios más destacables de esta edición:

- En la Parte I hacemos un reconocimiento de las contribuciones históricas a la teoría de controles, teoría de juegos, economía y neurociencia. Esto ayudará a sintonizar con una cobertura más integrada de estas ideas en los siguientes capítulos.
- En la Parte II se estudian los algoritmos de búsqueda en línea, y se ha añadido un capítulo nuevo sobre la satisfacción de las limitaciones. Este último proporciona una conexión natural con el material sobre la lógica.
- En la Parte III la lógica proposicional, que en la primera edición, se presentó como un peldaño a la lógica de primer orden, ahora se presenta como un lenguaje de representación útil por propio derecho, con rápidos algoritmos de deducción y diseños de agentes basados en circuitos. Los capítulos sobre la lógica de primer orden se han reorganizado para presentar el material de forma más clara, añadiendo el ejemplo del dominio de compras en Internet.
- En la Parte IV incluimos los métodos de planificación más actuales, tales como GRAPHPLAN y la planificación basada en la *satisfabilidad*, e incrementamos la cobertura de la planificación temporal, planificación condicional, planificación jerárquica y planificación multiagente.
- En la Parte V hemos aumentado el material de redes Bayesianas con algoritmos nuevos, tales como la eliminación de variables y el algoritmo Monte Carlo de cadena Markov; también hemos creado un capítulo nuevo sobre el razonamiento temporal incierto, abarcando los modelos ocultos de Markov, los filtros Kalman y las redes dinámicas Bayesianas. Los procesos de decisión de Markov se estudian en profundidad, añadiendo también secciones de teoría de juegos y diseños de mecanismos.
- En la Parte VI combinamos trabajos de aprendizaje estadístico, simbólico y neural, y añadimos secciones para impulsar los algoritmos, el algoritmo EM, aprendizaje basado en instancias, y métodos kernel «de núcleo» (soporte a máquinas vectoriales).
- En la Parte VII el estudio del procesamiento de lenguajes añade secciones sobre el procesamiento del discurso y la inducción gramatical, así como un capítulo so-

bre los modelos de lenguaje probabilístico, con aplicaciones en la recuperación de información y en la traducción automática. El estudio de la robótica enfatiza la integración de datos inciertos de sensores, y el capítulo sobre la visión actualiza el material sobre el reconocimiento de objetos.

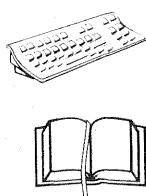
- En la Parte VIII introducimos una sección sobre las repercusiones éticas de la IA.

Utilización del libro

27 capítulos componen la totalidad del libro, que a su vez están compuestos por sesiones para clases que pueden durar una semana, por tanto para completar el libro se pueden necesitar hasta dos semestres. Alternativamente, se puede adaptar un curso de forma que se adecue a los intereses del instructor o del alumno. Si se utiliza el libro completo, servirá como soporte para cursos tanto si son reducidos, de introducción, para diplomaturas o para licenciados e ingenieros, de especialización, en temas avanzados. En la página Web aima.cs.berkeley.edu, se ofrecen programas de muestra recopilados de más de 600 escuelas y facultades, además de sugerencias que ayudarán a encontrar la secuencia más adecuada para sus necesidades.

Se han incluido 385 ejercicios que requieren una buena programación, y se han marcado con el icono de un teclado. Estos ejercicios se pueden resolver utilizando el repositorio de códigos que se encuentra en la página Web, aima.cs.berkeley.edu. Algunos de ellos son tan grandes que se pueden considerar proyectos de fin de trimestre. Algunos ejercicios requieren consultar otros libros de texto para investigar y se han marcado con el icono de un libro.

Los puntos importantes también se han marcado con un ícono que indica puntos importantes. Hemos incluido un índice extenso de 10.000 elementos que facilitan la utilización del libro. Además, siempre que aparece un **Término nuevo**, también se destaca en el margen.



TÉRMINO NUEVO

Utilización de la página Web

En la página Web se puede encontrar:

- implementaciones de los algoritmos del libro en diferentes lenguajes de programación,
- una relación de aproximadamente 600 universidades y centros, que han utilizado este libro, muchos de ellos con enlaces a los materiales de cursos en la red,
- una relación de unos 800 enlaces a sitios Web con contenido útil sobre la IA,
- una lista de todos los capítulos, uno a uno, material y enlaces complementarios,
- instrucciones sobre cómo unirse a un grupo de debate sobre el libro,
- instrucciones sobre cómo contactar con los autores mediante preguntas y comentarios,

- instrucciones sobre cómo informar de los errores del libro, en el caso probable de que existieran, y
- copias de las figuras del libro, junto con diapositivas y otro tipo de material para profesores.

Agradecimientos

Jitendra Malik ha escrito la mayor parte del Capítulo 24 (sobre la visión). Gran parte del Capítulo 25 (sobre robótica) de esta edición ha sido escrita por Sebastián Thrun, mientras que la primera edición fue escrita por John Canny. Doug Edwards investigó las notas históricas de la primera edición. Tim Huang, Mark Paskin y Cinthia Bruyns ayudó a realizar el formato de los diagramas y algoritmos. Alan Apt, Sondra Chavez, Toni Holm, Jake Warde, Irwin Zucker y Camille Trentacoste de Prentice Hall hicieron todo lo que pudieron para cumplir con el tiempo de desarrollo y aportaron muchas sugerencias sobre el diseño y el contenido del libro.

Stuart quiere agradecer a sus padres el continuo apoyo y aliento demostrado, y a su esposa, Loy Sheflott su gran paciencia e infinita sabiduría. También desea que Gordon y Lucy puedan leer esta obra muy pronto. Y a RUGS (Russell's Unusual Group of Students) por haber sido de verdad de gran ayuda.

Peter quiere agradecer a sus padres (Torsten y Gerda) el ánimo que le aportaron para impulsarle a empezar el proyecto, y a su mujer Kris, hijos y amigos por animarle y tolerar todas las horas que ha dedicado en escribir y en volver a escribir su trabajo.

Estamos en deuda con los bibliotecarios de las universidades de Berkeley, Stanford, y con el MIT y la NASA; y con los desarrolladores de CiteSeer y Google por haber revolucionado la forma en que hemos realizado gran parte de la investigación.

Obviamente, nos es imposible agradecer a todas las personas que han utilizado el libro y que han realizado sugerencias, sin embargo nos gustaría mostrar especial agradecimiento a los comentarios que han realizado personas como 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 Grahe, 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 Pijls, 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 Zilberstein, y los revisores anónimos proporcionados por Prentice Hall.

Acerca de la portada

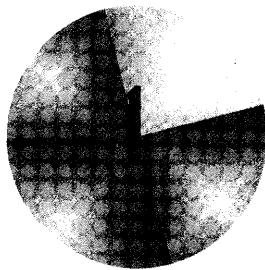
La ilustración de la portada ha sido diseñada por los autores y ejecutada por Lisa Marie Sardegna y Maryann Simmons utilizando SGI Inventor™ y Photshop™ de Adobe, y representa los siguientes elementos de la historia de la IA:

1. El algoritmo de planificación del *De Motu Animalium*, Aristóteles (400 a.C.).
2. El generador de conceptos del *Ars Magna* de Ramón Lull (1300 d.C.).
3. El motor de diferencias de Charles Babbage, un prototipo del primer computador universal (1848).
4. La notación de lógica de primer orden de Gottlob Frege (1789).
5. Los diagramas del razonamiento lógico de Lewis Carroll (1886).
6. La notación de redes probalísticas de Sewall Wright (1921).
7. Alan Turing (1912-1954).
8. El Robot Shakey (1969-1973).
9. Un sistema experto de diagnóstico actual (1993).

Sobre los autores

Stuart Russell nació en 1962 en Portsmouth, Inglaterra. En 1982, obtuvo su B.A. en Física formando parte de los primeros en el cuadro de honor de la Universidad de Oxford. En 1986 obtuvo el Ph. D. en Informática por la Universidad de Stanford. Más tarde empezó a trabajar en la Universidad de California, Berkeley, en donde es profesor de Informática, director del centro de Sistemas Inteligentes y colaborador del Smith-Zadeh en Ingeniería. En 1990, recibió el premio Presidential Young Investigator Award de la National Science Foundation (Fundación Nacional de las Ciencias), y en 1995 obtuvo el premio compartido en el Computers y Thought Award. En 1996 se convirtió en Millar Profesor de la University of California, y en el año 2000 fue nombrado rector de la universidad. En 1998, dio la lectura inaugural de la Stanford University, Forsythe Memorial Lectures. Forma parte como «Fellow», y es uno de los primeros miembros del Executive Council de la American Association for Artificial Intelligence. Ha publicado un gran número de trabajos, más de 100 sobre una gran variedad de temas en inteligencia artificial. Y es autor de otros dos libros: *The Use of Knowledge in Analogy and Induction*, y (con Erik Wefald) *Do the Right Thing: Studies in Limited Rationality*.

Peter Norvig es director de Search Quality de Google, Inc. y forma parte, como «Fellow» y miembro del Executive Council en la American Association for Artificial Intelligence. Anteriormente, fue director de la División de Ciencias Computacionales del Ames Research Center de la NASA, en donde supervisaba la investigación y el desarrollo en inteligencia artificial. Antes de eso, trabajó como director de ciencia en Junglee, donde ayudó a desarrollar uno de los primeros servicios de extracción de información en Internet, y trabajó también como científico senior en Sun Microsystems Laboratories, cuyo trabajo consistía en recuperar información inteligente. Recibió un B.S. en Matemáticas Aplicadas por la Brown University, y un Ph. D. en informática por la Universidad de California en Berkeley. Es profesor de la University of Southern California, y miembro de la facultad de investigación en Berkeley. Tiene en su haber más de 50 publicaciones en Informática entre las que se incluyen los libros: *Paradigms of AI Programming: Case Studies in Common Lisp*, *Verbmobil: A Translation System for Face-to-Face Dialog*, e *Intelligent Help Systems for UNIX*.



Introducción

Donde se intentará explicar por qué se considera a la inteligencia artificial un tema digno de estudio y donde se intentará definirla con exactitud; es esta tarea muy recomendable antes de emprender de lleno su estudio.

Los hombres se han denominado a sí mismos como *Homo sapiens* (hombre sabio) porque nuestras capacidades mentales son muy importantes para nosotros. Durante miles de años, hemos tratado de entender *cómo pensamos*; es decir, entender cómo un simple puñado de materia puede percibir, entender, predecir y manipular un mundo mucho más grande y complicado que ella misma. El campo de la **inteligencia artificial**, o IA, va más allá: no sólo intenta comprender, sino que también se esfuerza en construir entidades inteligentes.

La IA es una de las ciencias más recientes. El trabajo comenzó poco después de la Segunda Guerra Mundial, y el nombre se acuñó en 1956. La IA se cita, junto a la biología molecular, como un campo en el que a la mayoría de científicos de otras disciplinas «les gustaría trabajar». Un estudiante de ciencias físicas puede pensar razonablemente que todas las buenas ideas han sido ya propuestas por Galileo, Newton, Einstein y otros. Por el contrario, la IA aún tiene flecos sin cerrar en los que podrían trabajar varios Einstein a tiempo completo.

La IA abarca en la actualidad una gran variedad de subcampos, que van desde áreas de propósito general, como el aprendizaje y la percepción, a otras más específicas como el ajedrez, la demostración de teoremas matemáticos, la escritura de poesía y el diagnóstico de enfermedades. La IA sintetiza y automatiza tareas intelectuales y es, por lo tanto, potencialmente relevante para cualquier ámbito de la actividad intelectual humana. En este sentido, es un campo genuinamente universal.

1.1 ¿Qué es la IA?

RACIONALIDAD

Hemos proclamado que la IA es excitante, pero no hemos dicho qué *es*. La Figura 1.1 presenta definiciones de inteligencia artificial extraídas de ocho libros de texto. Las que aparecen en la parte superior se refieren a *procesos mentales* y al *razonamiento*, mientras que las de la parte inferior aluden a la *conducta*. Las definiciones de la izquierda miden el éxito en términos de la fidelidad en la forma de actuar de los *humanos*, mientras que las de la derecha toman como referencia un concepto ideal de inteligencia, que llamaremos **racionalidad**. Un sistema es racional si hace «lo correcto», en función de su conocimiento.

A lo largo de la historia se han seguido los cuatro enfoques mencionados. Como es de esperar, existe un enfrentamiento entre los enfoques centrados en los humanos y los centrados en torno a la *racionalidad*¹. El enfoque centrado en el comportamiento humano debe ser una ciencia empírica, que incluya hipótesis y confirmaciones mediante experimentos. El enfoque racional implica una combinación de matemáticas e ingeniería. Cada grupo al mismo tiempo ha ignorado y ha ayudado al otro. A continuación revisaremos cada uno de los cuatro enfoques con más detalle.

Sistemas que piensan como humanos	Sistemas que piensan racionalmente
«El nuevo y excitante esfuerzo de hacer que los computadores piensen... máquinas con mentes, en el más amplio sentido literal». (Haugeland, 1985)	«El estudio de las facultades mentales mediante el uso de modelos computacionales». (Charniak y McDermott, 1985)
«[La automatización de] actividades que vinculamos con procesos de pensamiento humano, actividades como la toma de decisiones, resolución de problemas, aprendizaje...» (Bellman, 1978)	«El estudio de los cálculos que hacen posible percibir, razonar y actuar». (Winston, 1992)
Sistemas que actúan como humanos	Sistemas que actúan racionalmente
«El arte de desarrollar máquinas con capacidad para realizar funciones que cuando son realizadas por personas requieren de inteligencia». (Kurzweil, 1990)	«La Inteligencia Computacional es el estudio del diseño de agentes inteligentes». (Poole <i>et al.</i> , 1998)
«El estudio de cómo lograr que los computadores realicen tareas que, por el momento, los humanos hacen mejor». (Rich y Knight, 1991)	«IA... está relacionada con conductas inteligentes en artefactos». (Nilsson, 1998)

Figura 1.1 Algunas definiciones de inteligencia artificial, organizadas en cuatro categorías.

¹ Conviene aclarar, que al distinguir entre comportamiento *humano* y *racional* no se está sugiriendo que los humanos son necesariamente «irracionales» en el sentido de «inestabilidad emocional» o «desequilibrio mental». Basta con darnos cuenta de que no somos perfectos: no todos somos maestros de ajedrez, incluso aquellos que conocemos todas las reglas del ajedrez; y desafortunadamente, no todos obtenemos un sobresaliente en un examen. Kahneman *et al.* (1982) ha elaborado un catálogo con algunos de los errores que sistemáticamente cometan los humanos cuando razonan.

Comportamiento humano: el enfoque de la Prueba de Turing

PRUEBA DE TURING

La **Prueba de Turing**, propuesta por Alan Turing (1950), se diseñó para proporcionar una definición operacional y satisfactoria de inteligencia. En vez de proporcionar una lista larga y quizás controvertida de cualidades necesarias para obtener inteligencia artificialmente, él sugirió una prueba basada en la incapacidad de diferenciar entre entidades inteligentes indiscutibles y seres humanos. El computador supera la prueba si un evaluador humano no es capaz de distinguir si las respuestas, a una serie de preguntas planteadas, son de una persona o no. En el Capítulo 26 se comentan detalles de esta prueba y se discute si un computador que supera la prueba es realmente inteligente. Hoy por hoy, podemos decir que programar un computador para que supere la prueba requiere un trabajo considerable. El computador debería poseer las siguientes capacidades:

PROCESAMIENTO DE LENGUAJE NATURAL

REPRESENTACIÓN DEL CONOCIMIENTO

RAZONAMIENTO AUTOMÁTICO

APRENDIZAJE MÁQUINA

PRUEBA DE TURING GLOBAL

VISTA COMPUTACIONAL

ROBÓTICA

- **Procesamiento de lenguaje natural** que le permita comunicarse satisfactoriamente en inglés.
- **Representación del conocimiento** para almacenar lo que se conoce o siente.
- **Razonamiento automático** para utilizar la información almacenada para responder a preguntas y extraer nuevas conclusiones.
- **Aprendizaje automático** para adaptarse a nuevas circunstancias y para detectar y extrapolar patrones.

La Prueba de Turing evitó deliberadamente la interacción *física* directa entre el evaluador y el computador, dado que para medir la inteligencia es innecesario simular físicamente a una persona. Sin embargo, la llamada Prueba Global de Turing incluye una señal de vídeo que permite al evaluador valorar la capacidad de percepción del evaluado, y también le da la oportunidad al evaluador de pasar objetos físicos «a través de una ventanita». Para superar la Prueba Global de Turing el computador debe estar dotado de

- **Visión computacional** para percibir objetos.
- **Robótica** para manipular y mover objetos.

Estas seis disciplinas abarcan la mayor parte de la IA, y Turing merece ser reconocido por diseñar una prueba que se conserva vigente después de 50 años. Los investigadores del campo de la IA han dedicado poco esfuerzo a la evaluación de sus sistemas con la Prueba de Turing, por creer que es más importante el estudio de los principios en los que se basa la inteligencia que duplicar un ejemplar. La búsqueda de un ingenio que «volara artificialmente» tuvo éxito cuando los hermanos Wright, entre otros, dejaron de imitar a los pájaros y comprendieron los principios de la aerodinámica. Los textos de ingeniería aerodinámica no definen el objetivo de su campo como la construcción de «máquinas que vuelen como palomas de forma que puedan incluso confundir a otras palomas».

Pensar como un humano: el enfoque del modelo cognitivo

Para poder decir que un programa dado piensa como un humano, es necesario contar con un mecanismo para determinar cómo piensan los humanos. Es necesario *penetrar* en el

funcionamiento de las mentes humanas. Hay dos formas de hacerlo: mediante introspección (intentando atrapar nuestros propios pensamientos conforme éstos van apareciendo) y mediante experimentos psicológicos. Una vez se cuente con una teoría lo suficientemente precisa sobre cómo trabaja la mente, se podrá expresar esa teoría en la forma de un programa de computador. Si los datos de entrada/salida del programa y los tiempos de reacción son similares a los de un humano, existe la evidencia de que algunos de los mecanismos del programa se pueden comparar con los que utilizan los seres humanos. Por ejemplo, a Allen Newell y Herbert Simon, que desarrollaron el «Sistema de Resolución General de Problemas» (SRGP) (Newell y Simon, 1961), no les bastó con que su programa resolviera correctamente los problemas propuestos. Lo que les interesaba era seguir la pista de las etapas del proceso de razonamiento y compararlas con las seguidas por humanos a los que se les enfrentó a los mismos problemas. En el campo interdisciplinario de la **ciencia cognitiva** convergen modelos computacionales de IA y técnicas experimentales de psicología intentando elaborar teorías precisas y verificables sobre el funcionamiento de la mente humana.

La ciencia cognitiva es un campo fascinante, merecedora de una enciclopedia dedicada a ella (Wilson y Keil, 1999). En este libro no se intenta describir qué se conoce de la cognición humana. Ocasionalmente se hacen comentarios acerca de similitudes o diferencias entre técnicas de IA y cognición humana. La auténtica ciencia cognitiva se fundamenta necesariamente en la investigación experimental en humanos y animales, y en esta obra se asume que el lector sólo tiene acceso a un computador para experimentar.

En los comienzos de la IA había confusión entre las distintas aproximaciones: un autor podría argumentar que un algoritmo resolvía adecuadamente una tarea y que *por tanto* era un buen modelo de representación humana, o viceversa. Los autores actuales hacen diferencia entre las dos reivindicaciones; esta distinción ha permitido que ambas disciplinas, IA y ciencia cognitiva, se desarrollen más rápidamente. Los dos campos continúan alimentándose entre sí, especialmente en las áreas de la visión y el lenguaje natural. En particular, el campo de la visión ha avanzado recientemente con la ayuda de una propuesta integrada que tiene en cuenta la evidencia neurofisiológica y los modelos computacionales.

Pensamiento racional: el enfoque de las «leyes del pensamiento»

El filósofo griego Aristóteles fue uno de los primeros en intentar codificar la «manera correcta de pensar», es decir, un proceso de razonamiento irrefutable. Sus **silogismos** son esquemas de estructuras de argumentación mediante las que siempre se llega a conclusiones correctas si se parte de premisas correctas (por ejemplo: «Sócrates es un hombre; todos los hombres son mortales; por lo tanto Sócrates es mortal»). Estas leyes de pensamiento supuestamente gobiernan la manera de operar de la mente; su estudio fue el inicio de un campo llamado **Lógica**.

Estudiosos de la lógica desarrollaron, en el siglo XIX, una notación precisa para definir sentencias sobre todo tipo de elementos del mundo y especificar relaciones entre

LOGISTA

ellos (compárese esto con la notación aritmética común, que prácticamente sólo sirve para representar afirmaciones acerca de la igualdad y desigualdad entre números). Ya en 1965 existían programas que, en principio, resolvían *cualquier* problema resoluble descrito en notación lógica². La llamada tradición **logista** dentro del campo de la inteligencia artificial trata de construir sistemas inteligentes a partir de estos programas.

Este enfoque presenta dos obstáculos. No es fácil transformar conocimiento informal y expresarlo en los términos formales que requieren de notación lógica, particularmente cuando el conocimiento que se tiene es inferior al 100 por 100. En segundo lugar, hay una gran diferencia entre poder resolver un problema «en principio» y hacerlo en la práctica. Incluso problemas con apenas una docena de datos pueden agotar los recursos computacionales de cualquier computador a menos que cuente con alguna directiva sobre los pasos de razonamiento que hay que llevar a cabo primero. Aunque los dos obstáculos anteriores están presentes en *todo* intento de construir sistemas de razonamiento computacional, surgieron por primera vez en la tradición lógica.

Actuar de forma racional: el enfoque del agente racional

AGENTE

Un **agente** es algo que razona (*agente* viene del latín *agere*, hacer). Pero de los agentes informáticos se espera que tengan otros atributos que los distingan de los «programas» convencionales, como que estén dotados de controles autónomos, que perciban su entorno, que persistan durante un período de tiempo prolongado, que se adapten a los cambios, y que sean capaces de alcanzar objetivos diferentes. Un **agente racional** es aquel que actúa con la intención de alcanzar el mejor resultado o, cuando hay incertidumbre, el mejor resultado esperado.

AGENTE RACIONAL

En el caso del enfoque de la IA según las «leyes del pensamiento», todo el énfasis se pone en hacer inferencias correctas. La obtención de estas inferencias correctas puede, a veces, formar *parte* de lo que se considera un agente racional, ya que una manera racional de actuar es llegar a la conclusión lógica de que si una acción dada permite alcanzar un objetivo, hay que llevar a cabo dicha acción. Sin embargo, el efectuar una inferencia correcta no depende siempre de la *racionalidad*, ya que existen situaciones para las que no hay nada correcto que hacer y en las que hay que tomar una decisión. Existen también formas de actuar racionalmente que no implican realizar inferencias. Por ejemplo, el retirar la mano de una estufa caliente es un acto reflejo mucho más eficiente que una respuesta lenta llevada a cabo tras una deliberación cuidadosa.

Todas las habilidades que se necesitan en la Prueba de Turing deben permitir emprender acciones racionales. Por lo tanto, es necesario contar con la capacidad para representar el conocimiento y razonar basándonos en él, porque ello permitirá alcanzar decisiones correctas en una amplia gama de situaciones. Es necesario ser capaz de generar sentencias comprensibles en lenguaje natural, ya que el enunciado de tales oraciones permite a los agentes desenvolverse en una sociedad compleja. El aprendizaje no se lleva a cabo por erudición exclusivamente, sino que profundizar en el conocimiento de cómo funciona el mundo facilita la concepción de estrategias mejores para manejarse en él.

² Si no se encuentra una solución, el programa nunca debe parar de buscarla.

La percepción visual es necesaria no sólo porque ver es divertido, sino porque es necesaria para poder tener una idea mejor de lo que una acción puede llegar a representar, por ejemplo, el ver un delicioso bocadillo contribuirá a que nos acerquemos a él.

Por esta razón, el estudiar la IA desde el enfoque del diseño de un agente racional ofrece al menos dos ventajas. La primera es más general que el enfoque que proporcionan las «leyes del pensamiento», dado que el efectuar inferencias correctas es sólo uno de los mecanismos existentes para garantizar la racionalidad. La segunda es más afín a la forma en la que se ha producido el avance científico que los enfoques basados en la conducta o pensamiento humano, porque la norma de la racionalidad está claramente definida y es de aplicación general. Por el contrario, la conducta humana se adapta bien a un entorno específico, y en parte, es producto de un proceso evolutivo complejo, en gran medida desconocido, que aún está lejos de llevarnos a la perfección. *Por tanto, esta obra se centrará en los principios generales que rigen a los agentes racionales y en los elementos necesarios para construirlos.* Más adelante quedará patente que a pesar de la aparente facilidad con la que se puede describir un problema, cuando se intenta resolver surgen una enorme variedad de cuestiones. El Capítulo 2 revisa algunos de estos aspectos con más detalle.

Un elemento importante a tener en cuenta es el siguiente: más bien pronto que tarde se observará cómo obtener una racionalidad perfecta (hacer siempre lo correcto) no es posible en entornos complejos. La demanda computacional que esto implica es demasiado grande. En la mayor parte de esta obra se adoptará la hipótesis de trabajo de que la racionalidad perfecta es un buen punto de partida para el análisis. Lo cual simplifica el problema y proporciona el escenario base adecuado sobre el que se asientan los cimientos de este campo. Los Capítulos 6 y 17 se centran explícitamente en el tema de la **racionalidad limitada** (actuar adecuadamente cuando no se cuenta con el tiempo suficiente para efectuar todos los cálculos que serían deseables).

RACIONALIDAD
LIMITADA

1.2 Los fundamentos de la inteligencia artificial

Esta sección presenta una breve historia de las disciplinas que han contribuido con ideas, puntos de vista y técnicas al desarrollo de la IA. Como toda revisión histórica, en este caso se centra en un pequeño número de personas, eventos e ideas e ignora otras que también fueron importantes. La historia se organiza en torno a una serie de cuestiones, dejando claro que no se quiere dar la impresión de que estas cuestiones son las únicas por las que las disciplinas se han preocupado y que el objetivo último de todas estas disciplinas era hacer avanzar la IA.

Filosofía (desde el año 428 a.C. hasta el presente)

- ¿Se pueden utilizar reglas formales para extraer conclusiones válidas?
- ¿Cómo se genera la inteligencia mental a partir de un cerebro físico?
- ¿De dónde viene el conocimiento?
- ¿Cómo se pasa del conocimiento a la acción?

Aristóteles (384-322 a.C.) fue el primero en formular un conjunto preciso de leyes que gobernaban la parte racional de la inteligencia. Él desarrolló un sistema informal para razonar  lógicamente con silogismos, que en principio permitía extraer conclusiones mecánicamente, a partir de premisas iniciales. Mucho después, Ramón Lull (d. 1315) tuvo la idea de que el razonamiento útil se podría obtener por medios artificiales. Sus «ideas» aparecen representadas en la portada de este manuscrito. Thomas Hobbes (1588-1679) propuso que el razonamiento era como la computación numérica, de forma que «nosotros sumamos y restamos silenciosamente en nuestros pensamientos». La automatización de la computación en sí misma estaba en marcha; alrededor de 1500, Leonardo da Vinci (1452-1519) diseñó, aunque no construyó, una calculadora mecánica; construcciones recientes han mostrado que su diseño era funcional. La primera máquina calculadora conocida se construyó alrededor de 1623 por el científico alemán Wilhelm Schickard (1592-1635), aunque la Pascalina, construida en 1642 por Blaise Pascal (1623-1662), sea más famosa. Pascal escribió que «la máquina aritmética produce efectos que parecen más similares a los pensamientos que a las acciones animales». Gottfried Wilhelm Leibniz (1646-1716) construyó un dispositivo mecánico con el objetivo de llevar a cabo operaciones sobre conceptos en lugar de sobre números, pero su campo de acción era muy limitado.

Ahora que sabemos que un conjunto de reglas pueden describir la parte racional y formal de la mente, el siguiente paso es considerar la mente como un sistema físico. René Descartes (1596-1650) proporciona la primera discusión clara sobre la distinción entre la mente y la materia y los problemas que surgen. Uno de los problemas de una concepción puramente física de la mente es que parece dejar poco margen de maniobra al libre albedrío: si el pensamiento está totalmente gobernado por leyes físicas, entonces una piedra podría «decidir» caer en dirección al centro de la Tierra gracias a su libre albedrío. A pesar de ser denodado defensor de la capacidad de razonamiento, Descartes fue un defensor del **dualismo**. Sostenía que existe una parte de la mente (o del alma o del espíritu) que está al margen de la naturaleza, exenta de la influencia de las leyes físicas. Los animales, por el contrario, no poseen esta cualidad dual; a ellos se le podría concebir como si se tratases de máquinas. Una alternativa al dualismo es el **materialismo**, que considera que las operaciones del cerebro realizadas de acuerdo a las leyes de la física *constituyen* la mente. El libre albedrío es simplemente la forma en la que la percepción de las opciones disponibles aparecen en el proceso de selección.

DUALISMO

MATERIALISMO

EMPÍRICO

INDUCCIÓN

Dada una mente física que gestiona conocimiento, el siguiente problema es establecer las fuentes de este conocimiento. El movimiento **empírico**, iniciado con el *Novum Organum*³, de Francis Bacon (1561-1626), se caracteriza por el aforismo de John Locke (1632-1704): «Nada existe en la mente que no haya pasado antes por los sentidos». David Hume (1711-1776) propuso en *A Treatise of Human Nature* (Hume, 1739) lo que actualmente se conoce como principio de **inducción**: las reglas generales se obtienen mediante la exposición a asociaciones repetidas entre sus elementos. Sobre la base de las propuestas de Ludwig Wittgenstein (1889-1951) y Bertrand Russell (1872-1970), el famoso Círculo de Viena, liderado por Rudolf Carnap (1891-1970), desarrolló la doctrina del **positivismo lógico**. Esa doctrina sostiene que todo el conocimiento se puede

³ Una actualización del *Organon*, o instrumento de pensamiento, de Aristóteles.

POSITIVISMO LÓGICO

SENTENCIA DE
OBSERVACIÓNTEORÍA DE LA
CONFIRMACIÓN

caracterizar mediante teorías lógicas relacionadas, en última instancia, con **sentencias de observación** que corresponden a estímulos sensoriales⁴. La **teoría de la confirmación** de Carnap y Carl Hempel (1905-1997) intenta explicar cómo el conocimiento se obtiene a partir de la experiencia. El libro de Carnap, *The Logical Structure of the World* (1928), define un procedimiento computacional explícito para la extracción de conocimiento a partir de experiencias primarias. Fue posiblemente la primera teoría en mostrar la mente como un proceso computacional.

El último elemento en esta discusión filosófica sobre la mente es la relación que existe entre conocimiento y acción. Este asunto es vital para la IA, ya que la inteligencia requiere tanto acción como razonamiento. Más aún, simplemente con comprender cómo se justifican determinadas acciones se puede llegar a saber cómo construir un agente cuyas acciones sean justificables (o racionales). Aristóteles argumenta que las acciones se pueden justificar por la conexión lógica entre los objetivos y el conocimiento de los efectos de las acciones (la última parte de este extracto también aparece en la portada de este libro):

¿Cómo es que el pensamiento viene acompañado en algunos casos de acciones y en otros no?, ¿en algunos casos por movimiento y en otros no? Parece como si la misma cosa sucediera tanto si razonáramos o hiciéramos inferencias sobre objetos que no cambian; pero en este caso el fin es una proposición especulativa... mientras la conclusión resultante de las dos premisas es una acción... Yo necesito abrigarme; una manta abriga. Yo necesito una manta. Qué necesito, qué debo hacer; necesito una manta. Necesito hacer una manta. Y la conclusión, «Yo tengo que hacer una manta», es una acción. (Nussbaum, 1978, p. 40)

En *Nicomachean Ethics* (Libro III. 3, 1112b), Aristóteles continúa trabajando en este tema, sugiriendo un algoritmo:

Nosotros no reflexionamos sobre los fines, sino sobre los medios. Un médico no reflexiona sobre si debe curar, ni un orador sobre si debe persuadir... Ellos asumen el fin y consideran cómo y con qué medios se obtienen, y si resulta fácil y es por tanto productivo; mientras que si sólo se puede alcanzar por un medio se tiene en consideración *cómo* se alcanzará por este y por qué medios se obtendrá *este*, hasta que se llegue a la causa primera..., y lo último en el orden del análisis parece ser lo primero en el orden de los acontecimientos. Y si se llega a un estado imposible, se abandona la búsqueda, como por ejemplo si se necesita dinero y no se puede conseguir; pero si hay una posibilidad se intentará.

El algoritmo de Aristóteles se implementó 2.300 años más tarde por Newell y Simon con la ayuda de su programa SRGP. El cual se conoce como sistema de planificación regresivo (véase el Capítulo 11).

El análisis basado en objetivos es útil, pero no indica qué hacer cuando varias acciones nos llevan a la consecución del objetivo, o cuando ninguna acción facilita su completa consecución. Antoine Arnauld (1612-1694) describió correctamente una forma cuantitativa para decidir qué acción llevar a cabo en un caso como este (véase el Capítulo 16). El libro *Utilitarianism* (Mill, 1863) de John Stuart Mill (1806-1873) propone

⁴ En este contexto, es posible comprobar o rechazar toda aseveración significativa mediante el análisis del significado de las palabras o mediante la experimentación. Dado que esto no es aplicable en la mayor parte del ámbito de la metafísica, como era intención, el positivismo lógico se hizo impopular en algunos círculos.

la idea de un criterio de decisión racional en todos los ámbitos de la actividad humana. En la siguiente sección se explica una teoría de la decisión más formalmente.

Matemáticas (aproximadamente desde el año 800 al presente)

- ¿Qué reglas formales son las adecuadas para obtener conclusiones válidas?
- ¿Qué se puede computar?
- ¿Cómo razonamos con información incierta?

Los filósofos delimitaron las ideas más importantes de la IA, pero para pasar de ahí a una ciencia formal es necesario contar con una formulación matemática en tres áreas fundamentales: lógica, computación y probabilidad.

El concepto de lógica formal se remonta a los filósofos de la antigua Grecia (véase el Capítulo 7), pero su desarrollo matemático comenzó realmente con el trabajo de George Boole (1815-1864) que definió la lógica proposicional o Booleana (Boole, 1847). En 1879, Gottlob Frege (1848-1925) extendió la lógica de Boole para incluir objetos y relaciones, y creó la lógica de primer orden que se utiliza hoy como el sistema más básico de representación de conocimiento⁵. Alfred Tarski (1902-1983) introdujo una teoría de referencia que enseña cómo relacionar objetos de una lógica con objetos del mundo real. El paso siguiente consistió en definir los límites de lo que se podía hacer con la lógica y la informática.

ALGORITMO

Se piensa que el primer **algoritmo** no trivial es el algoritmo Euclídeo para el cálculo del máximo común divisor. El considerar los algoritmos como objetos en sí mismos se remonta a la época de al-Khowarazmi, un matemático persa del siglo IX, con cuyos escritos también se introdujeron los números arábigos y el álgebra en Europa. Boole, entre otros, presentó algoritmos para llevar a cabo deducciones lógicas y hacia el final del siglo XIX se llevaron a cabo numerosos esfuerzos para formalizar el razonamiento matemático general con la lógica deductiva. En 1900, David Hilbert (1862-1943) presentó una lista de 23 problemas que acertadamente predijo ocuparían a los matemáticos durante todo ese siglo. En el último de ellos se preguntaba si existe un algoritmo que permita determinar la validez de cualquier proposición lógica en la que aparezcan números naturales (el famoso *Entscheidungsproblem*, o problema de decisión). Básicamente, lo que Hilbert se preguntaba es si hay límites fundamentales en la capacidad de los procedimientos efectivos de demostración. En 1930, Kurt Gödel (1906-1978) demostró que existe un procedimiento eficiente para demostrar cualquier aseveración verdadera en la lógica de primer orden de Frege y Russell, sin embargo con la lógica de primer orden no era posible capturar el principio de inducción matemática necesario para la caracterización de los números naturales. En 1931, demostró que, en efecto, existen límites reales. Mediante su **teorema de incompletitud** demostró que en cualquier lenguaje que tuviera la capacidad suficiente para expresar las propiedades de los números naturales, existen aseveraciones verdaderas no decidible en el sentido de que no es posible decidir su validez mediante ningún algoritmo.

TEOREMA DE INCOMPLETITUD

⁵ La notación para la lógica de primer orden propuesta por Frege no se ha aceptado universalmente, por razones que son aparentemente obvias cuando se observa el ejemplo que aparece en la cubierta de este libro.

El resultado fundamental anterior se puede interpretar también como la indicación de que existen algunas funciones de los números enteros que no se pueden representar mediante un algoritmo, es decir no se pueden calcular. Lo anterior llevó a Alan Turing (1912-1954) a tratar de caracterizar exactamente aquellas funciones que sí *eran* susceptibles de ser caracterizadas. La noción anterior es de hecho problemática hasta cierto punto, porque no es posible dar una definición formal a la noción de cálculo o procedimiento efectivo. No obstante, la tesis de Church-Turing, que afirma que la máquina de Turing (Turing, 1936) es capaz de calcular cualquier función computable, goza de aceptación generalizada ya que proporciona una definición suficiente. Turing también demostró que existen algunas funciones que no se pueden calcular mediante la máquina de Turing. Por ejemplo, ninguna máquina puede decidir *en general* si un programa dado producirá una respuesta a partir de unas entradas, o si seguirá calculando indefinidamente.

Si bien ser no decidable ni computable son importantes para comprender el proceso del cálculo, la noción de **intratabilidad** tuvo repercusiones más importantes. En términos generales se dice que un problema es intratable si el tiempo necesario para la resolución de casos particulares de dicho problema crece exponencialmente con el tamaño de dichos casos. La diferencia entre crecimiento polinomial y exponencial de la complejidad se destacó por primera vez a mediados de los años 60 (Cobham, 1964; Edmonds, 1965). Es importante porque un crecimiento exponencial implica la imposibilidad de resolver casos moderadamente grandes en un tiempo razonable. Por tanto, se debe optar por dividir el problema de la generación de una conducta inteligente en subproblemas que sean tratables en vez de manejar problemas intratables.

¿Cómo se puede reconocer un problema intratable? La teoría de la **NP-completitud**, propuesta por primera vez por Steven Cook (1971) y Richard Karp (1972) propone un método. Cook y Karp demostraron la existencia de grandes clases de problemas de razonamiento y búsqueda combinatoria canónica que son NP completos. Toda clase de problema a la que la clase de problemas NP completos se pueda reducir será seguramente intratable (aunque no se ha demostrado que los problemas NP completos son necesariamente intratables, la mayor parte de los teóricos así lo creen). Estos resultados contrastan con el optimismo con el que la prensa popular recibió a los primeros computadores, «Supercerebros Electrónicos» que eran «¡Más rápidos que Einstein!». A pesar del rápido incremento en la velocidad de los computadores, los sistemas inteligentes se caracterizarán por el uso cuidadoso que hacen de los recursos. De manera sucinta, ¡el mundo es un ejemplo de problema *extremadamente* grande! Recientemente la IA ha ayudado a explicar por qué algunos ejemplos de problemas NP completos son difíciles de resolver y otros son fáciles (Cheeseman *et al.*, 1991).

Además de la lógica y el cálculo, la tercera gran contribución de las matemáticas a la IA es la teoría de la **probabilidad**. El italiano Gerolamo Cardano (1501-1576) fue el primero en proponer la idea de probabilidad, presentándola en términos de los resultados de juegos de apuesta. La probabilidad se convirtió pronto en parte imprescindible de las ciencias cuantitativas, ayudando en el tratamiento de mediciones con incertidumbre y de teorías incompletas. Pierre Fermat (1601-1665), Blaise Pascal (1623-1662), James Bernoulli (1654-1705), Pierre Laplace (1749-1827), entre otros, hicieron avanzar esta teoría e introdujeron nuevos métodos estadísticos. Thomas Bayes (1702-1761) propuso una

INTRATABILIDAD

NP-COMPLETITUD

PROBABILIDAD

regla para la actualización de probabilidades subjetivas a la luz de nuevas evidencias. La regla de Bayes y el área resultante llamado análisis Bayesiano conforman la base de las propuestas más modernas que abordan el razonamiento incierto en sistemas de IA.

Economía (desde el año 1776 hasta el presente)

- ¿Cómo se debe llevar a cabo el proceso de toma de decisiones para maximizar el rendimiento?
- ¿Cómo se deben llevar a cabo acciones cuando otros no colaboren?
- ¿Cómo se deben llevar a cabo acciones cuando los resultados se obtienen en un futuro lejano?

La ciencia de la economía comenzó en 1776, cuando el filósofo escocés Adam Smith (1723-1790) publicó *An Inquiry into the Nature and Causes of the Wealth of Nations*. Aunque los antiguos griegos, entre otros, habían hecho contribuciones al pensamiento económico, Smith fue el primero en tratarlo como una ciencia, utilizando la idea de que las economías pueden concebirse como un conjunto de agentes individuales que intentan maximizar su propio estado de bienestar económico. La mayor parte de la gente cree que la economía sólo se trata de dinero, pero los economistas dicen que ellos realmente estudian cómo la gente toma decisiones que les llevan a obtener los beneficios esperados. Léon Walras (1834-1910) formalizó el tratamiento matemático del «beneficio deseado» o **utilidad**, y fue posteriormente mejorado por Frank Ramsey (1931) y después por John von Neumann y Oskar Morgenstern en su libro *The Theory of Games and Economic Behavior* (1944).

La **teoría de la decisión**, que combina la teoría de la probabilidad con la teoría de la utilidad, proporciona un marco completo y formal para la toma de decisiones (económicas o de otra índole) realizadas bajo incertidumbre, esto es, en casos en los que las descripciones probabilísticas capturan adecuadamente la forma en la que se toman las decisiones en el entorno; lo cual es adecuado para «grandes» economías en las que cada agente no necesita prestar atención a las acciones que lleven a cabo el resto de los agentes individualmente. Cuando se trata de «pequeñas» economías, la situación se asemeja más a la de un **juego**: las acciones de un jugador pueden afectar significativamente a la utilidad de otro (tanto positiva como negativamente). Los desarrollos de von Neumann y Morgenstern a partir de la **teoría de juegos** (véase también Luce y Raiffa, 1957) mostraban el hecho sorprendente de que, en algunos juegos, un agente racional debía actuar de forma aleatoria o, al menos, aleatoria en apariencia con respecto a sus contrincantes.

La gran mayoría de los economistas no se preocuparon de la tercera cuestión mencionada anteriormente, es decir, cómo tomar decisiones racionales cuando los resultados de las acciones no son inmediatos y por el contrario se obtienen los resultados de las acciones de forma *secuencial*. El campo de la **investigación operativa** persigue este objetivo; dicho campo emergió en la Segunda Guerra Mundial con los esfuerzos llevados a cabo en el Reino Unido en la optimización de instalaciones de radar, y posteriormente en aplicaciones civiles relacionadas con la toma de decisiones de dirección complejas. El trabajo de Richard Bellman (1957) formaliza una clase de problemas de decisión secuencial llamados **procesos de decisión de Markov**, que se estudiarán en los Capítulos 17 y 21.

SATISFACCIÓN

El trabajo en la economía y la investigación operativa ha contribuido en gran medida a la noción de agente racional que aquí se presenta, aunque durante muchos años la investigación en el campo de la IA se ha desarrollado por sendas separadas. Una razón fue la **complejidad** aparente que trae consigo el tomar decisiones racionales. Herbert Simon (1916-2001), uno de los primeros en investigar en el campo de la IA, ganó el premio Nobel en Economía en 1978 por su temprano trabajo, en el que mostró que los modelos basados en **satisfacción** (que toman decisiones que son «suficientemente buenas», en vez de realizar cálculos laboriosos para alcanzar decisiones óptimas) proporcionaban una descripción mejor del comportamiento humano real (Simon, 1947). En los años 90, hubo un resurgimiento del interés en las técnicas de decisión teórica para sistemas basados en agentes (Wellman, 1995).

NEUROCIENCIA

Neurociencia (desde el año 1861 hasta el presente)

- ¿Cómo procesa información el cerebro?

La **Neurociencia** es el estudio del sistema neurológico, y en especial del cerebro. La forma exacta en la que en un cerebro se genera el pensamiento es uno de los grandes misterios de la ciencia. Se ha observado durante miles de años que el cerebro está de alguna manera involucrado en los procesos de pensamiento, ya que fuertes golpes en la cabeza pueden ocasionar minusvalía mental. También es ampliamente conocido que los cerebros humanos son de alguna manera diferentes; aproximadamente en el 335 a.C. Aristóteles escribió, «de entre todos los animales el hombre tiene el cerebro más grande en proporción a su tamaño»⁶. Aunque, no fue hasta mediados del siglo XVIII cuando se aceptó mayoritariamente que el cerebro es la base de la conciencia. Hasta este momento, se pensaba que estaba localizado en el corazón, el bazo y la glándula pineal.

El estudio de Paul Broca (1824-1880) sobre la afasia (dificultad para hablar) en pacientes con el cerebro dañado, en 1861, le dio fuerza a este campo y convenció a la sociedad médica de la existencia de áreas localizadas en el cerebro responsables de funciones cognitivas específicas. En particular, mostró que la producción del habla se localizaba en una parte del hemisferio izquierdo; hoy en día conocida como el área de Broca⁷. En esta época ya se sabía que el cerebro estaba formado por células nerviosas o **neuronas**, pero no fue hasta 1873 cuando Camillo Golgi (1843-1926) desarrolló una técnica de coloración que permitió la observación de neuronas individuales en el cerebro (véase la Figura 1.2). Santiago Ramón y Cajal (1852-1934) utilizó esta técnica en sus estudios pioneros sobre la estructura neuronal del cerebro⁸.

En la actualidad se dispone de información sobre la relación existente entre las áreas del cerebro y las partes del cuerpo humano que controlan o de las que reciben impulsos

⁶ Desde entonces, se ha descubierto que algunas especies de delfines y ballenas tienen cerebros relativamente grandes. Ahora se piensa que el gran tamaño de los cerebros humanos se debe en parte a la mejora reciente en su sistema de refrigeración.

⁷ Muchos citan a Alexander Hood (1824) como una fuente posiblemente anterior.

⁸ Golgi insistió en la creencia de que las funciones cerebrales se desarrollaron inicialmente en el medio continuo en el que las neuronas estaban inmersas, mientras que Cajal propuso la «doctrina neuronal». Ambos compartieron el premio Nobel en 1906 pronunciando un discurso de aceptación antagónico.

NEURONAS

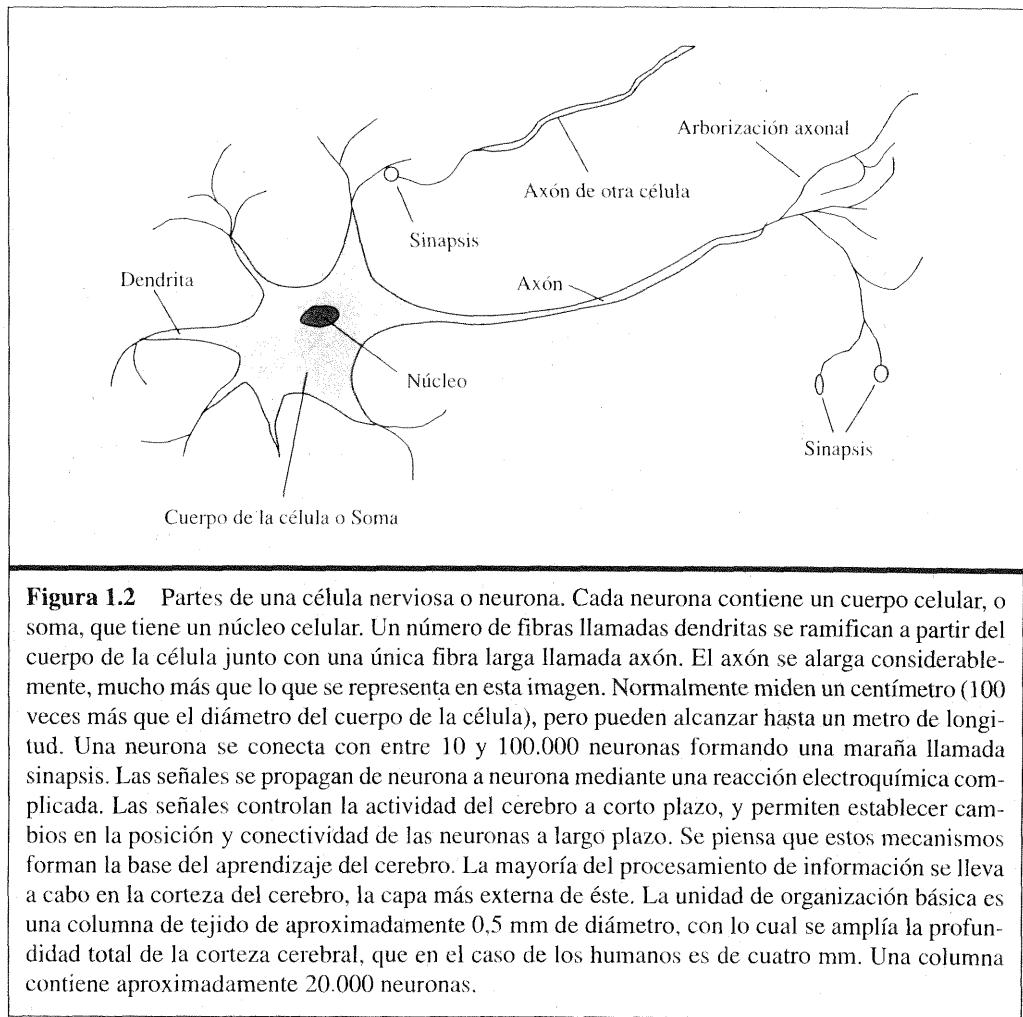


Figura 1.2 Partes de una célula nerviosa o neurona. Cada neurona contiene un cuerpo celular, o soma, que tiene un núcleo celular. Un número de fibras llamadas dendritas se ramifican a partir del cuerpo de la célula junto con una única fibra larga llamada axón. El axón se alarga considerablemente, mucho más que lo que se representa en esta imagen. Normalmente miden un centímetro (100 veces más que el diámetro del cuerpo de la célula), pero pueden alcanzar hasta un metro de longitud. Una neurona se conecta con entre 10 y 100.000 neuronas formando una maraña llamada sinapsis. Las señales se propagan de neurona a neurona mediante una reacción electroquímica complicada. Las señales controlan la actividad del cerebro a corto plazo, y permiten establecer cambios en la posición y conectividad de las neuronas a largo plazo. Se piensa que estos mecanismos forman la base del aprendizaje del cerebro. La mayoría del procesamiento de información se lleva a cabo en la corteza del cerebro, la capa más externa de éste. La unidad de organización básica es una columna de tejido de aproximadamente 0,5 mm de diámetro, con lo cual se amplía la profundidad total de la corteza cerebral, que en el caso de los humanos es de cuatro mm. Una columna contiene aproximadamente 20.000 neuronas.

sensoriales. Tales relaciones pueden cambiar de forma radical incluso en pocas semanas, y algunos animales parecen disponer de múltiples posibilidades. Más aún, no se tiene totalmente claro cómo algunas áreas se pueden encargar de ciertas funciones que eran responsabilidad de áreas dañadas. No hay prácticamente ninguna teoría que explique cómo se almacenan recuerdos individuales.

Los estudios sobre la actividad de los cerebros intactos comenzó en 1929 con el descubrimiento del electroencefalograma (EEG) desarrollado por Hans Berger. El reciente descubrimiento de las imágenes de resonancia magnética funcional (IRMf) (Ogawa *et al.*, 1990) está proporcionando a los neurólogos imágenes detalladas de la actividad cerebral sin precedentes, permitiéndoles obtener medidas que se corresponden con procesos cognitivos en desarrollo de manera muy interesante. Este campo está evolucionando gracias a los avances en los estudios en celdas individuales y su actividad neuronal. A pesar de estos avances, nos queda un largo camino para llegar a comprender cómo funcionan todos estos procesos cognitivos.



La conclusión verdaderamente increíble es que *una colección de simples células pueden llegar a generar razonamiento, acción, y conciencia* o, dicho en otras palabras, *los cerebros generan las inteligencias* (Searle, 1992). La única teoría alternativa es el misticismo: que nos dice que existe alguna esfera mística en la que las mentes operan fuera del control de la ciencia física.

Cerebros y computadores digitales realizan tareas bastante diferentes y tienen propiedades distintas. La Figura 1.3 muestra cómo hay 1.000 veces más neuronas en un cerebro humano medio que puertas lógicas en la UCP de un computador estándar. La ley de Moore⁹ predice que el número de puertas lógicas de la UCP se igualará con el de neuronas del cerebro alrededor del año 2020. Por supuesto, poco se puede inferir de esta predicción; más aún, la diferencia en la capacidad de almacenamiento es insignificante comparada con las diferencias en la velocidad de intercambio y en paralelismo. Los circuitos de los computadores pueden ejecutar una instrucción en un nanosegundo, mientras que las neuronas son millones de veces más lentas. Las neuronas y las sinapsis del cerebro están activas simultáneamente, mientras que los computadores actuales tienen una o como mucho varias UCP. Por tanto, incluso sabiendo que un computador es un millón de veces más rápido en cuanto a su velocidad de intercambio, el cerebro acaba siendo 100.000 veces más rápido en lo que hace.



Psicología (desde el año 1879 hasta el presente)

- ¿Cómo piensan y actúan los humanos y los animales?

La psicología científica se inició con los trabajos del físico alemán Hermann von Helmholtz (1821-1894), según se referencia habitualmente, y su discípulo Wilhelm Wundt (1832-1920). Helmholtz aplicó el método científico al estudio de la vista humana, y su obra *Handbook of Physiological Optics*, todavía en nuestros días, se considera como «el tratado actual más importante sobre la física y la fisiología de la vista humana» (Nalwa, 1993, p. 15). En 1879, Wundt abrió el primer laboratorio de psicología experimental en

	Computador	Cerebro Humano
Unidades computacionales	1 UCP, 10^8 puertas	10^{11} neuronas
Unidades de Almacenamiento	10^{10} bits RAM 10^{11} bits disco	10^{11} neuronas 10^{14} sinapsis
Duración de un ciclo	10^{-9} sec	10^{-3} sec
Ancho de banda	10^{10} bits/sec	10^{14} bits/sec
Memoria actualización/sec	10^9	10^{14}

Figura 1.3 Comparación básica entre los recursos de cómputo generales de que disponen los computadores (*circa* 2003) y el cerebro. Las cifras correspondientes a los computadores se han incrementado en al menos un factor 10 desde la primera edición de este libro, y se espera que suceda lo mismo en esta década. Las cifras correspondientes al cerebro no han cambiado en los últimos 10.000 años.

⁹ La ley de Moore dice que el número de transistores por pulgada cuadrada se duplica cada año o año y medio. La capacidad del cerebro humano se dobla aproximadamente cada dos o cuatro millones de años.

CONDUCTISMO

la Universidad de Leipzig. Wundt puso mucho énfasis en la realización de experimentos controlados cuidadosamente en la que sus operarios realizaban tareas de percepción o asociación al tiempo que sometían a introspección sus procesos mentales. Los meticulosos controles evolucionaron durante un largo período de tiempo hasta convertir la psicología en una ciencia, pero la naturaleza subjetiva de los datos hizo poco probable que un investigador pudiera contradecir sus propias teorías. Biólogos, estudiando el comportamiento humano, por el contrario, carecían de datos introspectivos y desarrollaron una metodología objetiva, tal y como describe H. S. Jennings (1906) en su influyente trabajo *Behavior of the Lower Organisms*. El movimiento **conductista**, liderado por John Watson (1878-1958) aplicó este punto de vista a los humanos, rechazando *cualquier* teoría en la que interviniieran procesos mentales, argumentando que la introspección no aportaba una evidencia fiable. Los conductistas insistieron en el estudio exclusivo de mediciones objetivas de percepciones (o *estímulos*) sobre animales y de las acciones resultantes (o *respuestas*). Construcciones mentales como conocimientos, creencias, objetivos y pasos en un razonamiento quedaron descartadas por ser consideradas «psicología popular» no científica. El conductismo hizo muchos descubrimientos utilizando ratas y palomas, pero tuvo menos éxito en la comprensión de los seres humanos. Aún así, su influencia en la psicología fue notable (especialmente en Estados Unidos) desde aproximadamente 1920 hasta 1960.

PSICOLOGÍA COGNITIVA

La conceptualización del cerebro como un dispositivo de procesamiento de información, característica principal de la **psicología cognitiva**, se remonta por lo menos a las obras de William James¹⁰ (1842-1910). Helmholtz también pone énfasis en que la percepción entraña cierto tipo de inferencia lógica inconsciente. Este punto de vista cognitivo se vio eclipsado por el conductismo en Estados Unidos, pero en la Unidad de Psicología Aplicada de Cambridge, dirigida por Frederic Bartlett (1886-1969), los modelos cognitivos emergieron con fuerza. La obra *The Nature of Explanation*, de Kenneth Craik (1943), discípulo y sucesor de Bartlett, re establece enérgicamente la legitimidad de términos «mentales» como creencias y objetivos, argumentando que son tan científicos como lo pueden ser la presión y la temperatura cuando se habla acerca de los gases, a pesar de que éstos estén formados por moléculas que no tienen ni presión ni temperatura. Craik establece tres elementos clave que hay que tener en cuenta para diseñar un agente basado en conocimiento: (1) el estímulo deberá ser traducido a una representación interna, (2) esta representación se debe manipular mediante procesos cognitivos para así generar nuevas representaciones internas, y (3) éstas, a su vez, se traducirán de nuevo en acciones. Dejó muy claro por qué consideraba que estos eran los requisitos idóneos para diseñar un agente:

Si el organismo tiene en su cabeza «un modelo a pequeña escala» de la realidad externa y de todas sus posibles acciones, será capaz de probar diversas opciones, decidir cuál es la mejor, planificar su reacción ante posibles situaciones futuras antes de que éstas surjan, emplear lo aprendido de experiencias pasadas en situaciones presentes y futuras, y en todo momento, reaccionar ante los imprevistos que acontezcan de manera satisfactoria, segura y más competente (Craik, 1943).

¹⁰ William James era hermano del novelista Henry James. Se comenta que Henry escribió novelas narrativas como si se tratara de psicología y William escribió sobre psicología como si se tratara de novelas narrativas.

Después de la muerte de Craik en un accidente de bicicleta en 1945, Donald Broadbent continuó su trabajo, y su libro *Perception and Communication* (1958) incluyó algunos de los primeros modelos de procesamiento de información del fenómeno psicológico. Mientras tanto, en Estados Unidos el desarrollo del modelo computacional llevó a la creación del campo de la **ciencia cognitiva**. Se puede decir que este campo comenzó en un simposio celebrado en el MIT, en septiembre de 1956 (como se verá a continuación este evento tuvo lugar sólo dos meses después de la conferencia en la que «nació» la IA). En este simposio, George Miller presentó *The Magic Number Seven*, Noam Chomsky presentó *Three Models of Language*, y Allen Newell y Herbert Simon presentaron *The Logic Theory Machine*. Estos tres artículos influyentes mostraron cómo se podían utilizar los modelos informáticos para modelar la psicología de la memoria, el lenguaje y el pensamiento lógico, respectivamente. Los psicólogos comparten en la actualidad el punto de vista común de que «la teoría cognitiva debe ser como un programa de computador» (Anderson, 1980), o dicho de otra forma, debe describir un mecanismo de procesamiento de información detallado, lo cual lleva consigo la implementación de algunas funciones cognitivas.

Ingeniería computacional (desde el año 1940 hasta el presente)

- ¿Cómo se puede construir un computador eficiente?

Para que la inteligencia artificial pueda llegar a ser una realidad se necesitan dos cosas: inteligencia y un artefacto. El computador ha sido el artefacto elegido. El computador electrónico digital moderno se inventó de manera independiente y casi simultánea por científicos en tres países involucrados en la Segunda Guerra Mundial. El equipo de Alan Turing construyó, en 1940, el primer computador *operacional* de carácter electromecánico, llamado Heath Robinson¹¹, con un único propósito: descifrar mensajes alemanes. En 1943 el mismo grupo desarrolló el Colossus, una máquina potente de propósito general basada en válvulas de vacío¹². El primer computador operacional *programable* fue el Z-3, inventado por Konrad Zuse en Alemania, en 1941. Zuse también inventó los números de coma flotante y el primer lenguaje de programación de alto nivel, Plankalkül. El primer computador *electrónico*, el ABC, fue creado por John Atanasoff junto a su discípulo Clifford Berry entre 1940 y 1942 en la Universidad Estatal de Iowa. Las investigaciones de Atanasoff recibieron poco apoyo y reconocimiento; el ENIAC, desarrollado en el marco de un proyecto militar secreto, en la Universidad de Pensilvania, por un equipo en el que trabajaban entre otros John Mauchly y John Eckert, puede considerarse como el precursor de los computadores modernos.

Desde mediados del siglo pasado, cada generación de dispositivos *hardware* ha llevado un aumento en la velocidad de proceso y en la capacidad de almacenamiento.

¹¹ Heath Robinson fue un caricaturista famoso por sus dibujos, que representaban artefactos de uso diario, caprichosos y absurdamente complicados, por ejemplo, uno para untar mantequilla en el pan tostado.

¹² En la postguerra, Turing quiso utilizar estos computadores para investigar en el campo de la IA, por ejemplo, desarrollando uno de los primeros programas para jugar a la ajedrez (Turing *et al.*, 1953). El gobierno británico bloqueó sus esfuerzos.

así como una reducción de precios. La potencia de los computadores se dobla cada 18 meses aproximadamente y seguirá a este ritmo durante una o dos décadas más. Después, se necesitará ingeniería molecular y otras tecnologías novedosas.

Por supuesto que antes de la aparición de los computadores ya había dispositivos de cálculo. Las primeras máquinas automáticas, que datan del siglo XVII, ya se mencionaron en la página seis. La primera máquina programable fue un telar, desarrollado en 1805 por Joseph Marie Jacquard (1752-1834) que utilizaba tarjetas perforadas para almacenar información sobre los patrones de los bordados. A mediados del siglo XIX, Charles Babbage (1792-1871) diseñó dos máquinas, que no llegó a construir. La «Máquina de Diferencias», que aparece en la portada de este manuscrito, se concibió con la intención de facilitar los cálculos de tablas matemáticas para proyectos científicos y de ingeniería. Finalmente se construyó y se presentó en 1991 en el Museo de la Ciencia de Londres (Swade, 1993). La «Máquina Analítica» de Babbage era mucho más ambiciosa: incluía memoria direccionable, programas almacenados y saltos condicionales; fue el primer artefacto dotado de los elementos necesarios para realizar una computación universal. Ada Lovelace, colega de Babbage e hija del poeta Lord Byron, fue seguramente la primera programadora (el lenguaje de programación Ada se llama así en honor a esta programadora). Ella escribió programas para la inacabada Máquina Analítica e incluso especuló acerca de la posibilidad de que la máquina jugara al ajedrez y compusiese música.

La IA también tiene una deuda con la parte *software* de la informática que ha proporcionado los sistemas operativos, los lenguajes de programación, y las herramientas necesarias para escribir programas modernos (y artículos sobre ellos). Sin embargo, en este área la deuda se ha saldado: la investigación en IA ha generado numerosas ideas novedosas de las que se ha beneficiado la informática en general, como por ejemplo el tiempo compartido, los intérpretes imperativos, los computadores personales con interfaces gráficas y ratones, entornos de desarrollo rápido, listas enlazadas, administración automática de memoria, y conceptos claves de la programación simbólica, funcional, dinámica y orientada a objetos.

Teoría de control y cibernética (desde el año 1948 hasta el presente)

- ¿Cómo pueden los artefactos operar bajo su propio control?

Ktesibios de Alejandría (250 a.C.) construyó la primera máquina auto controlada: un reloj de agua con un regulador que mantenía el flujo de agua circulando por él, con un ritmo constante y predecible. Esta invención cambió la definición de lo que un artefacto podía hacer. Anteriormente, solamente seres vivos podían modificar su comportamiento como respuesta a cambios en su entorno. Otros ejemplos de sistemas de control auto regulables y retroalimentados son el motor de vapor, creado por James Watt (1736-1819), y el termostato, inventado por Cornelis Drebbel (1572-1633), que también inventó el submarino. La teoría matemática de los sistemas con retroalimentación estables se desarrolló en el siglo XIX.

TEORÍA DE CONTROL

La figura central del desarrollo de lo que ahora se llama la **teoría de control** fue Norbert Wiener (1894-1964). Wiener fue un matemático brillante que trabajó en sistemas de control biológicos y mecánicos y en sus vínculos con la cognición. De la misma forma que Craik (quien también utilizó sistemas de control como modelos psicológicos), Wiener y sus colegas Arturo Rosenblueth y Julian Bigelow desafiaron la ortodoxia conductista (Rosenblueth *et al.*, 1943). Ellos veían el comportamiento determinista como algo emergente de un mecanismo regulador que intenta minimizar el «error» (la diferencia entre el estado presente y el estado objetivo). A finales de los años 40, Wiener, junto a Warren McCulloch, Walter Pitts y John von Neumann, organizaron una serie de conferencias en las que se exploraban los nuevos modelos cognitivos matemáticos y computacionales, e influyeron en muchos otros investigadores en el campo de las ciencias del comportamiento. El libro de Wiener, *Cybernetics* (1948), fue un *bestseller* y desveló al público las posibilidades de las máquinas con inteligencia artificial.

CIBERNÉTICA**FUNCIÓN OBJETIVO**

La teoría de control moderna, especialmente la rama conocida como control óptimo estocástico, tiene como objetivo el diseño de sistemas que maximizan una **función objetivo** en el tiempo. Lo cual se asemeja ligeramente a nuestra visión de lo que es la IA: diseño de sistemas que se comportan de forma óptima. ¿Por qué, entonces, IA y teoría de control son dos campos diferentes, especialmente teniendo en cuenta la cercana relación entre sus creadores? La respuesta está en el gran acoplamiento existente entre las técnicas matemáticas con las que estaban familiarizados los investigadores y entre los conjuntos de problemas que se abordaban desde cada uno de los puntos de vista. El cálculo y el álgebra matricial, herramientas de la teoría de control, se utilizaron en la definición de sistemas que se podían describir mediante conjuntos fijos de variables continuas; más aún, el análisis exacto es sólo posible en sistemas *lineales*. La IA se fundó en parte para escapar de las limitaciones matemáticas de la teoría de control en los años 50. Las herramientas de inferencia lógica y computación permitieron a los investigadores de IA afrontar problemas relacionados con el lenguaje, visión y planificación, que estaban completamente fuera del punto de mira de la teoría de control.

Lingüística (desde el año 1957 hasta el presente)

- ¿Cómo está relacionado el lenguaje con el pensamiento?

En 1957, B. F. Skinner publicó *Verbal Behavior*. La obra presentaba una visión extensa y detallada desde el enfoque conductista al aprendizaje del lenguaje, y estaba escrita por los expertos más destacados de este campo. Curiosamente, una revisión de este libro llegó a ser tan famosa como la obra misma, y provocó el casi total desinterés por el conductismo. El autor de la revisión fue Noam Chomsky, quien acababa de publicar un libro sobre su propia teoría, *Syntactic Structures*. Chomsky mostró cómo la teoría conductista no abordaba el tema de la creatividad en el lenguaje: no explicaba cómo es posible que un niño sea capaz de entender y construir oraciones que nunca antes ha escuchado. La teoría de Chomsky (basada en modelos sintácticos que se remontaban al lingüista indio Panini, aproximadamente 350 a.C.) sí podía explicar lo anterior y, a diferencia de teorías anteriores, poseía el formalismo suficiente como para permitir su programación.

La lingüística moderna y la IA «nacieron», al mismo tiempo y maduraron juntas, solapándose en un campo híbrido llamado **lingüística computacional o procesamiento del lenguaje natural**. El problema del entendimiento del lenguaje se mostró pronto mucho más complejo de lo que se había pensado en 1957. El entendimiento del lenguaje requiere la comprensión de la materia bajo estudio y de su contexto, y no solamente el entendimiento de la estructura de las sentencias. Lo cual puede parecer obvio, pero no lo fue para la mayoría de la comunidad investigadora hasta los años 60. Gran parte de los primeros trabajos de investigación en el área de la **representación del conocimiento** (el estudio de cómo representar el conocimiento de forma que el computador pueda razonar a partir de dicha representación) estaban vinculados al lenguaje y a la búsqueda de información en el campo del lenguaje, y su base eran las investigaciones realizadas durante décadas en el análisis filosófico del lenguaje.

1.3 Historia de la inteligencia artificial

Una vez revisado el material básico estamos ya en condiciones de cubrir el desarrollo de la IA propiamente dicha.

Génesis de la inteligencia artificial (1943-1955)

Warren McCulloch y Walter Pitts (1943) han sido reconocidos como los autores del primer trabajo de IA. Partieron de tres fuentes: conocimientos sobre la fisiología básica y funcionamiento de las neuronas en el cerebro, el análisis formal de la lógica proposicional de Russell y Whitehead y la teoría de la computación de Turing. Propusieron un modelo constituido por neuronas artificiales, en el que cada una de ellas se caracterizaba por estar «activada» o «desactivada»; la «activación» se daba como respuesta a la estimulación producida por una cantidad suficiente de neuronas vecinas. El estado de una neurona se veía como «equivalente, de hecho, a una proposición con unos estímulos adecuados». Mostraron, por ejemplo, que cualquier función de cómputo podría calcularse mediante alguna red de neuronas interconectadas, y que todos los conectores lógicos (*and*, *or*, *not*, etc.) se podrían implementar utilizando estructuras de red sencillas. McCulloch y Pitts también sugirieron que redes adecuadamente definidas podrían aprender. Donald Hebb (1949) propuso y demostró una sencilla regla de actualización para modificar las intensidades de las conexiones entre neuronas. Su regla, ahora llamada **de aprendizaje Hebbiano o de Hebb**, sigue vigente en la actualidad.

Dos estudiantes graduados en el Departamento de Matemáticas de Princeton, Marvin Minsky y Dean Edmonds, construyeron el primer computador a partir de una red neuronal en 1951. El SNARC, como se llamó, utilizaba 3.000 válvulas de vacío y un mecanismo de piloto automático obtenido de los desechos de un avión bombardero B-24 para simular una red con 40 neuronas. El comité encargado de evaluar el doctorado de Minsky veía con escepticismo el que este tipo de trabajo pudiera considerarse como matemático, pero se dice que von Newmann dijo, «Si no lo es actualmente, algún día lo será».

Minsky posteriormente probó teoremas influyentes que mostraron las limitaciones de la investigación con redes neuronales.

Hay un número de trabajos iniciales que se pueden caracterizar como de IA, pero fue Alan Turing quien articuló primero una visión de la IA en su artículo *Computing Machinery and Intelligence*, en 1950. Ahí, introdujo la prueba de Turing, el aprendizaje automático, los algoritmos genéricos y el aprendizaje por refuerzo.

Nacimiento de la inteligencia artificial (1956)

Princeton acogió a otras de las figuras señeras de la IA, John McCarthy. Posteriormente a su graduación, McCarthy se transladó al Dartmouth College, que se erigiría en el lugar del nacimiento oficial de este campo. McCarthy convenció a Minsky, Claude Shannon y Nathaniel Rochester para que le ayudaran a aumentar el interés de los investigadores americanos en la teoría de autómatas, las redes neuronales y el estudio de la inteligencia. Organizaron un taller con una duración de dos meses en Dartmouth en el verano de 1956. Hubo diez asistentes en total, entre los que se incluían Trenchard More de Princeton, Arthur Samuel de IBM, y Ray Solomonoff y Oliver Selfridge del MIT.

Dos investigadores del Carnegie Tech¹³, Allen Newell y Herbert Simon, acapararon la atención. Si bien los demás también tenían algunas ideas y, en algunos casos, programas para aplicaciones determinadas como el juego de damas, Newell y Simon contaban ya con un programa de razonamiento, el Teórico Lógico (TL), del que Simon afirmaba: «Hemos inventado un programa de computación capaz de pensar de manera no numérica, con lo que ha quedado resuelto el venerable problema de la dualidad mente-cuerpo»¹⁴. Poco después del término del taller, el programa ya era capaz de demostrar gran parte de los teoremas del Capítulo 2 de *Principia Matemática* de Russell y Whitehead. Se dice que Russell se manifestó complacido cuando Simon le mostró que la demostración de un teorema que el programa había generado era más corta que la que aparecía en *Principia*. Los editores de la revista *Journal of Symbolic Logic* resultaron menos impresionados y rechazaron un artículo cuyos autores eran Newell, Simon y el Teórico Lógico (TL).

El taller de Dartmouth no produjo ningún avance notable, pero puso en contacto a las figuras importantes de este campo. Durante los siguientes 20 años, el campo estuvo dominado por estos personajes, así como por sus estudiantes y colegas del MIT, CMU, Stanford e IBM. Quizá lo último que surgió del taller fue el consenso en adoptar el nuevo nombre propuesto por McCarthy para este campo: **Inteligencia Artificial**. Quizá «racionalidad computacional» hubiese sido más adecuado, pero «IA» se ha mantenido.

Revisando la propuesta del taller de Dartmouth (McCarthy *et al.*, 1955), se puede apreciar por qué fue necesario para la IA convertirse en un campo separado. ¿Por qué

¹³ Actualmente Universidad Carnegie Mellon (UCM).

¹⁴ Newell y Simon también desarrollaron un lenguaje de procesamiento de listas, IPL, para poder escribir el TL. No disponían de un compilador y lo tradujeron a código máquina a mano. Para evitar errores, trabajaron en paralelo, diciendo en voz alta números binarios, conforme escribían cada instrucción para asegurarse de que ambos coincidían.

no todo el trabajo hecho en el campo de la IA se ha realizado bajo el nombre de teoría de control, o investigación operativa, o teoría de la decisión, que, después de todo, persiguen objetivos similares a los de la IA? O, ¿por qué no es la IA una rama de las matemáticas? La primera respuesta es que la IA desde el primer momento abarcó la idea de duplicar facultades humanas como la creatividad, la auto-mejora y el uso del lenguaje. Ninguno de los otros campos tenían en cuenta esos temas. La segunda respuesta está relacionada con la metodología. La IA es el único de estos campos que es claramente una rama de la informática (aunque la investigación operativa comparte el énfasis en la simulación por computador), además la IA es el único campo que persigue la construcción de máquinas que funcionen automáticamente en medios complejos y cambiantes.

Entusiasmo inicial, grandes esperanzas (1952-1969)

Los primeros años de la IA estuvieron llenos de éxitos (aunque con ciertas limitaciones). Teniendo en cuenta lo primitivo de los computadores y las herramientas de programación de aquella época, y el hecho de que sólo unos pocos años antes, a los computadores se les consideraba como artefactos que podían realizar trabajos aritméticos y nada más, resultó sorprendente que un computador hiciese algo remotamente inteligente. La comunidad científica, en su mayoría, prefirió creer que «una máquina nunca podría hacer *tareas*» (véase el Capítulo 26 donde aparece una extensa lista de *tareas* recopilada por Turing). Naturalmente, los investigadores de IA responderían demostrando la realización de una *tarea* tras otra. John McCarthy se refiere a esta época como la era de «¡Mira, mamá, ahora sin manos!».

Al temprano éxito de Newell y Simon siguió el del sistema de resolución general de problemas, o SRGP. A diferencia del Teórico Lógico, desde un principio este programa se diseñó para que imitara protocolos de resolución de problemas de los seres humanos. Dentro del limitado número de puzzles que podía manejar, resultó que la secuencia en la que el programa consideraba que los subobjetivos y las posibles acciones eran semejantes a la manera en que los seres humanos abordaban los mismos problemas. Es decir, el SRGP posiblemente fue el primer programa que incorporó el enfoque de «pensar como un ser humano». El éxito del SRGP y de los programas que le siguieron, como los modelos de cognición, llevaron a Newell y Simon (1976) a formular la famosa hipótesis del **sistema de símbolos físicos**, que afirma que «un sistema de símbolos físicos tiene los medios suficientes y necesarios para generar una acción inteligente». Lo que ellos querían decir es que cualquier sistema (humano o máquina) que exhibiese inteligencia debería operar manipulando estructuras de datos compuestas por símbolos. Posteriormente se verá que esta hipótesis se ha modificado atendiendo a distintos puntos de vista.

En IBM, Nathaniel Rochester y sus colegas desarrollaron algunos de los primeros programas de IA. Herbert Gelernter (1959) construyó el demostrador de teoremas de geometría (DTG), el cual era capaz de probar teoremas que muchos estudiantes de matemáticas podían encontrar muy complejos de resolver. A comienzos 1952, Arthur Samuel escribió una serie de programas para el juego de las damas que eventualmente aprendieron a jugar hasta alcanzar un nivel equivalente al de un *amateur*. De paso, echó por

tierra la idea de que los computadores sólo pueden hacer lo que se les dice: su programa pronto aprendió a jugar mejor que su creador. El programa se presentó en la televisión en febrero de 1956 y causó una gran impresión. Como Turing, Samuel tenía dificultades para obtener el tiempo de cómputo. Trabajaba por las noches y utilizaba máquinas que aún estaban en período de prueba en la planta de fabricación de IBM. El Capítulo 6 trata el tema de los juegos, y en el Capítulo 21 se describe con detalle las técnicas de aprendizaje utilizadas por Samuel.

John McCarthy se trasladó de Dartmouth al MIT, donde realizó tres contribuciones cruciales en un año histórico: 1958. En el Laboratorio de IA del MIT Memo Número 1, McCarthy definió el lenguaje de alto nivel **Lisp**, que se convertiría en el lenguaje de programación dominante en la IA. Lisp es el segundo lenguaje de programación más antiguo que se utiliza en la actualidad, ya que apareció un año después de FORTRAN. Con Lisp, McCarthy tenía la herramienta que necesitaba, pero el acceso a los escasos y costosos recursos de cómputo aún era un problema serio. Para solucionarlo, él, junto a otros miembros del MIT, inventaron el tiempo compartido. También, en 1958, McCarthy publicó un artículo titulado *Programs with Common Sense*, en el que describía el Generador de Consejos, un programa hipotético que podría considerarse como el primer sistema de IA completo. Al igual que el Teórico Lógico y el Demostrador de Teoremas de Geometría, McCarthy diseñó su programa para buscar la solución a problemas utilizando el conocimiento. Pero, a diferencia de los otros, manejaba el conocimiento general del mundo. Por ejemplo, mostró cómo algunos axiomas sencillos permitían a un programa generar un plan para conducirnos hasta el aeropuerto y tomar un avión. El programa se diseñó para que aceptase nuevos axiomas durante el curso normal de operación, permitiéndole así ser competente en áreas nuevas, sin *necesidad de reprogramación*. El Generador de Consejos incorporaba así los principios centrales de la representación del conocimiento y el razonamiento: es útil contar con una representación formal y explícita del mundo y de la forma en que la acción de un agente afecta al mundo, así como, ser capaces de manipular estas representaciones con procesos deductivos. Es sorprendente constatar cómo mucho de lo propuesto en el artículo escrito en 1958 permanece vigente incluso en la actualidad.

1958 fue el año en el que Marvin Minsky se trasladó al MIT. Sin embargo, su colaboración inicial no duró demasiado. McCarthy se centró en la representación y el razonamiento con lógica formal, mientras que Minsky estaba más interesado en lograr que los programas funcionaran y eventualmente desarrolló un punto de vista anti-lógico. En 1963 McCarthy creó el Laboratorio de IA en Stanford. Su plan para construir la versión más reciente del Generador de Consejos con ayuda de la lógica sufrió un considerable impulso gracias al descubrimiento de J. A. Robinson del método de resolución (un algoritmo completo para la demostración de teoremas para la lógica de primer orden; véase el Capítulo 9). El trabajo realizado en Stanford hacía énfasis en los métodos de propósito general para el razonamiento lógico. Algunas aplicaciones de la lógica incluían los sistemas de planificación y respuesta a preguntas de Cordell Green (1969b), así como el proyecto de robótica de Shakey en el nuevo Instituto de Investigación de Stanford (Stanford Research Institute, SRI). Este último proyecto, comentado en detalle en el Capítulo 25, fue el primero que demostró la total integración del razonamiento lógico y la actividad física.

Minsky supervisó el trabajo de una serie de estudiantes que eligieron un número de problemas limitados cuya solución pareció requerir inteligencia. Estos dominios limi-

MICROMUNDOS

tados se conocen como **micromundos**. El programa SAINT de James Slagle (1963a) fue capaz de resolver problemas de integración de cálculo en forma cerrada, habituales en los primeros cursos de licenciatura. El programa ANALOGY de Tom Evans (1968) resolvía problemas de analogía geométrica que se aplicaban en las pruebas de medición de inteligencia, semejante al de la Figura 1.4. El programa STUDENT de Daniel Bobrow (1967) podía resolver problemas de álgebra del tipo:

Si el número de clientes de Tom es dos veces el cuadrado del 20 por ciento de la cantidad de anuncios que realiza, y éstos ascienden a 45, ¿cuántos clientes tiene Tom?

El micromundo más famoso fue el mundo de los bloques, que consiste en un conjunto de bloques sólidos colocados sobre una mesa (más frecuentemente, en la simulación de ésta), como se muestra en la Figura 1.5. Una tarea típica de este mundo es la reordenación de los bloques de cierta manera, con la ayuda de la mano de un robot que es capaz de tomar un bloque cada vez. El mundo de los bloques fue el punto de partida para el proyecto de visión de David Huffman (1971), la visión y el trabajo de propagación con restricciones de David Waltz (1975), la teoría del aprendizaje de Patrick Winston (1970), del programa para la comprensión de lenguaje natural de Terry Winograd (1972) y del planificador de Scott Fahlman (1974).

El trabajo realizado por McCulloch y Pitts con redes neuronales hizo florecer esta área. El trabajo de Winograd y Cowan (1963) mostró cómo un gran número de elementos podría representar un concepto individual de forma colectiva, lo cual llevaba consigo un aumento proporcional en robustez y paralelismo. Los métodos de aprendizaje de Hebb se reforzaron con las aportaciones de Bernie Widrow (Widrow y Hoff, 1960; Widrow, 1962), quien llamó **adalines** a sus redes, y por Frank Rosenblatt (1962) con sus **perceptrones**. Rosenblatt demostró el famoso teorema del perceptrón, con lo que mostró que su algoritmo de aprendizaje podría ajustar las intensidades de las conexiones de un perceptrón para que se adaptaran a los datos de entrada, siempre y cuando existiera una correspondencia. Estos temas se explicarán en el Capítulo 20.

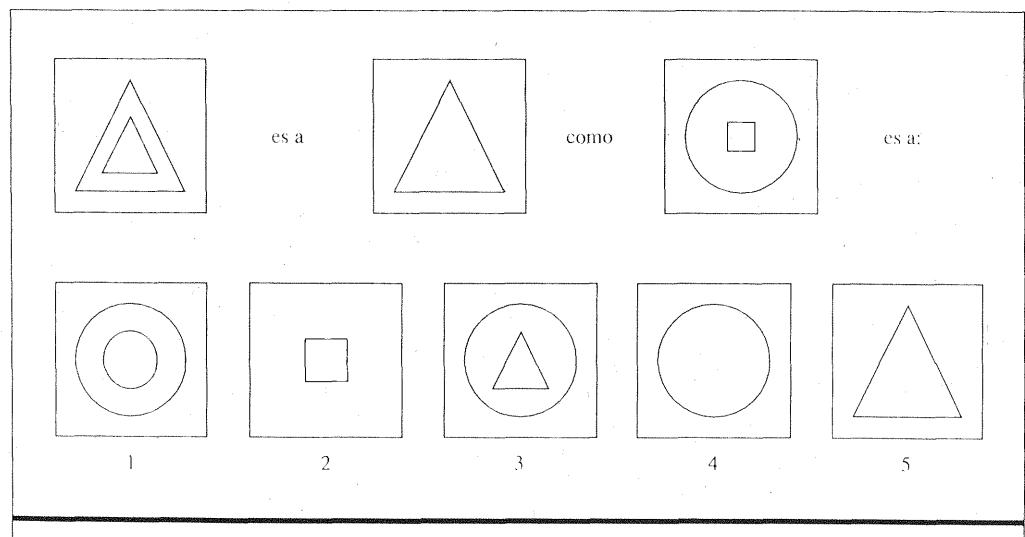


Figura 1.4 Ejemplo de un problema resuelto por el programa ANALOGY de Evans.

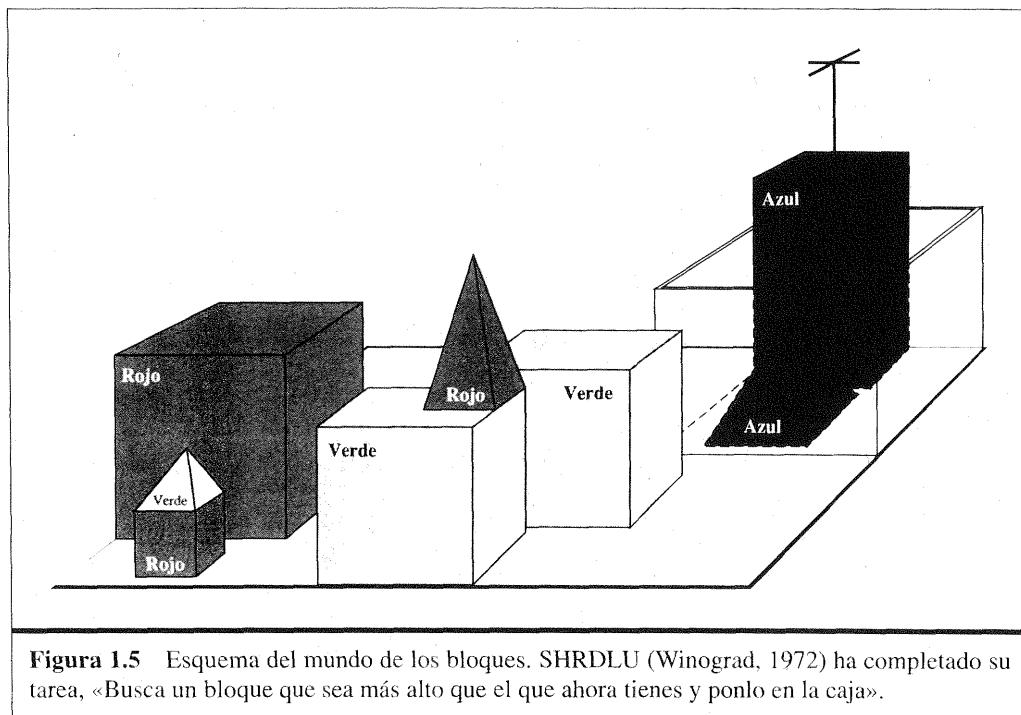


Figura 1.5 Esquema del mundo de los bloques. SHRDLU (Winograd, 1972) ha completado su tarea, «Busca un bloque que sea más alto que el que ahora tienes y ponlo en la caja».

Una dosis de realidad (1966-1973)

Desde el principio, los investigadores de IA hicieron públicas, sin timidez, predicciones sobre el éxito que les esperaba. Con frecuencia, se cita el siguiente comentario realizado por Herbert Simon en 1957:

Sin afán de sorprenderlos y dejarlos atónitos, pero la forma más sencilla que tengo de resumirlo es diciéndoles que actualmente en el mundo existen máquinas capaces de pensar, aprender y crear. Además, su aptitud para hacer lo anterior aumentará rápidamente hasta que (en un futuro previsible) la magnitud de problemas que serán capaces de resolver irá a la par que la capacidad de la mente humana para hacer lo mismo.

Términos como «futuro previsible» pueden interpretarse de formas distintas, pero Simon también hizo predicciones más concretas: como que en diez años un computador llegaría a ser campeón de ajedrez, y que se podría demostrar un importante teorema matemático con una máquina. Estas predicciones se cumplirían (al menos en parte) dentro de los siguientes 40 años y no en diez. El exceso de confianza de Simon se debió a la prometedora actuación de los primeros sistemas de IA en problemas simples. En la mayor parte de los casos resultó que estos primeros sistemas fallaron estrepitosamente cuando se utilizaron en problemas más variados o de mayor dificultad.

El primer tipo de problemas surgió porque la mayoría de los primeros programas contaban con poco o ningún conocimiento de las materias objeto de estudio; obtenían resultados gracias a sencillas manipulaciones sintácticas. Una anécdota típica tuvo lugar cuando se comenzaba a trabajar en la traducción automática, actividad que recibía un

generoso patrocinio del Consejo Nacional para la Investigación de Estados Unidos en un intento de agilizar la traducción de artículos científicos rusos en vísperas del lanzamiento del Sputnik en 1957. Al principio se consideró que todo se reduciría a sencillas transformaciones sintácticas apoyadas en las gramáticas rusas e inglesa y al emplazamiento de palabras mediante un diccionario electrónico, lo que bastaría para obtener el significado exacto de las oraciones. La realidad es que para traducir es necesario contar con un conocimiento general sobre el tema, que permita resolver ambigüedades y así, precisar el contenido de una oración. La famosa retraducción del ruso al inglés de la frase «el espíritu es fuerte pero la carne es débil», cuyo resultado fue «el vodka es bueno pero la carne está podrida» es un buen ejemplo del tipo de dificultades que surgieron. En un informe presentado en 1966, el comité consultivo declaró que «no se ha logrado obtener ninguna traducción de textos científicos generales ni se prevé obtener ninguna en un futuro inmediato». Se canceló todo el patrocinio del gobierno estadounidense que se había asignado a los proyectos académicos sobre traducción. Hoy día, la traducción automática es una herramienta imperfecta pero de uso extendido en documentos técnicos, comerciales, gubernamentales y de Internet.

El segundo problema fue que muchos de los problemas que se estaban intentando resolver mediante la IA eran intratables. La mayoría de los primeros programas de IA resolvían problemas experimentando con diversos pasos hasta que se llegara a encontrar una solución. Esto funcionó en los primeros programas debido a que los micromundos con los que se trabajaba contenían muy pocos objetos, y por lo tanto muy pocas acciones posibles y secuencias de soluciones muy cortas. Antes de que se desarrollara la teoría de la complejidad computacional, se creía que para «aumentar» el tamaño de los programas de forma que estos pudiesen solucionar grandes problemas sería necesario incrementar la velocidad del *hardware* y aumentar las memorias. El optimismo que acompañó al logro de la demostración de problemas, por ejemplo, pronto se vio eclipsado cuando los investigadores fracasaron en la demostración de teoremas que implicaban más de unas pocas decenas de condiciones. *El hecho de que, en principio, un programa sea capaz de encontrar una solución no implica que tal programa encierre todos los mecanismos necesarios para encontrar la solución en la práctica.*

La ilusoria noción de una ilimitada capacidad de cómputo no sólo existió en los programas para la resolución de problemas. Los primeros experimentos en el campo de la **evolución automática** (ahora llamados **algoritmos genéticos**) (Friedberg, 1958; Friedberg *et al.*, 1959) estaban basados en la, sin duda correcta, premisa de que efectuando una adecuada serie de pequeñas mutaciones a un programa de código máquina se podría generar un programa con buen rendimiento aplicable en cualquier tarea sencilla. Después surgió la idea de probar con mutaciones aleatorias aplicando un proceso de selección con el fin de conservar aquellas mutaciones que hubiesen demostrado ser más útiles. No obstante, las miles de horas de CPU dedicadas, no dieron lugar a ningún avance tangible. Los algoritmos genéticos actuales utilizan representaciones mejores y han tenido más éxito.

La incapacidad para manejar la «explosión combinatoria» fue una de las principales críticas que se hicieron a la IA en el informe de Lighthill (Lighthill, 1973), informe en el que se basó la decisión del gobierno británico para retirar la ayuda a las investigaciones sobre IA, excepto en dos universidades. (La tradición oral presenta un cuadro un

poco distinto y más animado, en el que se vislumbran ambiciones políticas y animadversiones personales, cuya descripción está fuera del ámbito de esta obra.)

El tercer obstáculo se derivó de las limitaciones inherentes a las estructuras básicas que se utilizaban en la generación de la conducta inteligente. Por ejemplo, en 1969, en el libro de Minsky y Papert, *Perceptrons*, se demostró que si bien era posible lograr que los perceptrones (una red neuronal simple) aprendieran cualquier cosa que pudiesen representar, su capacidad de representación era muy limitada. En particular, un perceptrón con dos entradas no se podía entrenar para que aprendiese a reconocer cuándo sus dos entradas eran diferentes. Si bien los resultados que obtuvieron no eran aplicables a redes más complejas multicapa, los fondos para la investigación de las redes neuronales se redujeron a prácticamente nada. Es irónico que los nuevos algoritmos de aprendizaje de retroalimentación utilizados en las redes multicapa y que fueron la causa del gran resurgimiento de la investigación en redes neuronales de finales de los años 80, en realidad, se hayan descubierto por primera vez en 1969 (Bryson y Ho, 1969).

Sistemas basados en el conocimiento: ¿clave del poder? (1969-1979)

El cuadro que dibujaba la resolución de problemas durante la primera década de la investigación en la IA estaba centrado en el desarrollo de mecanismos de búsqueda de propósito general, en los que se entrelazaban elementos de razonamiento básicos para encontrar así soluciones completas. A estos procedimientos se les ha denominado **métodos débiles**, debido a que no tratan problemas más amplios o más complejos. La alternativa a los métodos débiles es el uso de conocimiento específico del dominio que facilita el desarrollo de etapas de razonamiento más largas, pudiéndose así resolver casos recurrentes en dominios de conocimiento restringido. Podría afirmarse que para resolver un problema en la práctica, es necesario saber de antemano la correspondiente respuesta.

El programa DENDRAL (Buchanan *et al.*, 1969) constituye uno de los primeros ejemplos de este enfoque. Fue diseñado en Stanford, donde Ed Feigenbaum (discípulo de Herbert Simon), Bruce Buchanan (filósofo convertido en informático) y Joshua Lederberg (genetista ganador del Premio Nobel) colaboraron en la solución del problema de inferir una estructura molecular a partir de la información proporcionada por un espectrómetro de masas. El programa se alimentaba con la fórmula elemental de la molécula (por ejemplo, $C_6H_{13}NO_2$) y el espectro de masas, proporcionando las masas de los distintos fragmentos de la molécula generada después de ser bombardeada con un haz de electrones. Por ejemplo, un espectro de masas con un pico en $m = 15$, correspondería a la masa de un fragmento de metilo (CH_3).

La versión más simple del programa generaba todas las posibles estructuras que correspondieran a la fórmula, luego predecía el espectro de masas que se observaría en cada caso, y comparaba éstos con el espectro real. Como era de esperar, el método anterior resultó pronto inviable para el caso de moléculas con un tamaño considerable. Los creadores de DENDRAL consultaron con químicos analíticos y se dieron cuenta de que éstos trabajaban buscando patrones conocidos de picos en el espectro que sugerían estructuras comunes en la molécula. Por ejemplo, para identificar el subgrupo (con un peso de 28) de las cetonas ($C=O$) se empleó la siguiente regla:

si hay dos picos en x_1 y x_2 tales que

- a) $x_1 + x_2 = M + 28$ (siendo M la masa de toda la molécula);
- b) $x_1 = 28$ es un pico alto;
- c) $x_2 = 28$ es un pico alto;
- d) al menos una de x_1 y x_2 es alta.

entonces existe un subgrupo de cetonas

Al reconocer que la molécula contiene una subestructura concreta se reduce el número de posibles candidatos de forma considerable. La potencia de DENDRAL se basaba en que:

Toda la información teórica necesaria para resolver estos problemas se ha proyectado desde su forma general [componente predicho por el espectro] («primeros principios») a formas eficientes especiales («recetas de cocina»). (Feigenbaum *et al.*, 1971)

La trascendencia de DENDRAL se debió a ser el primer sistema de *conocimiento intenso* que tuvo éxito: su base de conocimiento estaba formada por grandes cantidades de reglas de propósito particular. En sistemas diseñados posteriormente se incorporaron también los elementos fundamentales de la propuesta de McCarthy para el Generador de Consejos, la nítida separación del conocimiento (en forma de reglas) de la parte correspondiente al razonamiento.

Teniendo en cuenta esta lección, Feigenbaum junto con otros investigadores de Stanford dieron comienzo al Proyecto de Programación Heurística, PPH, dedicado a determinar el grado con el que la nueva metodología de los **sistemas expertos** podía aplicarse a otras áreas de la actividad humana. El siguiente gran esfuerzo se realizó en el área del diagnóstico médico. Feigenbaum, Buchanan y el doctor Edward Shortliffe diseñaron el programa MYCIN, para el diagnóstico de infecciones sanguíneas. Con 450 reglas aproximadamente, MYCIN era capaz de hacer diagnósticos tan buenos como los de un experto y, desde luego, mejores que los de un médico recién graduado. Se distinguía de DENDRAL en dos aspectos principalmente. En primer lugar, a diferencia de las reglas de DENDRAL, no se contaba con un modelo teórico desde el cual se pudiesen deducir las reglas de MYCIN. Fue necesario obtenerlas a partir de extensas entrevistas con los expertos, quienes las habían obtenido de libros de texto, de otros expertos o de su experiencia directa en casos prácticos. En segundo lugar, las reglas deberían reflejar la incertidumbre inherente al conocimiento médico. MYCIN contaba con un elemento que facilitaba el cálculo de incertidumbre denominado **factores de certeza** (véase el Capítulo 13), que al parecer (en aquella época) correspondía muy bien a la manera como los médicos ponderaban las evidencias al hacer un diagnóstico.

La importancia del conocimiento del dominio se demostró también en el área de la comprensión del lenguaje natural. Aunque el sistema SHRDLU de Winograd para la comprensión del lenguaje natural había suscitado mucho entusiasmo, su dependencia del análisis sintáctico provocó algunos de los mismos problemas que habían aparecido en los trabajos realizados en la traducción automática. Era capaz de resolver los problemas de ambigüedad e identificar los pronombres utilizados, gracias a que se había diseñado especialmente para un área (el mundo de los bloques). Fueron varios los investigadores que, como Eugene Charniak, estudiante de Winograd en el MIT, opinaron que para una sólida comprensión del lenguaje era necesario contar con un conocimiento general sobre el mundo y un método general para usar ese conocimiento.

En Yale, el lingüista transformado en informático Roger Schank reforzó lo anterior al afirmar: «No existe eso que llaman sintaxis», lo que irritó a muchos lingüistas, pero sirvió para iniciar un útil debate. Schank y sus estudiantes diseñaron una serie de programas (Schank y Abelson, 1977; Wilensky, 1978; Schank y Riesbeck, 1981; Dyer, 1983) cuyo objetivo era la comprensión del lenguaje natural. El foco de atención estaba menos en el lenguaje *per se* y más en los problemas vinculados a la representación y razonamiento del conocimiento necesario para la comprensión del lenguaje. Entre los problemas estaba el de la representación de situaciones estereotipo (Cullingford, 1981), la descripción de la organización de la memoria humana (Rieger, 1976; Kolodner, 1983) y la comprensión de planes y objetivos (Wilensky, 1983).

El crecimiento generalizado de aplicaciones para solucionar problemas del mundo real provocó el respectivo aumento en la demanda de esquemas de representación del conocimiento que funcionaran. Se desarrolló una considerable cantidad de lenguajes de representación y razonamiento diferentes. Algunos basados en la lógica, por ejemplo el lenguaje Prolog gozó de mucha aceptación en Europa, aceptación que en Estados Unidos fue para la familia del PLANNER. Otros, siguiendo la noción de **Marcos** de Minsky (1975), se decidieron por un enfoque más estructurado, al recopilar información sobre objetos concretos y tipos de eventos, organizando estos tipos en grandes jerarquías taxonómicas, similares a las biológicas.

MARCOS

La IA se convierte en una industria (desde 1980 hasta el presente)

El primer sistema experto comercial que tuvo éxito, R1, inició su actividad en Digital Equipment Corporation (McDermott, 1982). El programa se utilizaba en la elaboración de pedidos de nuevos sistemas informáticos. En 1986 representaba para la compañía un ahorro estimado de 40 millones de dólares al año. En 1988, el grupo de Inteligencia Artificial de DEC había distribuido ya 40 sistemas expertos, y había más en camino. Du Pont utilizaba ya 100 y estaban en etapa de desarrollo 500 más, lo que le generaba ahorro de diez millones de dólares anuales aproximadamente. Casi todas las compañías importantes de Estados Unidos contaban con su propio grupo de IA, en el que se utilizaban o investigaban sistemas expertos.

En 1981 los japoneses anunciaron el proyecto «Quinta Generación», un plan de diez años para construir computadores inteligentes en los que pudiese ejecutarse Prolog. Como respuesta Estados Unidos constituyó la Microelectronics and Computer Technology Corporation (MCC), consorcio encargado de mantener la competitividad nacional en estas áreas. En ambos casos, la IA formaba parte de un gran proyecto que incluía el diseño de chips y la investigación de la relación hombre máquina. Sin embargo, los componentes de IA generados en el marco de MCC y del proyecto Quinta Generación nunca alcanzaron sus objetivos. En el Reino Unido, el informe Alvey restauró el patrocinio suspendido por el informe Lighthill¹⁵.

¹⁵ Para evitar confusiones, se creó un nuevo campo denominado Sistemas Inteligentes Basados en Conocimiento (IKBS, *Intelligent Knowledge-Based Systems*) ya que la Inteligencia Artificial había sido oficialmente cancelada.

En su conjunto, la industria de la IA creció rápidamente, pasando de unos pocos millones de dólares en 1980 a billones de dólares en 1988. Poco después de este período llegó la época llamada «El Invierno de la IA», que afectó a muchas empresas que no fueron capaces de desarrollar los extravagantes productos prometidos.

Regreso de las redes neuronales (desde 1986 hasta el presente)

Aunque la informática había abandonado de manera general el campo de las redes neuronales a finales de los años 70, el trabajo continuó en otros campos. Físicos como John Hopfield (1982) utilizaron técnicas de la mecánica estadística para analizar las propiedades de almacenamiento y optimización de las redes, tratando colecciones de nodos como colecciones de átomos. Psicólogos como David Rumelhart y Geoff Hinton continuaron con el estudio de modelos de memoria basados en redes neuronales. Como se verá en el Capítulo 20, el impulso más fuerte se produjo a mediados de la década de los 80, cuando por lo menos cuatro grupos distintos reinventaron el algoritmo de aprendizaje de retroalimentación, mencionado por vez primera en 1969 por Bryson y Ho. El algoritmo se aplicó a diversos problemas de aprendizaje en los campos de la informática y la psicología, y la gran difusión que conocieron los resultados obtenidos, publicados en la colección *Parallel Distributed Processing* (Rumelhart y McClelland, 1986), suscitó gran entusiasmo.

CONEXIONISTAS

Aquellos modelos de inteligencia artificial llamados **conexionistas**¹⁶ fueron vistos por algunos como competidores tanto de los modelos simbólicos propuestos por Newell y Simon como de la aproximación lógica de McCarthy entre otros (Smolensky, 1988). Puede parecer obvio que los humanos manipulan símbolos hasta cierto nivel, de hecho, el libro *The Symbolic Species* (1997) de Terrence Deacon sugiere que esta es la *característica que define* a los humanos, pero los conexiónistas más ardientes se preguntan si la manipulación de los símbolos desempeña algún papel justificable en determinados modelos de cognición. Este interrogante no ha sido aún clarificado, pero la tendencia actual es que las aproximaciones conexiónistas y simbólicas son complementarias y no competidoras.

IA se convierte en una ciencia (desde 1987 hasta el presente)

En los últimos años se ha producido una revolución tanto en el contenido como en la metodología de trabajo en el campo de la inteligencia artificial.¹⁷ Actualmente es más usual el desarrollo sobre teorías ya existentes que proponer teorías totalmente novedo-

¹⁶ Se usa la traducción literal del término *connectionist* por no existir un término equivalente en español (*N. del RT*).

¹⁷ Hay quien ha caracterizado este cambio como la victoria de los pulcros (aquellos que consideran que las teorías de IA deben basarse rigurosamente en las matemáticas) sobre los desaliñados (aquellos que después de intentar muchas ideas, escriben algunos programas y después evalúan las que aparentemente funcionan). Ambos enfoques son útiles. Esta tendencia en favor de una mayor pulcritud es señal de que el campo ha alcanzado cierto nivel de estabilidad y madurez. Lo cual no implica que tal estabilidad se puede ver alterada con el surgimiento de otras ideas poco estructuradas.

sas, tomar como base rigurosos teoremas o sólidas evidencias experimentales más que intuición, y demostrar la utilidad de las aplicaciones en el mundo real más que crear ejemplos de juguete.

La IA se fundó en parte en el marco de una rebelión en contra de las limitaciones de los campos existentes como la teoría de control o la estadística, y ahora abarca estos campos. Tal y como indica David McAllester (1998),

En los primeros años de la IA parecía perfectamente posible que las nuevas formas de la computación simbólica; por ejemplo, los marcos y las redes semánticas, hicieran que la mayor parte de la teoría clásica pasara a ser obsoleta. Esto llevó a la IA a una especie de aislamiento, que la separó del resto de las ciencias de la computación. En la actualidad se está abandonando este aislamiento. Existe la creencia de que el aprendizaje automático no se debe separar de la teoría de la información, que el razonamiento incierto no se debe separar de los modelos estocásticos, de que la búsqueda no se debe aislar de la optimización clásica y el control, y de que el razonamiento automático no se debe separar de los métodos formales y del análisis estático.

En términos metodológicos, se puede decir, con rotundidad, que la IA ya forma parte del ámbito de los métodos científicos. Para que se acepten, las hipótesis se deben someter a rigurosos experimentos empíricos, y los resultados deben analizarse estadísticamente para identificar su relevancia (Cohen, 1995). El uso de Internet y el compartir repositorios de datos de prueba y código, ha hecho posible que ahora se puedan contrastar experimentos.

Un buen modelo de la tendencia actual es el campo del reconocimiento del habla. En la década de los 70 se sometió a prueba una gran variedad de arquitecturas y enfoques. Muchos de ellos fueron un tanto *ad hoc* y resultaban frágiles, y fueron probados sólo en unos pocos ejemplos elegidos especialmente. En años recientes, las aproximaciones basadas en los **modelos de Markov ocultos**, MMO, han pasado a dominar el área. Dos son las características de los MMO que tienen relevancia. Primero, se basan en una rigurosa teoría matemática, lo cual ha permitido a los investigadores del lenguaje basarse en los resultados de investigaciones matemáticas hechas en otros campos a lo largo de varias décadas. En segundo lugar, los modelos se han generado mediante un proceso de aprendizaje en grandes *corpus* de datos de lenguaje reales. Lo cual garantiza una funcionalidad robusta, y en sucesivas pruebas ciegas, los MMO han mejorado sus resultados a un ritmo constante. La tecnología del habla y el campo relacionado del reconocimiento de caracteres manuscritos están actualmente en transición hacia una generalizada utilización en aplicaciones industriales y de consumo.

Las redes neuronales también siguen esta tendencia. La mayor parte del trabajo realizado con redes neuronales en la década de los 80 se realizó con la idea de dejar a un lado lo que se podía hacer y de descubrir en qué se diferenciaban las redes neuronales de otras técnicas «tradicionales». La utilización de metodologías mejoradas y marcos teóricos, ha autorizado que este campo alcance un grado de conocimiento que ha permitido que ahora las redes neuronales se puedan comparar con otras técnicas similares de campos como la estadística, el reconocimiento de patrones y el aprendizaje automático, de forma que las técnicas más prometedoras pueden aplicarse a cualquier problema. Como resultado de estos desarrollos, la tecnología denominada **minería de datos** ha generado una nueva y vigorosa industria.

La aparición de *Probabilistic Reasoning in Intelligent Systems* de Judea Pearl (1988) hizo que se aceptara de nuevo la probabilidad y la teoría de la decisión como parte de la IA, como consecuencia del resurgimiento del interés despertado y gracias especialmente al artículo *In Defense of Probability* de Peter Cheeseman (1985). El formalismo de las **redes de Bayes** apareció para facilitar la representación eficiente y el razonamiento riguroso en situaciones en las que se disponía de conocimiento incierto. Este enfoque supera con creces muchos de los problemas de los sistemas de razonamiento probabilístico de las décadas de los 60 y 70; y ahora domina la investigación de la IA en el razonamiento incierto y los sistemas expertos. Esta aproximación facilita el aprendizaje a partir de la experiencia, y combina lo mejor de la IA clásica y las redes neuronales. El trabajo de Judea Pearl (1982a) y de Eric Horvitz y David Heckerman (Horvitz y Heckerman, 1986; Horvitz *et al.*, 1986) sirvió para promover la noción de sistemas expertos *normativos*: es decir, los que actúan racionalmente de acuerdo con las leyes de la teoría de la decisión, sin que intenten imitar las etapas de razonamiento de los expertos humanos. El sistema operativo Windows™ incluye varios sistemas expertos de diagnóstico normativos para la corrección de problemas. En los Capítulos 13 y 16 se aborda este tema.

Revoluciones similares y suaves se han dado en robótica, visión por computador, y aprendizaje automático. La comprensión mejor de los problemas y de su complejidad, junto a un incremento en la sofisticación de las matemáticas ha facilitado el desarrollo de una agenda de investigación y de métodos más robustos. En cualquier caso, la formalización y especialización ha llevado también a la fragmentación: áreas como la visión y la robótica están cada vez más aislados de la «rama central» de la IA. La concepción unificadora de IA como diseño de agentes racionales puede facilitar la unificación de estos campos diferentes.

Emergencia de los sistemas inteligentes (desde 1995 hasta el presente)

Quizás animados por el progreso en la resolución de subproblemas de IA, los investigadores han comenzado a trabajar de nuevo en el problema del «agente total». El trabajo de Allen Newell, John Laird, y Paul Rosenbloom en SOAR (Newell, 1990; Laird *et al.*, 1987) es el ejemplo mejor conocido de una arquitectura de agente completa. El llamado «movimiento situado» intenta entender la forma de actuar de los agentes inmersos en entornos reales, que disponen de sensores de entradas continuas. Uno de los medios más importantes para los agentes inteligentes es Internet. Los sistemas de IA han llegado a ser tan comunes en aplicaciones desarrolladas para la Web que el sufijo «-bot» se ha introducido en el lenguaje común. Más aún, tecnologías de IA son la base de muchas herramientas para Internet, como por ejemplo motores de búsqueda, sistemas de recomendación, y los sistemas para la construcción de portales Web.

Además de la primera edición de este libro de texto (Russell y Norvig, 1995), otros libros de texto han adoptado recientemente la perspectiva de agentes (Poole *et al.*, 1998; Nilsson, 1998). Una de las conclusiones que se han extraído al tratar de construir agentes completos ha sido que se deberían reorganizar los subcampos aislados de la IA para

que sus resultados se puedan interrelacionar. En particular, ahora se cree mayoritariamente que los sistemas sensoriales (visión, sónar, reconocimiento del habla, etc.) no pueden generar información totalmente fidedigna del medio en el que habitan. Otra segunda consecuencia importante, desde la perspectiva del agente, es que la IA se ha ido acercando a otros campos, como la teoría de control y la economía, que también tratan con agentes.

1.4 El estado del arte

¿Qué es capaz de hacer la IA hoy en día? Responder de manera concisa es difícil porque hay muchas actividades divididas en muchos subcampos. Aquí se presentan unas cuantas aplicaciones; otras aparecerán a lo largo del texto.

Planificación autónoma: a un centenar de millones de millas de la Tierra, el programa de la NASA Agente Remoto se convirtió en el primer programa de planificación autónoma a bordo que controlaba la planificación de las operaciones de una nave espacial desde abordo (Jonsson *et al.*, 2000). El Agente Remoto generaba planes a partir de objetivos generales especificados desde tierra, y monitorizaba las operaciones de la nave espacial según se ejecutaban los planes (detección, diagnóstico y recuperación de problemas según ocurrían).

Juegos: Deep Blue de IBM fue el primer sistema que derrotó a un campeón mundial en una partida de ajedrez cuando superó a Garry Kasparov por un resultado de 3.5 a 2.5 en una partida de exhibición (Goodman y Keene, 1997). Kasparov dijo que había percibido un «nuevo tipo de inteligencia» al otro lado del tablero. La revista *Newsweek* describió la partida como «La partida final». El valor de las acciones de IBM se incrementó en 18 billones de dólares.

Control autónomo: el sistema de visión por computador ALVINN fue entrenado para dirigir un coche de forma que siguiese una línea. Se instaló en una furgoneta controlada por computador en el NAVLAB de UCM y se utilizó para dirigir al vehículo por Estados Unidos. Durante 2.850 millas controló la dirección del vehículo en el 98 por ciento del trayecto. Una persona lo sustituyó en el dos por ciento restante, principalmente en vías de salida. El NAVLAB posee videocámaras que transmiten imágenes de la carretera a ALVINN, que posteriormente calcula la mejor dirección a seguir, basándose en las experiencias acumuladas en los viajes de entrenamiento.

Diagnóstico: los programas de diagnóstico médico basados en el análisis probabilista han llegado a alcanzar niveles similares a los de médicos expertos en algunas áreas de la medicina. Heckerman (1991) describe un caso en el que un destacado experto en la patología de los nodos linfáticos se mofó del diagnóstico generado por un programa en un caso especialmente difícil. El creador del programa le sugirió que le preguntase al computador cómo había generado el diagnóstico. La máquina indicó los factores más importantes en los que había basado su decisión y explicó la ligera interacción existente entre varios de los síntomas en este caso. Eventualmente, el experto aceptó el diagnóstico del programa.

Planificación logística: durante la crisis del Golfo Pérsico de 1991, las fuerzas de Estados Unidos desarrollaron la herramienta *Dynamic Analysis and Replanning Tool*

(DART) (Cross y Walker, 1994), para automatizar la planificación y organización logística del transporte. Lo que incluía hasta 50.000 vehículos, carga y personal a la vez, teniendo en cuenta puntos de partida, destinos, rutas y la resolución de conflictos entre otros parámetros. Las técnicas de planificación de IA permitieron que se generara un plan en cuestión de horas que podría haber llevado semanas con otros métodos. La agencia DARPA (*Defense Advanced Research Project Agency*) afirmó que esta aplicación por sí sola había más que amortizado los 30 años de inversión de DARPA en IA.

Robótica: muchos cirujanos utilizan hoy en día asistentes robot en operaciones de microcirugía. HipNav (DiGioia *et al.*, 1996) es un sistema que utiliza técnicas de visión por computador para crear un modelo tridimensional de la anatomía interna del paciente y después utiliza un control robotizado para guiar el implante de prótesis de cadera.

Procesamiento de lenguaje y resolución de problemas: PROVER B (Littman *et al.*, 1999) es un programa informático que resuelve crucigramas mejor que la mayoría de los humanos, utilizando restricciones en programas de relleno de palabras, una gran base de datos de crucigramas, y varias fuentes de información como diccionarios y bases de datos *online*, que incluyen la lista de películas y los actores que intervienen en ellas, entre otras cosas. Por ejemplo, determina que la pista «Historia de Niza» se puede resolver con «ETAGE» ya que su base de datos incluye el par pista/solución «Historia en Francia/ETAGE» y porque reconoce que los patrones «Niza X» y «X en Francia» a menudo tienen la misma solución. El programa no sabe que Niza es una ciudad de Francia, pero es capaz de resolver el puzzle.

Estos son algunos de los ejemplos de sistemas de inteligencia artificial que existen hoy en día. No se trata de magia o ciencia ficción, son más bien ciencia, ingeniería y matemáticas, para los que este libro proporciona una introducción.

1.5 Resumen

En este capítulo se define la IA y se establecen los antecedentes culturales que han servido de base. Algunos de los aspectos más destacables son:

- Cada uno tiene una visión distinta de lo que es la IA. Es importante responder a las dos preguntas siguientes: ¿Está interesado en el razonamiento y el comportamiento? ¿Desea modelar seres humanos o trabajar a partir de un ideal estándar?
- En este libro se adopta el criterio de que la inteligencia tiene que ver principalmente con las **acciones racionales**. Desde un punto de vista ideal, un **agente intelectual** es aquel que emprende la mejor acción posible ante una situación dada. Se estudiará el problema de la construcción de agentes que sean inteligentes en este sentido.
- Los filósofos (desde el año 400 a.C.) facilitaron el poder imaginar la IA, al concebir la idea de que la mente es de alguna manera como una máquina que funciona a partir del conocimiento codificado en un lenguaje interno, y al considerar que el pensamiento servía para seleccionar la acción a llevar a cabo.
- Las matemáticas proporcionaron las herramientas para manipular tanto las aseveraciones de certeza lógicas, como las inciertas de tipo probabilista. Asimismo,

prepararon el terreno para un entendimiento de lo que es el cálculo y el razonamiento con algoritmos.

- Los economistas formalizaron el problema de la toma de decisiones para maximizar los resultados esperados.
- Los psicólogos adoptaron la idea de que los humanos y los animales podían considerarse máquinas de procesamiento de información. Los lingüistas demostraron que el uso del lenguaje se ajusta a ese modelo.
- Los informáticos proporcionaron los artefactos que hicieron posible la aplicación de la IA. Los programas de IA tienden a ser extensos y no podrían funcionar sin los grandes avances en velocidad y memoria aportados por la industria informática.
- La teoría de control se centra en el diseño de dispositivos que actúan de forma óptima con base en la retroalimentación que reciben del entorno en el que están inmersos. Inicialmente, las herramientas matemáticas de la teoría de control eran bastante diferentes a las técnicas que utilizaba la IA, pero ambos campos se están acercando.
- La historia de la IA ha pasado por ciclos de éxito, injustificado optimismo y consecuente desaparición de entusiasmo y apoyos financieros. También ha habido ciclos caracterizados por la introducción de enfoques nuevos y creativos y de un perfeccionamiento sistemático de los mejores.
- La IA ha avanzado más rápidamente en la década pasada debido al mayor uso del método científico en la experimentación y comparación de propuestas.
- Los avances recientes logrados en el entendimiento de las bases teóricas de la inteligencia han ido aparejados con las mejoras realizadas en la optimización de los sistemas reales. Los subcampos de la IA se han integrado más y la IA ha encontrado elementos comunes con otras disciplinas.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

El estatus metodológico de la inteligencia artificial se ha investigado en *The Sciences of the Artificial*, escrito por Herb Simon (1981), en el cual se analizan áreas de investigación interesadas en el desarrollo de artefactos complejos. Explica cómo la IA se puede ver como ciencia y matemática. Cohen (1995) proporciona una visión de la metodología experimental en el marco de la IA. Ford y Hayes (1995) presentan una revisión crítica de la utilidad de la Prueba de Turing.

Artificial Intelligence: The Very Idea, de John Haugeland (1985), muestra una versión amena de los problemas prácticos y filosóficos de la IA. La ciencia cognitiva está bien descrita en varios textos recientes (Johnson-Laird, 1988; Stillings *et al.*, 1995; Thagard, 1996) y en la *Encyclopedia of the Cognitive Sciences* (Wilson y Keil, 1999). Baker (1989) cubre la parte sintáctica de la lingüística moderna, y Chierchia y McConnell-Ginet (1990) la semántica. Jurafsky y Martin (2000) revisan la lingüística computacional.

Los primeros trabajos en el campo de la IA se citan en *Computers and Thought* (1963), de Feigenbaum y Feldman, en *Semantic Information Processing* de Minsky, y en la serie *Machine Intelligence*, editada por Donald Michie. Webber y Nilsson (1981)

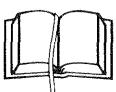
y Luger (1995) han recogido una nutrida cantidad de artículos influyentes. Los primeros artículos sobre redes neuronales están reunidos en *Neurocomputing* (Anderson y Rosenfeld, 1988). La *Encyclopedia of AI* (Shapiro, 1992) contiene artículos de investigación sobre prácticamente todos los temas de IA. Estos artículos son muy útiles para iniciarse en las diversas áreas presentes en la literatura científica.

El trabajo más reciente se encuentra en las actas de las mayores conferencias de IA: la *International Joint Conference on AI* (IJCAI), de carácter bianual, la *European Conference on AI* (ECAI), de carácter bianual, y la *National Conference on AI*, conocida normalmente como AAAI por la organización que la patrocina. Las revistas científicas que presentan aspectos generales de la IA más importantes son *Artificial Intelligence*, *Computational Intelligence*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *IEEE Intelligent Systems*, y la revista electrónica *Journal of Artificial Intelligence Research*. Hay también numerosas revistas y conferencias especializadas en áreas concretas, que se mencionarán en los capítulos apropiados. La asociaciones profesionales de IA más importantes son la American Association for Artificial Intelligence (AAAI), la ACM Special Interest Group in Artificial Intelligence (SIGART), y la Society for Artificial Intelligence and Simulation of Behaviour (AISB). La revista *AI Magazine* de AAAI contiene muchos artículos de interés general y manuales, y su página Web, aaai.org contiene noticias e información de referencia.

EJERCICIOS

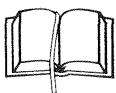


El propósito de los siguientes ejercicios es estimular la discusión, y algunos de ellos se podrían utilizar como proyectos. Alternativamente, se podría hacer un esfuerzo inicial para solucionarlos ahora, de forma que una vez se haya leído todo el libro se puedan revisar estos primeros intentos.



1.1 Defina con sus propias palabras: (a) inteligencia, (b) inteligencia artificial, (c) agente.

1.2 Lea el artículo original de Turing sobre IA (Turing, 1950). En él se comentan algunas objeciones potenciales a su propuesta y a su prueba de inteligencia. ¿Cuáles de estas objeciones tiene todavía validez? ¿Son válidas sus refutaciones? ¿Se le ocurren nuevas objeciones a esta propuesta teniendo en cuenta los desarrollos realizados desde que se escribió el artículo? En el artículo, Turing predijo que para el año 2000 sería probable que un computador tuviera un 30 por ciento de posibilidades de superar una Prueba de Turing dirigida por un evaluador inexperto con una duración de cinco minutos. ¿Considera razonable lo anterior en el mundo actual? ¿Y en los próximos 50 años?

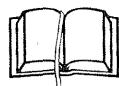


1.3 Todos los años se otorga el premio Loebner al programa que lo hace mejor en una Prueba de Turing concreta. Investigue y haga un informe sobre el último ganador del premio Loebner. ¿Qué técnica utiliza? ¿Cómo ha hecho que progrese la investigación en el campo de la IA?

1.4 Hay clases de problemas bien conocidos que son intratables para los computadores, y otras clases sobre los cuales un computador no pueda tomar una decisión. ¿Quiere esto decir que es imposible lograr la IA?

1.5 Supóngase que se extiende ANALOGY, el programa de Evans, como para alcanzar una puntuación de 200 en una prueba normal de cociente de inteligencia. ¿Quiere decir lo anterior que se ha creado un programa más inteligente que un ser humano? Explíquese.

1.6 ¿Cómo puede la introspección (revisión de los pensamientos íntimos) ser inexacta? ¿Se puede estar equivocado sobre lo que se cree? Discútase.



1.7 Consulte en la literatura existente sobre la IA si alguna de las siguientes tareas se puede efectuar con computadores:

- a)** Jugar una partida de tenis de mesa (ping-pong) decentemente.
- b)** Conducir un coche en el centro del Cairo.
- c)** Comprar comestibles para una semana en el mercado.
- d)** Comprar comestibles para una semana en la web.
- e)** Jugar una partida de *bridge* decentemente a nivel de competición.
- f)** Descubrir y demostrar nuevos teoremas matemáticos.
- g)** Escribir intencionadamente una historia divertida.
- h)** Ofrecer asesoría legal competente en un área determinada.
- i)** Traducir inglés hablado al sueco hablado en tiempo real.
- j)** Realizar una operación de cirugía compleja.

En el caso de las tareas que no sean factibles de realizar en la actualidad, trate de describir cuáles son las dificultades y calcule para cuándo se podrán superar.

1.8 Algunos autores afirman que la percepción y las habilidades motoras son la parte más importante de la inteligencia y que las capacidades de «alto nivel» son más bien parásitas (simples añadidos a las capacidades básicas). Es un hecho que la mayor parte de la evolución y que la mayor parte del cerebro se han concentrado en la percepción y las habilidades motoras, en tanto la IA ha descubierto que tareas como juegos e inferencia lógica resultan más sencillas, en muchos sentidos, que percibir y actuar en el mundo real. ¿Consideraría usted que ha sido un error la concentración tradicional de la IA en las capacidades cognitivas de alto nivel?

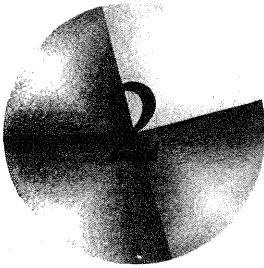
1.9 ¿Por qué la evolución tiende a generar sistemas que actúan racionalmente? ¿Qué objetivos deben intentar alcanzar estos sistemas?

1.10 ¿Son racionales las acciones reflejas (como retirar la mano de una estufa caliente)? ¿Son inteligentes?

1.11 «En realidad los computadores no son inteligentes, hacen solamente lo que le dicen los programadores». ¿Es cierta la última aseveración, e implica a la primera?

1.12 «En realidad los animales no son inteligentes, hacen solamente lo que le dicen sus genes». ¿Es cierta la última aseveración, e implica a la primera?

1.13 «En realidad los animales, los humanos y los computadores no pueden ser inteligentes, ellos sólo hacen lo que los átomos que los forman les dictan siguiendo las leyes de la física». ¿Es cierta la última aseveración, e implica a la primera?



Agentes inteligentes

Donde se discutirá la naturaleza de los agentes ideales, sus diversos hábitats y las formas de organizar los tipos de agentes existentes.

El Capítulo 1 identifica el concepto de **agente racional** como central en la perspectiva de la inteligencia artificial que presenta este libro. Esta noción se concreta más a lo largo de este capítulo. Se mostrará como el concepto de racionalidad se puede aplicar a una amplia variedad de agentes que operan en cualquier medio imaginable. En el libro, la idea es utilizar este concepto para desarrollar un pequeño conjunto de principios de diseño que sirvan para construir agentes útiles, sistemas que se puedan llamar razonablemente **inteligentes**.

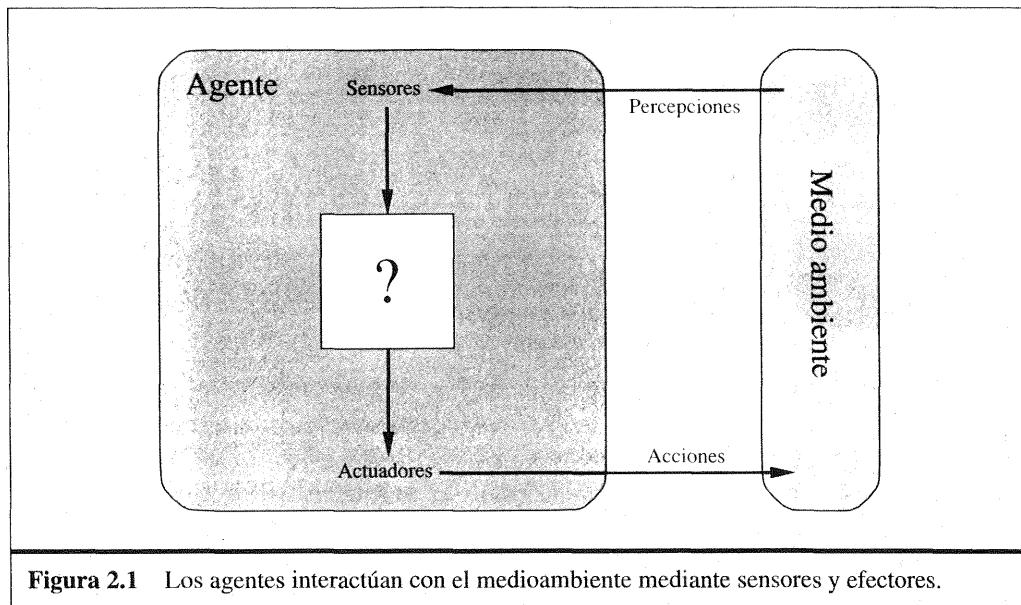
Se comienza examinando los agentes, los medios en los que se desenvuelven, y la interacción entre éstos. La observación de que algunos agentes se comportan mejor que otros nos lleva naturalmente a la idea de agente racional, aquel que se comporta tan bien como puede. La forma de actuar del agente depende de la naturaleza del medio; algunos hábitats son más complejos que otros. Se proporciona una categorización cruda del medio y se muestra cómo las propiedades de un hábitat influyen en el diseño de agentes adecuados para ese entorno. Se presenta un número de «esquemas» básicos para el diseño de agentes, a los que se dará cuerpo a lo largo del libro.

2.1 Agentes y su entorno

MEDIOAMBIENTE

Un **agente** es cualquier cosa capaz de percibir su **medioambiente** con la ayuda de **sensores** y actuar en ese medio utilizando **actuadores**¹. La Figura 2.1 ilustra esta idea sim-

¹ Se usa este término para indicar el elemento que reacciona a un estímulo realizando una acción. (N. del RT).



ple. Un agente humano tiene ojos, oídos y otros órganos sensoriales además de manos, piernas, boca y otras partes del cuerpo para actuar. Un agente robot recibe pulsaciones del teclado, archivos de información y paquetes vía red a modo de entradas sensoriales y actúa sobre el medio con mensajes en el monitor, escribiendo ficheros y enviando paquetes por la red. Se trabajará con la hipótesis general de que cada agente puede percibir sus propias acciones (pero no siempre sus efectos).

El término **percepción** se utiliza en este contexto para indicar que el agente puede recibir entradas en cualquier instante. La **secuencia de percepciones** de un agente refleja el historial completo de lo que el agente ha recibido. En general, *un agente tomará una decisión en un momento dado dependiendo de la secuencia completa de percepciones hasta ese instante*. Si se puede especificar qué decisión tomará un agente para cada una de las posibles secuencias de percepciones, entonces se habrá explicado más o menos todo lo que se puede decir de un agente. En términos matemáticos se puede decir que el comportamiento del agente viene dado por la **función del agente** que proyecta una percepción dada en una acción.

La función que describe el comportamiento de un agente se puede presentar en *forma de tabla*: en la mayoría de los casos esta tabla sería muy grande (infinita a menos que se limite el tamaño de la secuencia de percepciones que se quiera considerar). Dado un agente, con el que se quiera experimentar, se puede, en principio, construir esta tabla teniendo en cuenta todas las secuencias de percepción y determinando qué acción lleva a cabo el agente en respuesta². La tabla es, por supuesto, una caracterización *externa* del agente. *Inicialmente*, la función del agente para un agente artificial se imple-

² Si el agente selecciona la acción de manera aleatoria, entonces sería necesario probar cada secuencia muchas veces para identificar la probabilidad de cada acción. Se puede pensar que actuar de manera aleatoria es ridículo, pero como se verá posteriormente puede ser muy inteligente.

PROGRAMA DEL AGENTE

mentará mediante el **programa del agente**. Es importante diferenciar estas dos ideas. La función del agente es una descripción matemática abstracta; el programa del agente es una implementación completa, que se ejecuta sobre la arquitectura del agente.

Para ilustrar esta idea se utilizará un ejemplo muy simple, el mundo de la aspiradora presentado en la Figura 2.2. Este mundo es tan simple que se puede describir todo lo que en él sucede; es un mundo hecho a medida, para el que se pueden inventar otras variaciones. Este mundo en particular tiene solamente dos localizaciones: cuadrícula A y B. La aspiradora puede percibir en qué cuadrante se encuentra y si hay suciedad en él. Puede elegir si se mueve hacia la izquierda, derecha, aspirar la suciedad o no hacer nada. Una función muy simple para el agente vendría dada por: si la cuadrícula en la que se encuentra está sucia, entonces aspirar, de otra forma cambiar de cuadrícula. Una muestra parcial de la función del agente representada en forma de tabla aparece en la Figura 2.3. Un programa de agente simple para esta función de agente se mostrará posteriormente en la Figura 2.8.

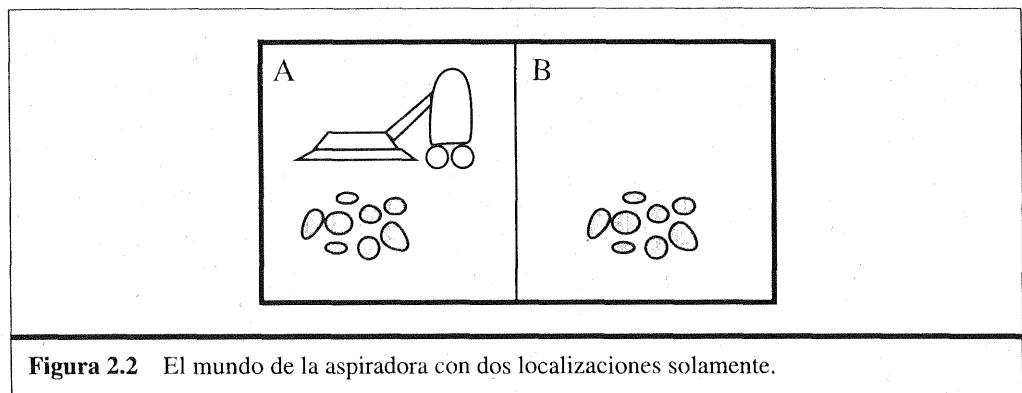


Figura 2.2 El mundo de la aspiradora con dos localizaciones solamente.

Secuencia de percepciones	Acción
[A, Limpio]	Derecha
[A, Sucio]	Aspirar
[B, Limpio]	Izquierda
[B, Sucio]	Aspirar
[A, Limpio], [A, Limpio]	Derecha
[A, Limpio], [A, Sucio]	Aspirar
—	—
—	—
—	—
[A, Limpio], [A, Limpio], [A, Limpio]	Derecha
[A, Limpio], [A, Limpio], [A, Sucio]	Aspirar
—	—
—	—
—	—

Figura 2.3 Tabla parcial de una función de agente sencilla para el mundo de la aspiradora que se muestra en la Figura 2.2.

Revisando la Figura 2.3, se aprecia que se pueden definir varios agentes para el mundo de la aspiradora simplemente rellenando la columna de la derecha de formas distintas. La pregunta obvia, entonces es: *¿cuál es la mejor forma de llenar una tabla?* En otras palabras, ¿qué hace que un agente sea bueno o malo, inteligente o estúpido? Estas preguntas se responden en la siguiente sección.

Antes de terminar esta sección, es necesario remarcar que la noción de agente es supuestamente una herramienta para el análisis de sistemas, y no una caracterización absoluta que divida el mundo entre agentes y no agentes. Se puede ver una calculadora de mano como un agente que elige la acción de mostrar «4» en la pantalla, dada la secuencia de percepciones « $2 + 2 =$ ». Pero este análisis difícilmente puede mejorar nuestro conocimiento acerca de las calculadoras.

2.2 Buen comportamiento: el concepto de racionalidad

AGENTE RACIONAL

Un **agente racional** es aquel que hace lo correcto; en términos conceptuales, cada elemento de la tabla que define la función del agente se tendría que llenar correctamente. Obviamente, hacer lo correcto es mejor que hacer algo incorrecto, pero ¿qué significa hacer lo correcto? Como primera aproximación, se puede decir que lo correcto es aquello que permite al agente obtener un resultado mejor. Por tanto, se necesita determinar una forma de medir el éxito. Ello, junto a la descripción del entorno y de los sensores y actuadores del agente, proporcionará una especificación completa de la tarea que desempeña el agente. Dicho esto, ahora es posible definir de forma más precisa qué significa la racionalidad.

MEDIDAS DE RENDIMIENTO

Medidas de rendimiento

Las **medidas de rendimiento** incluyen los criterios que determinan el éxito en el comportamiento del agente. Cuando se sitúa un agente en un medio, éste genera una secuencia de acciones de acuerdo con las percepciones que recibe. Esta secuencia de acciones hace que su hábitat pase por una secuencia de estados. Si la secuencia es la deseada, entonces el agente habrá actuado correctamente. Obviamente, no hay una única medida adecuada para todos los agentes. Se puede preguntar al agente por su opinión subjetiva acerca de su propia actuación, pero muchos agentes serían incapaces de contestar, y otros podrían engañarse a sí mismos³. Por tanto hay que insistir en la importancia de utilizar medidas de rendimiento objetivas, que normalmente determinará el diseñador encargado de la construcción del agente.

Si retomamos el ejemplo de la aspiradora de la sección anterior, se puede proponer utilizar como medida de rendimiento la cantidad de suciedad limpiada en un período de

³ Los agentes humanos son conocidos en particular por su «acidez», hacen creer que no quieren algo después de no haberlo podido conseguir, por ejemplo, «Ah bueno, de todas formas no quería ese estúpido Premio Nobel».



ocho horas. Con agentes racionales, por supuesto, se obtiene lo que se demanda. Un agente racional puede maximizar su medida de rendimiento limpiando la suciedad, tirando la basura al suelo, limpiándola de nuevo, y así sucesivamente. Una medida de rendimiento más adecuada recompensaría al agente por tener el suelo limpio. Por ejemplo, podría ganar un punto por cada cuadrícula limpia en cada período de tiempo (quizás habría que incluir algún tipo de penalización por la electricidad gastada y el ruido generado). *Como regla general, es mejor diseñar medidas de utilidad de acuerdo con lo que se quiere para el entorno, más que de acuerdo con cómo se cree que el agente debe comportarse.*

La selección de la medida de rendimiento no es siempre fácil. Por ejemplo, la noción de «suelo limpio» del párrafo anterior está basada en un nivel de limpieza medio a lo largo del tiempo. Además, este nivel medio de limpieza se puede alcanzar de dos formas diferentes, llevando a cabo una limpieza mediocre pero continua o limpiando en profundidad, pero realizando largos descansos. La forma más adecuada de hacerlo puede venir dada por la opinión de un encargado de la limpieza profesional, pero en realidad es una cuestión filosófica profunda con fuertes implicaciones. ¿Qué es mejor, una vida temeraria con altos y bajos, o una existencia segura pero aburrida? ¿Qué es mejor, una economía en la que todo el mundo vive en un estado de moderada pobreza o una en la que algunos viven en la abundancia y otros son muy pobres? Estas cuestiones se dejan como ejercicio para los lectores diligentes.

Racionalidad

La rationalidad en un momento determinado depende de cuatro factores:

- La medida de rendimiento que define el criterio de éxito.
- El conocimiento del medio en el que habita acumulado por el agente.
- Las acciones que el agente puede llevar a cabo.
- La secuencia de percepciones del agente hasta este momento.

DEFINICIÓN DE AGENTE RACIONAL



Esto nos lleva a la **definición de agente racional**:

En cada posible secuencia de percepciones, un agente racional deberá emprender aquella acción que supuestamente maximice su medida de rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones y en el conocimiento que el agente mantiene almacenado.

Considerando que el agente aspiradora limpia una cuadrícula si está sucia y se mueve a la otra si no lo está (ésta es la función del agente que aparece en la tabla de la Figura 2.3), ¿se puede considerar racional? ¡Depende! Primero, se debe determinar cuál es la medida de rendimiento, qué se conoce del entorno, y qué sensores y actuadores tiene el agente. Si asumimos que:

- La medida de rendimiento premia con un punto al agente por cada cuadro limpio en un período de tiempo concreto, a lo largo de una «vida» de 1.000 períodos.
- La «geografía» del medio se conoce *a priori* (Figura 2.2), pero que la distribución de la suciedad y la localización inicial del agente no se conocen. Las cuadrículas se mantienen limpias y aspirando se limpia la cuadrícula en que se encuentre el agente. Las acciones *Izquierda* y *Derecha* mueven al agente hacia la izquierda y

derecha excepto en el caso de que ello pueda llevar al agente fuera del recinto, en este caso el agente permanece donde se encuentra.

- Las únicas acciones permitidas son *Izquierda*, *Derecha*, *Aspirar* y *NoOp* (no hacer nada).
- El agente percibe correctamente su localización y si esta localización contiene suciedad.

Puede afirmarse que *bajo estas circunstancias* el agente es verdaderamente racional; el rendimiento que se espera de este agente es por lo menos tan alto como el de cualquier otro agente. El Ejercicio 2.4 pide que se pruebe este hecho.

Fácilmente se puede observar que el agente puede resultar irracional en circunstancias diferentes. Por ejemplo, cuando toda la suciedad se haya eliminado el agente oscilará innecesariamente hacia delante y atrás; si la medida de rendimiento incluye una penalización de un punto por cada movimiento hacia la derecha e izquierda, la respuesta del agente será pobre. Un agente más eficiente no hará nada si está seguro de que todas las cuadrículas están limpias. Si una cuadrícula se ensucia de nuevo, el agente debe identificarlo en una de sus revisiones ocasionales y limpiarla. Si no se conoce la geografía del entorno, el agente tendrá que explorarla y no quedarse parado en las cuadrículas A y B. El Ejercicio 2.4 pide que se diseñen agentes para estos casos.

Omnisciencia, aprendizaje y autonomía

OMNISCIENCIA

Es necesario tener cuidado al distinguir entre racionalidad y **omnisciencia**. Un agente omnisciente conoce el resultado de su acción y actúa de acuerdo con él; sin embargo, en realidad la omnisciencia no es posible. Considerando el siguiente ejemplo: estoy paseando por los Campos Elíseos y veo un amigo al otro lado de la calle. No hay tráfico alrededor y no tengo ningún compromiso, entonces, actuando racionalmente, comenzaría a cruzar la calle. Al mismo tiempo, a 33.000 pies de altura, se desprende la puerta de un avión⁴, y antes de que termine de cruzar al otro lado de la calle me encuentro aplastado. ¿Fue irracional cruzar la calle? Sería de extrañar que en mi nota necrológica apareciera «Un idiota intentando cruzar la calle».

Este ejemplo muestra que la racionalidad no es lo mismo que la perfección. La racionalidad maximiza el rendimiento esperado, mientras la perfección maximiza el resultado real. Alejarse de la necesidad de la perfección no es sólo cuestión de hacer justicia con los agentes. El asunto es que resulta imposible diseñar un agente que siempre lleve a cabo, de forma sucesiva, las mejores acciones después de un acontecimiento, a menos que se haya mejorado el rendimiento de las bolas de cristal o las máquinas de tiempo.

La definición propuesta de racionalidad no requiere omnisciencia, ya que la elección racional depende sólo de la secuencia de percepción hasta *la fecha*. Es necesario asegurarse de no haber permitido, por descuido, que el agente se dedique decididamente a llevar a cabo acciones poco inteligentes. Por ejemplo, si el agente no mirase a ambos lados de la calle antes de cruzar una calle muy concurrida, entonces su secuencia de per-

⁴ Véase N. Henderson, «New door latches urged for Boeing 747 jumbo jets» (es urgente dotar de nuevas cerraduras a las puertas de los Boeing jumbo 747), *Washington Post*, 24 de agosto de 1989.

RECOPILACIÓN DE INFORMACIÓN

EXPLORACIÓN

APRENDIZAJE

AUTONOMÍA

cepción no le indicaría que se está acercando un gran camión a gran velocidad. ¿La definición de racionalidad nos está indicando que está bien cruzar la calle? ¡Todo lo contrario! Primero, no sería racional cruzar la calle sólo teniendo esta secuencia de percepciones incompleta: el riesgo de accidente al cruzarla sin mirar es demasiado grande. Segundo, un agente racional debe elegir la acción de «mirar» antes de intentar cruzar la calle, ya que el mirar maximiza el rendimiento esperado. Llevar a cabo acciones *con la intención de modificar percepciones futuras*, en ocasiones proceso denominado **recopilación de información**, es una parte importante de la racionalidad y se comenta en profundidad en el Capítulo 16. Un segundo ejemplo de recopilación de información lo proporciona la **exploración** que debe llevar a cabo el agente aspiradora en un medio inicialmente desconocido.

La definición propuesta implica que el agente racional no sólo recopile información, sino que **aprenda** lo máximo posible de lo que está percibiendo. La configuración inicial del agente puede reflejar un conocimiento preliminar del entorno, pero a medida que el agente adquiere experiencia éste puede modificarse y aumentar. Hay casos excepcionales en los que se conoce totalmente el entorno *a priori*. En estos casos, el agente no necesita percibir y aprender; simplemente actúa de forma correcta. Por supuesto, estos agentes son muy frágiles. Considérese el caso del humilde escarabajo estercolero. Después de cavar su nido y depositar en él su huevos, tomó una bola de estiércol de una pila cercana para tapar su entrada. Si *durante el trayecto* se le quita la bola, el escarabajo continuará su recorrido y hará como si estuviera tapando la entrada del nido, sin tener la bola y sin darse cuenta de ello. La evolución incorporó una suposición en la conducta del escarabajo, y cuando se viola, el resultado es un comportamiento insatisfactorio. La avispa cavadora es un poco más inteligente. La avispa hembra cavará una madriguera, saldrá de ella, picará a una oruga y la llevará a su madriguera, se introducirá en la madriguera para comprobar que todo está bien, arrastrará la oruga hasta el fondo y pondrá sus huevos. La oruga servirá como fuente de alimento cuando los huevos se abran. Hasta ahora todo bien, pero si un entomólogo desplaza la oruga unos centímetros fuera cuando la avispa está revisando la situación, ésta volverá a la etapa de «arrastre» que figura en su plan, y continuará con el resto del plan sin modificación alguna, incluso después de que se intervenga para desplazar la oruga. La avispa cavadora no es capaz de aprender que su plan innato está fallando, y por tanto no lo cambiará.

Los agentes con éxito dividen las tareas de calcular la función del agente en tres períodos diferentes: cuando se está diseñando el agente, y están los diseñadores encargados de realizar algunos de estos cálculos; cuando está pensando en la siguiente operación, el agente realiza más cálculos; y cuando está aprendiendo de la experiencia, el agente lleva a cabo más cálculos para decidir cómo modificar su forma de comportarse.

Se dice que un agente carece de **autonomía** cuando se apoya más en el conocimiento inicial que le proporciona su diseñador que en sus propias percepciones. Un agente racional debe ser autónomo, debe saber aprender a determinar cómo tiene que compensar el conocimiento incompleto o parcial inicial. Por ejemplo, el agente aspiradora que aprenda a prever dónde y cuándo aparecerá suciedad adicional lo hará mejor que otro que no aprenda. En la práctica, pocas veces se necesita autonomía completa desde el comienzo: cuando el agente haya tenido poca o ninguna experiencia, tendrá que actuar de forma aleatoria a menos que el diseñador le haya proporcionado ayuda. Así, de la

misma forma que la evolución proporciona a los animales sólo los reactivos necesarios para que puedan sobrevivir lo suficiente para aprender por ellos mismos, sería razonable proporcionar a los agentes que disponen de inteligencia artificial un conocimiento inicial, así como de la capacidad de aprendizaje. Después de las suficientes experiencias interaccionando con el entorno, el comportamiento del agente racional será efectivamente *independiente* del conocimiento que poseía inicialmente. De ahí, que la incorporación del aprendizaje facilite el diseño de agentes racionales individuales que tendrán éxito en una gran cantidad de medios.

2.3 La naturaleza del entorno

ENTORNOS DE TRABAJO

REAS

Ahora que se tiene una definición de racionalidad, se está casi preparado para pensar en la construcción de agentes racionales. Primero, sin embargo, hay que centrarse en los **entornos de trabajo**, que son esencialmente los «problemas» para los que los agentes racionales son las «soluciones». Para ello se comienza mostrando cómo especificar un entorno de trabajo, ilustrando el proceso con varios ejemplos. Posteriormente se mostrará que el entorno de trabajo ofrece diferentes posibilidades, de forma que cada una de las posibilidades influyen directamente en el diseño del programa del agente.

Especificación del entorno de trabajo

En la discusión de la racionalidad de un agente aspiradora simple, hubo que especificar las medidas de rendimiento, el entorno, y los actuadores y sensores del agente. Todo ello forma lo que se llama el **entorno de trabajo**, para cuya denominación se utiliza el acrónimo **REAS** (Rendimiento, Entorno, Actuadores, Sensores). En el diseño de un agente, el primer paso debe ser siempre especificar el entorno de trabajo de la forma más completa posible.

El mundo de la aspiradora fue un ejemplo simple; considérese ahora un problema más complejo: un taxista automático. Este ejemplo se utilizará a lo largo del capítulo. Antes de alarma al lector, conviene aclarar que en la actualidad la construcción de un taxi automatizado está fuera del alcance de la tecnología actual. Véase en la página 31 la descripción de un robot conductor que ya existe en la actualidad, o lea las actas de la conferencia *Intelligent Transportation Systems*. La tarea de conducir un automóvil, en su totalidad, es extremadamente *ilimitada*. No hay límite en cuanto al número de nuevas combinaciones de circunstancias que pueden surgir (por esta razón se eligió esta actividad en la presente discusión). La Figura 2.4 resume la descripción REAS para el entorno de trabajo del taxi. El próximo párrafo explica cada uno de sus elementos en más detalle.

Primero, ¿cuál es el **entorno de trabajo** en el que el taxista automático aspira a conducir? Dentro de las cualidades deseables que debería tener se incluyen el que llegue al destino correcto; que minimice el consumo de combustible; que minimice el tiempo de viaje y/o coste; que minimice el número de infracciones de tráfico y de molestias a otros conductores; que maximice la seguridad, la comodidad del pasajero y el

Tipo de agente	Medidas de rendimiento	Entorno	Actuadores	Sensores
Taxista	Seguro, rápido, legal, viaje confortable, maximización del beneficio	Carreteras, otro tráfico, peatones, clientes	Dirección, acelerador, freno, señal, bocina, visualizador	Cámaras, sónar, velocímetro, GPS, tacómetro, visualizador de la aceleración, sensores del motor, teclado

Figura 2.4 Descripción REAS del entorno de trabajo de un taxista automático.

beneficio. Obviamente, alguno de estos objetivos entra en conflicto por lo que habrá que llegar a acuerdos.

Siguiente, ¿cuál es el **entorno** en el que se encontrará el taxi? Cualquier taxista debe estar preparado para circular por distintas carreteras, desde caminos rurales y calles urbanas hasta autopistas de 12 carriles. En las carreteras se pueden encontrar con tráfico, peatones, animales, obras, coches de policía, charcos y baches. El taxista también tiene que comunicarse tanto con pasajeros reales como potenciales. Hay también elecciones opcionales. El taxi puede operar en California del Sur, donde la nieve es raramente un problema, o en Alaska, donde raramente no lo es. Puede conducir siempre por la derecha, o puede ser lo suficientemente flexible como para que circule por la izquierda cuando se encuentre en el Reino Unido o en Japón. Obviamente, cuanto más restringido esté el entorno, más fácil será el problema del diseño.

Los **actuadores** disponibles en un taxi automático serán más o menos los mismos que los que tiene a su alcance un conductor humano: el control del motor a través del acelerador y control sobre la dirección y los frenos. Además, necesitará tener una pantalla de visualización o un sintetizador de voz para responder a los pasajeros, y quizás algún mecanismo para comunicarse, educadamente o de otra forma, con otros vehículos.

Para alcanzar sus objetivos en el entorno en el que circula, el taxi necesita saber dónde está, qué otros elementos están en la carretera, y a qué velocidad circula. Sus **sensores** básicos deben, por tanto, incluir una o más cámaras de televisión dirigidas, un velocímetro y un tacómetro. Para controlar el vehículo adecuadamente, especialmente en las curvas, debe tener un acelerador; debe conocer el estado mecánico del vehículo, de forma que necesitará sensores que controlen el motor y el sistema eléctrico. Debe tener instrumentos que no están disponibles para un conductor medio: un sistema de posicionamiento global vía satélite (GPS) para proporcionarle información exacta sobre su posición con respecto a un mapa electrónico, y sensores infrarrojos o sonares para detectar las distancias con respecto a otros coches y obstáculos. Finalmente, necesitará un teclado o micrófono para que el pasajero le indique su destino.

La Figura 2.5 muestra un esquema con los elementos REAS básicos para diferentes clases de agentes adicionales. Más ejemplos aparecerán en el Ejercicio 2.5. Puede sorprender a algunos lectores que se incluya en la lista de tipos de agente algunos programas que operan en la totalidad del entorno artificial definido por las entradas del teclado y los caracteres impresos en el monitor. «Seguramente», nos podemos preguntar,

Tipo de agente	Medidas de rendimiento	Entorno	Actuadores	Sensores
Sistema de diagnóstico médico	Pacientes sanos, reducir costes, demandas	Pacientes, hospital, personal	Visualizar preguntas, pruebas, diagnósticos, tratamientos, casos	Teclado para la entrada de síntomas, conclusiones, respuestas de pacientes
Sistema de análisis de imágenes de satélites	Categorización de imagen correcta	Conexión con el satélite en órbita	Visualizar la categorización de una escena	Matriz de pixels de colores
Robot para la selección de componentes	Porcentaje de componentes clasificados en los cubos correctos	Cinta transportadora con componentes, cubos	Brazo y mano articulados	Cámara, sensor angular
Controlador de una refinería	Maximizar la pureza, producción y seguridad	Refinería, operadores	Válvulas, bombas, calentadores, monitores	Temperatura, presión, sensores químicos
Tutor de inglés interactivo	Maximizar la puntuación de los estudiantes en los exámenes	Conjunto de estudiantes, agencia examinadora	Visualizar los ejercicios, sugerencias, correcciones	Teclado de entrada

Figura 2.5 Ejemplos de tipos de agentes y sus descripciones REAS.

tar, «¿este no es un entorno real, verdad?». De hecho, lo que importa no es la distinción entre un medio «real» y «artificial», sino la complejidad de la relación entre el comportamiento del agente, la secuencia de percepción generada por el medio y la medida de rendimiento. Algunos entornos «reales» son de hecho bastante simples. Por ejemplo, un robot diseñado para inspeccionar componentes según pasan por una cinta transportadora puede hacer uso de varias suposiciones simples: que la cinta siempre estará iluminada, que conocerá todos los componentes que circulen por la cinta, y que hay solamente dos acciones (aceptar y rechazar).

En contraste, existen algunos **agentes software** (o robots *software* o **softbots**) en entornos ricos y prácticamente ilimitados. Imagine un softbot diseñado para pilotar el simulador de vuelo de un gran avión comercial. El simulador constituye un medio muy detallado y complejo que incluye a otros aviones y operaciones de tierra, y el agente *software* debe elegir, en tiempo real, una de entre un amplio abanico de posibilidades. O imagine un robot diseñado para que revise fuentes de información en Internet y para que muestre aquellas que sean interesantes a sus clientes. Para lograrlo, deberá poseer cierta habilidad en el procesamiento de lenguaje natural, tendrá que aprender qué es lo que le interesa a cada cliente, y tendrá que ser capaz de cambiar sus planes dinámicamente.

mente, por ejemplo, cuando se interrumpa la conexión con una fuente de información o cuando aparezca una nueva. Internet es un medio cuya complejidad rivaliza con la del mundo físico y entre cuyos habitantes se pueden incluir muchos agentes artificiales.

Propiedades de los entornos de trabajo

El rango de los entornos de trabajo en los que se utilizan técnicas de IA es obviamente muy grande. Sin embargo, se puede identificar un pequeño número de dimensiones en las que categorizar estos entornos. Estas dimensiones determinan, hasta cierto punto, el diseño más adecuado para el agente y la utilización de cada una de las familias principales de técnicas en la implementación del agente. Primero se enumeran la dimensiones, y después se analizan varios entornos de trabajo para ilustrar estas ideas. Las definiciones dadas son informales; capítulos posteriores proporcionan definiciones más precisas y ejemplos de cada tipo de entorno.

TOTALMENTE
OBSEVABLE

- **Totalmente observable vs. parcialmente observable.**

Si los sensores del agente le proporcionan acceso al estado completo del medio en cada momento, entonces se dice que el entorno de trabajo es totalmente observable⁵. Un entorno de trabajo es, efectivamente, totalmente observable si los sensores detectan todos los aspectos que son relevantes en la toma de decisiones; la relevancia, en cada momento, depende de las medidas de rendimiento. Entornos totalmente observables son convenientes ya que el agente no necesita mantener ningún estado interno para saber qué sucede en el mundo. Un entorno puede ser parcialmente observable debido al ruido y a la existencia de sensores poco exactos o porque los sensores no reciben información de parte del sistema, por ejemplo, un agente aspiradora con sólo un sensor de suciedad local no puede saber si hay suciedad en la otra cuadrícula, y un taxi automatizado no pude saber qué están pensando otros conductores.

DETERMINISTA

ESTOCÁSTICO

- **Determinista vs. estocástico.**

Si el siguiente estado del medio está totalmente determinado por el estado actual y la acción ejecutada por el agente, entonces se dice que el entorno es determinista; de otra forma es estocástico. En principio, un agente no se tiene que preocupar de la incertidumbre en un medio totalmente observable y determinista. Sin embargo, si el medio es parcialmente observable entonces puede *parecer* estocástico. Esto es particularmente cierto si se trata de un medio complejo, haciendo difícil el mantener constancia de todos los aspectos observados. Así, a menudo es mejor pensar en entornos deterministas o estocásticos *desde el punto de vista del agente*. El agente taxi es claramente estocástico en este sentido, ya que no se puede predecir el comportamiento del tráfico exactamente; más aún, una rueda se puede reventar y un motor se puede gripar sin previo aviso. El mundo de la aspiradora es deter-

⁵ La primera edición de este libro utiliza los términos **accesible** e **inaccesible** en vez de **total** y **parcialmente observable**; **no determinista** en vez de **estocástico**; y **no episódico** en vez de **secuencial**. La nueva terminología es más consistente con el uso establecido.

minista, como ya se describió, pero las variaciones pueden incluir elementos estocásticos como la aparición de suiedad aleatoria y un mecanismo de succión ineficiente (Ejercicio 2.12). Si el medio es determinista, excepto para las acciones de otros agentes, decimos que el medio es **estratégico**.

ESTRÁTÉGICO

EPISÓDICO

SECUENCIAL

ESTÁTICO

DINÁMICO

SEMIDINÁMICO

DISCRETO

CONTINUO

- **Episódico vs. secuencial⁶.**

En un entorno de trabajo episódico, la experiencia del agente se divide en episodios atómicos. Cada episodio consiste en la percepción del agente y la realización de una única acción posterior. Es muy importante tener en cuenta que el siguiente episodio no depende de las acciones que se realizaron en episodios previos. En los medios episódicos la elección de la acción en cada episodio depende sólo del episodio en sí mismo. Muchas tareas de clasificación son episódicas. Por ejemplo, un agente que tenga que seleccionar partes defectuosas en una cadena de montaje basa sus decisiones en la parte que está evaluando en cada momento, sin tener en cuenta decisiones previas; más aún, a la decisión presente no le afecta el que la próxima fase sea defectuosa. En entornos secuenciales, por otro lado, la decisión presente puede afectar a decisiones futuras. El ajedrez y el taxista son secuenciales: en ambos casos, las acciones que se realizan a corto plazo pueden tener consecuencias a largo plazo. Los medios episódicos son más simples que los secuenciales porque la gente no necesita pensar con tiempo.

- **Estático vs. dinámico.**

Si el entorno puede cambiar cuando el agente está deliberando, entonces se dice que el entorno es dinámico para el agente; de otra forma se dice que es estático. Los medios estáticos son fáciles de tratar ya que el agente no necesita estar pendiente del mundo mientras está tomando una decisión sobre una acción, ni necesita preocuparse sobre el paso del tiempo. Los medios dinámicos, por el contrario, están preguntando continuamente al agente qué quiere hacer; si no se ha decidido aún, entonces se entiende que ha tomado la decisión de no hacer nada. Si el entorno no cambia con el paso del tiempo, pero el rendimiento del agente cambia, entonces se dice que el medio es **semidinámico**. El taxista es claramente dinámico: tanto los otros coches como el taxi se están moviendo mientras el algoritmo que guía la conducción indica qué es lo próximo a hacer. El ajedrez, cuando se juega con un reloj, es semideterminista. Los crucigramas son estáticos.

- **Discreto vs. continuo.**

La distinción entre discreto y continuo se puede aplicar al *estado* del medio, a la forma en la que se maneja el *tiempo* y a las *percepciones* y *acciones* del agente. Por ejemplo, un medio con estados discretos como el del juego del ajedrez tiene un número finito de estados distintos: El ajedrez tiene un conjunto discreto de percepciones y acciones. El taxista conduciendo define un estado continuo y un problema de tiempo continuo: la velocidad y la ubicación del taxi y de los otros vehículos pasan por un rango de valores continuos de forma suave a lo largo del

⁶ La palabra «secuencial» se utiliza también en el campo de la informática como antónimo de «paralelo». Los dos significados no están relacionados.

tiempo. La conducción del taxista es también continua (ángulo de dirección, etc.). Las imágenes captadas por cámaras digitales son discretas, en sentido estricto, pero se tratan típicamente como representaciones continuas de localizaciones e intensidades variables.

AGENTE INDIVIDUAL

MULTIAGENTE

COMPETITIVO

COOPERATIVO

• Agente individual vs. multiagente.

La distinción entre el entorno de un agente individual y el de un sistema multiagente puede parecer suficientemente simple. Por ejemplo, un agente resolviendo un crucigrama por sí mismo está claramente en un entorno de agente individual, mientras que un agente que juega al ajedrez está en un entorno con dos agentes. Sin embargo hay algunas diferencias sutiles. Primero, se ha descrito que una entidad *puede* percibirse como un agente, pero no se ha explicado qué entidades se *deben* considerar agentes. ¿Tiene el agente A (por ejemplo el agente taxista) que tratar un objeto B (otro vehículo) como un agente, o puede tratarse meramente como un objeto con un comportamiento estocástico, como las olas de la playa o las hojas que mueve el viento? La distinción clave está en identificar si el comportamiento de B está mejor descrito por la maximización de una medida de rendimiento cuyo valor depende del comportamiento de A. Por ejemplo, en el ajedrez, la entidad oponente B intenta maximizar su medida de rendimiento, la cual, según las reglas, minimiza la medida de rendimiento del agente A. Por tanto, el ajedrez es un entorno multiagente **competitivo**. Por otro lado, en el medio definido por el taxista circulando, el evitar colisiones maximiza la medida de rendimiento de todos los agentes, así pues es un entorno multiagente parcialmente **cooperativo**. Es también parcialmente competitivo ya que, por ejemplo, sólo un coche puede ocupar una plaza de aparcamiento. Los problemas en el diseño de agentes que aparecen en los entornos multiagente son a menudo bastante diferentes de los que aparecen en entornos con un único agente; por ejemplo, la **comunicación** a menudo emerge como un comportamiento racional en entornos multiagente; en algunos entornos competitivos parcialmente observables el **comportamiento estocástico** es racional ya que evita las dificultades de la predicción.

Como es de esperar, el caso más complejo es el *parcialmente observable, estocástico, secuencial, dinámico, continuo y multiagente*. De hecho, suele suceder que la mayoría de las situaciones reales son tan complejas que sería discutible clasificarlas como *realmente deterministas*. A efectos prácticos, se deben tratar como estocásticas. Un taxista circulando es un problema, complejo a todos los efectos.

La Figura 2.6 presenta las propiedades de un número de entornos familiares. Hay que tener en cuenta que las respuestas no están siempre preparadas de antemano. Por ejemplo, se ha presentado el ajedrez como totalmente observable; en sentido estricto, esto es falso porque ciertas reglas que afectan al movimiento de las torres, el enroque y a movimientos por repetición requieren que se recuerden algunos hechos sobre la historia del juego que no están reflejados en el estado del tablero. Estas excepciones, por supuesto, no tienen importancia si las comparamos con aquellas que aparecen en el caso del taxista, el tutor de inglés, o el sistema de diagnóstico médico.

Entornos de trabajo	Observable	Determinista	Episódico	Estático	Discreto	Agentes
Crucigrama Ajedrez con reloj	Totalmente Totalmente	Determinista Estratégico	Secuencial Secuencial	Estático Semi	Discreto Discreto	Individual Multi
Póker Backgammon	Parcialmente Totalmente	Estratégico Estocástico	Secuencial Secuencial	Estático Estático	Discreto Discreto	Multi Multi
Taxi circulando Diagnóstico médico	Parcialmente Parcialmente	Estocástico Estocástico	Secuencial Secuencial	Dinámico Dinámico	Continuo Continuo	Multi Individual
Análisis de imagen Robot clasificador	Totalmente Parcialmente	Determinista Estocástico	Episódico Episódico	Semi Dinámico	Continuo Continuo	Individual Individual
Controlador de refinería Tutor interactivo de inglés	Parcialmente Parcialmente	Estocástico Estocástico	Secuencial Secuencial	Dinámico Dinámico	Continuo Discreto	Individual Multi

Figura 2.6 Ejemplos de entornos de trabajo y sus características.

Otras entradas de la tabla dependen de cómo se haya definido el entorno de trabajo. Se ha definido el sistema de diagnóstico médico como un único agente porque no es rentable modelar el proceso de la enfermedad en un paciente como un agente; pero incluso el sistema de diagnóstico médico podría necesitar tener en cuenta a pacientes recalcitrantes y empleados escépticos, de forma que el entorno podría tener un aspecto multiagente. Más aún, el diagnóstico médico es episódico si se concibe como proporcionar un diagnóstico a partir de una lista de síntomas; el problema es secuencial si ello trae consigo la propuesta de una serie de pruebas, un proceso de evaluación a lo largo del tratamiento, y demás aspectos. Muchos entornos son, también, episódicos si se observan desde un nivel de abstracción más alto que el de las acciones individuales del agente. Por ejemplo, un torneo de ajedrez consiste en una secuencia de juegos; cada juego es un episodio, pero (a la larga) la contribución de los movimientos en una partida al resultado general que obtenga el agente no se ve afectada por los movimientos realizados en la partida anterior. Por otro lado, las decisiones tomadas en una partida concreta son ciertamente de tipo secuencial.

El repositorio de código asociado a este libro (aima.cs.berkeley.edu) incluye la implementación de un número de entornos, junto con un simulador de entornos de propósito general que sitúa uno o más agentes en un entorno simulado, observa su comportamiento a lo largo del tiempo, y los evalúa de acuerdo con una medida de rendimiento dada. Estos experimentos no sólo se han realizado para un medio concreto, sino que se han realizado con varios problemas obtenidos de una **clase de entornos**. Por ejemplo, para evaluar un taxista en un tráfico simulado, sería interesante hacer varias simulaciones con diferente tipo de tráfico, claridad y condiciones atmosféricas. Si se diseña un agente para un escenario concreto, se pueden sacar ventajas de las propiedades específicas de ese caso en particular, pero puede no identificarse un buen diseño para conducir en general. Por esta razón, el repositorio de código también incluye un **generador de entornos** para cada clase de medios que selecciona hábitats particulares (con ciertas posibilidades) en los que ejecutar los agentes. Por ejemplo, el generador de un entorno

para un agente aspiradora inicializa el patrón de suciedad y la localización del agente de forma aleatoria. Después, es interesante evaluar la eficacia media del agente en el contexto de la clase del entorno. Un agente racional para una clase de entorno maximiza el rendimiento medio. Los Ejercicios del 2.7 al 2.12 guían el proceso de desarrollo de una clase de entornos y la evaluación de varios agentes.

2.4 Estructura de los agentes

PROGRAMA
DEL AGENTE

ARQUITECTURA

Hasta este momento se ha hablado de los agentes describiendo su *conducta*, la acción que se realiza después de una secuencia de percepciones dada. Ahora, se trata de centrarse en el núcleo del problema y hablar sobre cómo trabajan internamente. El trabajo de la IA es diseñar el **programa del agente** que implemente la función del agente que proyecta las percepciones en las acciones. Se asume que este programa se ejecutará en algún tipo de computador con sensores físicos y actuadores, lo cual se conoce como **arquitectura**:

$$\text{Agente} = \text{arquitectura} + \text{programa}$$

Obviamente, el programa que se elija tiene que ser apropiado para la arquitectura. Si el programa tiene que recomendar acciones como *Caminar*, la arquitectura tiene que tener piernas. La arquitectura puede ser un PC común, o puede ser un coche robotizado con varios computadores, cámaras, y otros sensores a bordo. En general, la arquitectura hace que las percepciones de los sensores estén disponibles para el programa, ejecuta los programas, y se encarga de que los actuadores pongan en marcha las acciones generadas. La mayor parte de este libro se centra en el diseño de programas para agentes, aunque los Capítulos 24 y 25 tratan sobre sensores y actuadores.

Programas de los agentes

Los programas de los agentes que se describen en este libro tienen la misma estructura: reciben las percepciones actuales como entradas de los sensores y devuelven una acción a los actuadores⁷. Hay que tener en cuenta la diferencia entre los programas de los agentes, que toman la percepción actual como entrada, y la función del agente, que recibe la percepción histórica completa. Los programas de los agentes reciben sólo la percepción actual como entrada porque no hay nada más disponible en el entorno; si las acciones del agente dependen de la secuencia completa de percepciones, el agente tendría que recordar las percepciones.

Los programas de los agentes se describirán con la ayuda de un sencillo lenguaje pseudocódigo que se define en el Apéndice B. El repositorio de código disponible en Inter-

⁷ Hay otras posibilidades para definir la estructura del programa para el agente; por ejemplo, los programas para agentes pueden ser **subrutinas** que se ejecuten asincrónicamente en el entorno de trabajo. Cada una de estas subrutinas tiene un puerto de entrada y salida y consisten en un bucle que interpreta las entradas del puerto como percepciones y escribe acciones en el puerto de salida.

función AGENTE-DIRIGIDO-MEDIANTE TABLA(*percepción*) **devuelve** una acción
variables estáticas: *percepciones*, una secuencia, vacía inicialmente
tabla, una tabla de acciones, indexada por las secuencias de percepciones, totalmente definida inicialmente

añadir la *percepción* al final de las *percepciones*
acción \leftarrow CONSULTA(*percepciones*, *tabla*)
devolver *acción*

Figura 2.7 El programa AGENTE-DIRIGIDO-MEDIANTE TABLA se invoca con cada nueva percepción y devuelve una acción en cada momento. Almacena la secuencia de percepciones utilizando su propia estructura de datos privada.

net contiene implementaciones en lenguajes de programación reales. Por ejemplo, la Figura 2.7 muestra un programa de agente muy sencillo que almacena la secuencia de percepciones y después las compara con las secuencias almacenadas en la tabla de acciones para decidir qué hacer. La tabla representa explícitamente la función que define el programa del agente. Para construir un agente racional de esta forma, los diseñadores deben realizar una tabla que contenga las acciones apropiadas para cada secuencia posible de percepciones.

Intuitivamente se puede apreciar por qué la propuesta de dirección-mediante-tabla para la construcción de agentes está condenada al fracaso. Sea P el conjunto de posibles percepciones y T el tiempo de vida del agente (el número total de percepciones que recibirá). La tabla de búsqueda contendrá $\sum_{t=1}^T |P|^t$ entradas. Si consideramos ahora el taxi automatizado: la entrada visual de una cámara individual es de 27 megabytes por segundo (30 fotografías por segundo, 640×480 pixels con 24 bits de información de colores). Lo cual genera una tabla de búsqueda con más de $10^{250,000,000,000}$ entradas por hora de conducción. Incluso la tabla de búsqueda del ajedrez (un fragmento del mundo real pequeño y obediente) tiene por lo menos 10^{150} entradas. El tamaño exageradamente grande de estas tablas (el número de átomos en el universo observable es menor que 10^{80}) significa que (a) no hay agente físico en este universo que tenga el espacio suficiente como para almacenar la tabla, (b) el diseñador no tendrá tiempo para crear la tabla, (c) ningún agente podría aprender todas las entradas de la tabla a partir de su experiencia, y (d) incluso si el entorno es lo suficientemente simple para generar una tabla de un tamaño razonable, el diseñador no tiene quien le asesore en la forma en la que llenar la tabla.

A pesar de todo ello, el AGENTE-DIRIGIDO-MEDIANTE TABLA *hace* lo que nosotros queremos: implementa la función deseada para el agente. El desafío clave de la IA es encontrar la forma de escribir programas, que en la medida de lo posible, reproduzcan un comportamiento racional a partir de una pequeña cantidad de código en vez de a partir de una tabla con un gran número de entradas. Existen bastantes ejemplos que muestran qué se puede hacer con éxito en otras áreas: por ejemplo, las grandes tablas de las raíces cuadradas utilizadas por ingenieros y estudiantes antes de 1970 se han reemplazado por un programa de cinco líneas que implementa el método de Newton en las calculadoras electrónicas. La pregunta es, en el caso del comportamiento inteligente general, ¿puede la IA hacer lo que Newton hizo con las raíces cuadradas? Creemos que la respuesta es afirmativa.

En lo que resta de esta sección se presentan los cuatro tipos básicos de programas para agentes que encarnan los principios que subyacen en casi todos los sistemas inteligentes.

- Agentes reactivos simples.
- Agentes reactivos basados en modelos.
- Agentes basados en objetivos.
- Agentes basados en utilidad.

Después se explica, en términos generales, cómo convertir todos ellos en *agentes que aprendan*.

Agentes reactivos simples

AGENTE REACTIVO SIMPLE

El tipo de agente más sencillo es el **agente reactivo simple**. Estos agentes seleccionan las acciones sobre la base de las percepciones *actuales*, ignorando el resto de las percepciones históricas. Por ejemplo, el agente aspiradora cuya función de agente se presentó en la Figura 2.3 es un agente reactivo simple porque toma sus decisiones sólo con base en la localización actual y si ésta está sucia. La Figura 2.8 muestra el programa para este agente.

Hay que tener en cuenta que el programa para el agente aspiradora es muy pequeño comparado con su tabla correspondiente. La reducción más clara se obtiene al ignorar la historia de percepción, que reduce el número de posibilidades de 4^T a sólo 4. Otra reducción se basa en el hecho de que cuando la cuadrícula actual está sucia, la acción no depende de la localización.

Imáginese que es el conductor del taxi automático. Si el coche que circula delante frena, y las luces de freno se encienden, entonces lo advertiría y comenzaría a frenar. En otras palabras, se llevaría a cabo algún tipo de procesamiento sobre las señales visuales para establecer la condición que se llama «El coche que circula delante está frenando». Esto dispara algunas conexiones establecidas en el programa del agente para que se ejecute la acción «iniciar frenado». Esta conexión se denomina **regla de condición-acción**⁸, y se representa por

si el-coche-que-circula-delante-está-frenando **entonces** iniciar-frenada.

función AGENTE-ASPIRADORA-REACTIVO([localización, estado]) devuelve una acción

si estado = Sucio **entonces devolver** Aspirar
de otra forma, si localización = A **entonces devolver** Derecha
de otra forma, si localización = B **entonces devolver** Izquierda

Figura 2.8 Programa para el agente aspiradora de reactivo simple en el entorno definido por las dos cuadrículas. Este programa implementa la función de agente presentada en la Figura 2.3.

⁸ También llamadas reglas de situación-acción, producciones, o reglas si-entonces.

Los humanos también tienen muchas de estas conexiones, algunas de las cuales son respuestas aprendidas (como en el caso de la conducción) y otras son reacciones innatas (como parpadear cuando algo se acerca al ojo). A lo largo de esta obra, se estudiarán diferentes formas en las que se pueden aprender e implementar estas conexiones.

El programa de la Figura 2.8 es específico para el entorno concreto de la aspiradora. Una aproximación más general y flexible es la de construir primero un intérprete de propósito general para reglas de condición-acción y después crear conjuntos de reglas para entornos de trabajo específicos. La Figura 2.9 presenta la estructura de este programa general de forma esquemática, mostrando cómo las reglas de condición-acción permiten al agente generar la conexión desde las percepciones a las acciones. No se preocupe si le parece trivial; pronto se complicará. Se utilizan rectángulos para denotar el estado interno actual del proceso de toma de decisiones del agente y óvalos para representar la información base utilizada en el proceso. El programa del agente, que es también muy simple, se muestra en la Figura 2.10. La función INTERPRETAR-ENTRADA genera una descripción abstracta del estado actual a partir de la percepción, y la función REGLA-COINCIDENCIA devuelve la primera regla del conjunto de reglas que coincide con la descripción del estado dada. Hay que tener en cuenta que la descripción en términos de «reglas»

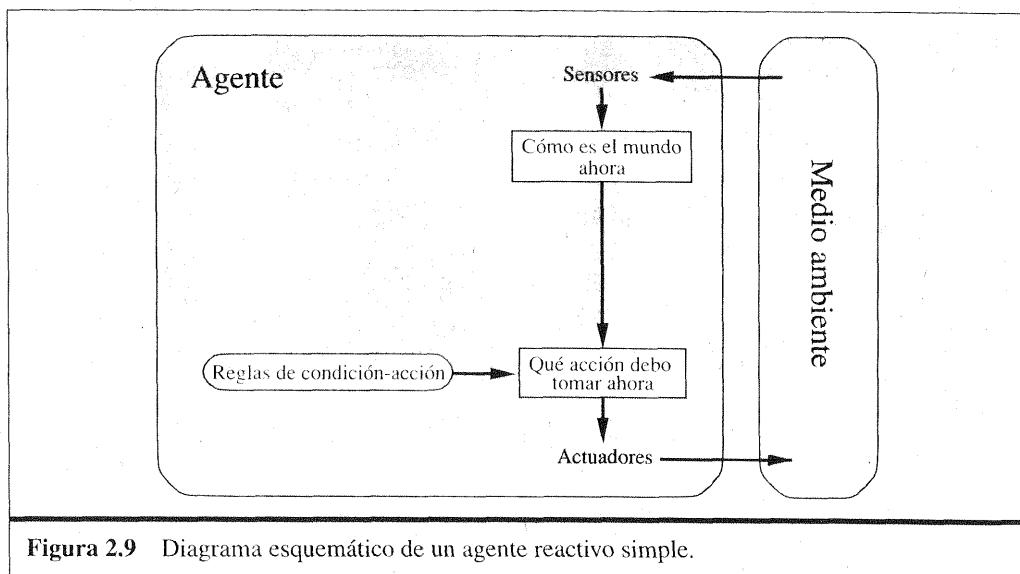


Figura 2.9 Diagrama esquemático de un agente reactivo simple.

función AGENTE-REACTIVO-SIMPLE(percepción) devuelve una acción
estático: *reglas*, un conjunto de reglas condición-acción

```

estado ← INTERPRETAR-ENTRADA(percepción)
regla ← REGLA-COINCIDENCIA(estado, reglas)
acción ← REGLA-ACCIÓN[regla]
devolver acción
  
```

Figura 2.10 Un agente reactivo simple, que actúa de acuerdo a la regla cuya condición coincida con el estado actual, definido por la percepción.

y «coincidencias» es puramente conceptual; las implementaciones reales pueden ser tan simples como colecciones de puertas lógicas implementando un circuito booleano.



Los agentes reactivos simples tienen la admirable propiedad de ser simples, pero poseen una inteligencia muy limitada. El agente de la Figura 2.10 funcionará *sólo si se puede tomar la decisión correcta sobre la base de la percepción actual, lo cual es posible sólo si el entorno es totalmente observable*. Incluso el que haya una pequeña parte que no se pueda observar puede causar serios problemas. Por ejemplo, la regla de frenado dada anteriormente asume que la condición *el-coche-que-circula-delante-está-frenando* se puede determinar a partir de la percepción actual (imagen de vídeo actual) si el coche de enfrente tiene un sistema centralizado de luces de freno. Desafortunadamente, los modelos antiguos tienen diferentes configuraciones de luces traseras, luces de frenado, y de intermitentes, y no es siempre posible saber a partir de una única imagen si el coche está frenando. Un agente reactivo simple conduciendo detrás de un coche de este tipo puede frenar continuamente y de manera innecesaria, o peor, no frenar nunca.

Un problema similar aparece en el mundo de la aspiradora. Supongamos que se elimina el sensor de localización de un agente aspiradora reactivo simple, y que sólo tiene un sensor de suciedad. Un agente de este tipo tiene sólo dos percepciones posibles: *[Sucio]* y *[Limpio]*. Puede *Aspirar* cuando se encuentra con *[Sucio]*. ¿Qué debe hacer cuando se encuentra con *[Limpio]*? Si se desplaza a la *Izquierda* se equivoca (siempre) si está en la cuadrícula *A*, y si de desplaza a la *Derecha* se equivoca (siempre) si está en la cuadrícula *B*. Los bucles infinitos son a menudo inevitables para los agentes reactivos simples que operan en algunos entornos parcialmente observables.

ALEATORIO

Salir de los bucles infinitos es posible si los agentes pueden seleccionar sus acciones **aleatoriamente**. Por ejemplo, si un agente aspiradora percibe *[Limpio]*, puede lanzar una moneda y elegir entre *Izquierda* y *Derecha*. Es fácil mostrar que el agente se moverá a la otra cuadrícula en una media de dos pasos. Entonces, si la cuadrícula está sucia, la limpiará y la tarea de limpieza se completará. Por tanto, un agente reactivo simple con capacidad para elegir acciones de manera aleatoria puede mejorar los resultados que proporciona un agente reactivo simple determinista.

ESTADO INTERNO

En la Sección 2.3 se mencionó que un comportamiento aleatorio de un tipo adecuado puede resultar racional en algunos entornos multiagente. En entornos de agentes individuales, el comportamiento aleatorio *no* es normalmente racional. Es un truco útil que ayuda a los agentes reactivos simples en algunas situaciones, pero en la mayoría de los casos se obtendrán mejores resultados con agentes deterministas más sofisticados.

Agentes reactivos basados en modelos

La forma más efectiva que tienen los agentes de manejar la visibilidad parcial es *almacenar información de las partes del mundo que no pueden ver*. O lo que es lo mismo, el agente debe mantener algún tipo de **estado interno** que dependa de la historia percibida y que de ese modo refleje por lo menos alguno de los aspectos no observables del estado actual. Para el problema de los frenos, el estado interno no es demasiado extenso, sólo la fotografía anterior de la cámara, facilitando al agente la detección de dos luces rojas encendiéndose y apagándose simultáneamente a los costados del vehículo. Para

otros aspectos de la conducción, como un cambio de carril, el agente tiene que mantener información de la posición del resto de los coches si no los puede ver.

La actualización de la información de estado interno según pasa el tiempo requiere codificar dos tipos de conocimiento en el programa del agente. Primero, se necesita alguna información acerca de cómo evoluciona el mundo independientemente del agente, por ejemplo, que un coche que está adelantando estará más cerca, detrás, que en un momento inmediatamente anterior. Segundo, se necesita más información sobre cómo afectan al mundo las acciones del agente, por ejemplo, que cuando el agente gire hacia la derecha, el coche gira hacia la derecha o que después de conducir durante cinco minutos hacia el norte en la autopista se avanzan cinco millas hacia el norte a partir del punto en el que se estaba cinco minutos antes. Este conocimiento acerca de «cómo funciona el mundo», tanto si está implementado con un circuito booleano simple o con teorías científicas completas, se denomina **modelo** del mundo. Un agente que utilice este modelo es un **agente basado en modelos**.

AGENTE BASADO
EN MODELOS

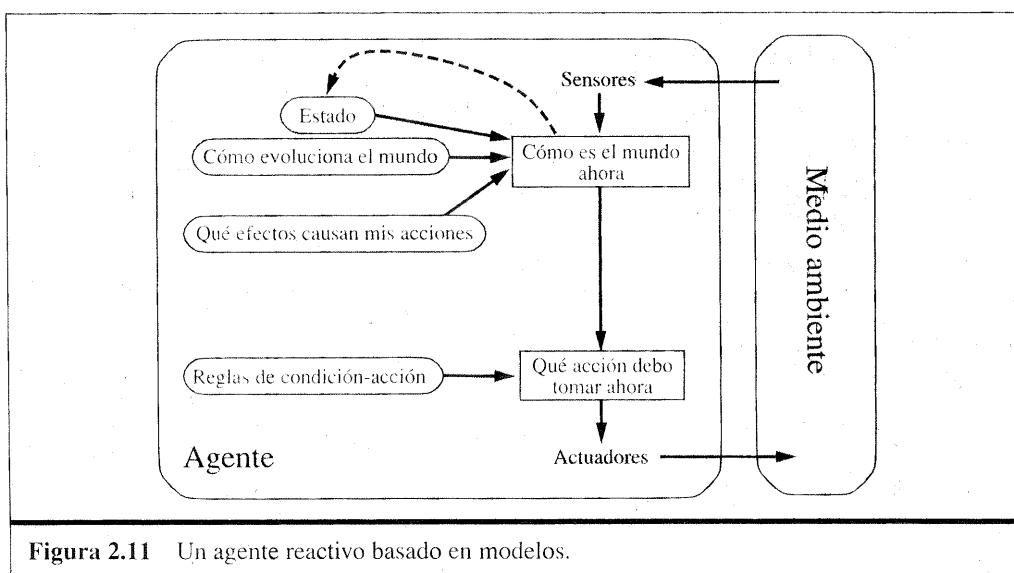


Figura 2.11 Un agente reactivo basado en modelos.

función AGENTE-REACTIVO-CON-ESTADO(*percepción*) **devuelve** una acción
estático: *estado*, una descripción actual del estado del mundo
reglas, un conjunto de reglas condición-acción
acción, la acción más reciente, inicialmente ninguna

```

estado  $\leftarrow$  ACTUALIZAR-ESTADO(estado, acción, percepción)
regla  $\leftarrow$  REGLA-COINCIDENCIA(estado, reglas)
acción  $\leftarrow$  REGLA-ACCIÓN[regla]
devolver acción
```

Figura 2.12 Un agente reactivo basado en modelos, que almacena información sobre el estado actual del mundo utilizando un modelo interno. Después selecciona una acción de la misma forma que el agente reactivo.

La Figura 2.11 proporciona la estructura de un agente reactivo simple con estado interno, muestra cómo la percepción actual se combina con el estado interno antiguo para generar la descripción actualizada del estado actual. La Figura 2.12 muestra el programa del agente. La parte interesante es la correspondiente a la función Actualizar-Estado, que es la responsable de la creación de la nueva descripción del estado interno. Además de interpretar la nueva percepción a partir del conocimiento existente sobre el estado, utiliza información relativa a la forma en la que evoluciona el mundo para conocer más sobre las partes del mundo que no están visibles; para ello debe conocer cuál es el efecto de las acciones del agente sobre el estado del mundo. Los Capítulos 10 y 17 ofrecen ejemplos detallados.

Agentes basados en objetivos

El conocimiento sobre el estado actual del mundo no es siempre suficiente para decidir qué hacer. Por ejemplo, en un cruce de carreteras, el taxista puede girar a la izquierda, girar a la derecha o seguir hacia adelante. La decisión correcta depende de dónde quiere ir el taxi. En otras palabras, además de la descripción del estado actual, el agente necesita algún tipo de información sobre su **meta** que describa las situaciones que son deseables, por ejemplo, llegar al destino propuesto por el pasajero. El programa del agente se puede combinar con información sobre los resultados de las acciones posibles (la misma información que se utilizó para actualizar el estado interno en el caso del agente reflexivo) para elegir las acciones que permitan alcanzar el objetivo. La Figura 2.13 muestra la estructura del agente basado en objetivos.

En algunas ocasiones, la selección de acciones basadas en objetivos es directa, cuando alcanzar los objetivos es el resultado inmediato de una acción individual. En otras oca-

META

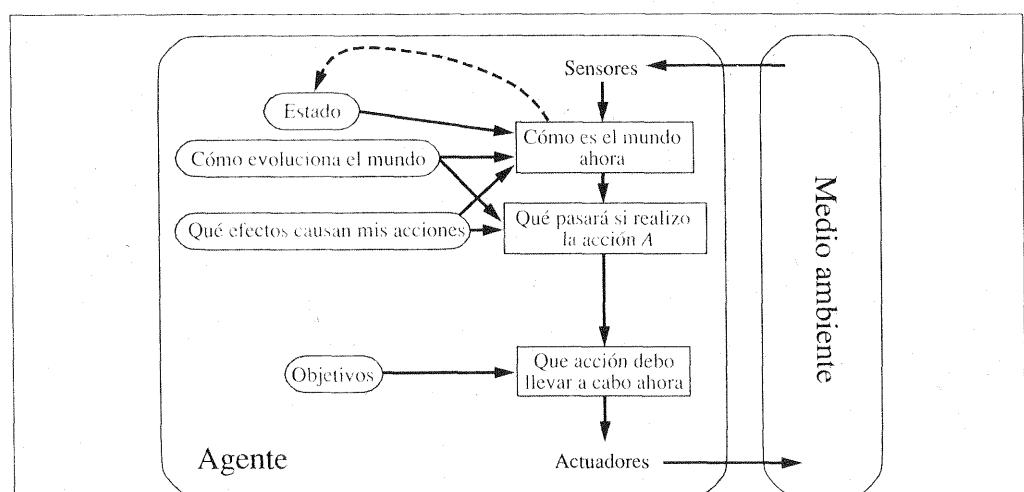


Figura 2.13 Un agente basado en objetivos y basado en modelos, que almacena información del estado del mundo así como del conjunto de objetivos que intenta alcanzar, y que es capaz de seleccionar la acción que eventualmente lo guiará hacia la consecución de sus objetivos.

siones, puede ser más complicado, cuando el agente tiene que considerar secuencias complejas para encontrar el camino que le permita alcanzar el objetivo. **Búsqueda** (Capítulos del 3 al 6) y **planificación** (Capítulos 11 y 12) son los subcampos de la IA centrados en encontrar secuencias de acciones que permitan a los agentes alcanzar sus metas.

Hay que tener en cuenta que la toma de decisiones de este tipo es fundamentalmente diferente de las reglas de condición–acción descritas anteriormente, en las que hay que tener en cuenta consideraciones sobre el futuro (como «¿qué pasará si yo hago esto y esto?» y «¿me hará esto feliz?»). En los diseños de agentes reactivos, esta información no está representada explícitamente, porque las reglas que maneja el agente proyectan directamente las percepciones en las acciones. El agente reactivo frena cuando ve luces de freno. Un agente basado en objetivos, en principio, puede razonar que si el coche que va delante tiene encendidas las luces de frenado, está reduciendo su velocidad. Dada la forma en la que el mundo evoluciona normalmente, la única acción que permite alcanzar la meta de no chocarse con otros coches, es frenar.

Aunque el agente basado en objetivos pueda parecer menos eficiente, es más flexible ya que el conocimiento que soporta su decisión está representado explícitamente y puede modificarse. Si comienza a llover, el agente puede actualizar su conocimiento sobre cómo se comportan los frenos; lo cual implicará que todas las formas de actuar relevantes se alteren automáticamente para adaptarse a las nuevas circunstancias. Para el agente reactivo, por otro lado, se tendrán que rescribir muchas reglas de condición–acción. El comportamiento del agente basado en objetivos puede cambiarse fácilmente para que se dirija a una localización diferente. Las reglas de los agentes reactivos relacionadas con cuándo girar y cuándo seguir recto son válidas sólo para un destino concreto y tienen que modificarse cada vez que el agente se dirija a cualquier otro lugar distinto.

Agentes basados en utilidad

Las metas por sí solas no son realmente suficientes para generar comportamiento de gran calidad en la mayoría de los entornos. Por ejemplo, hay muchas secuencias de acciones que llevarán al taxi a su destino (y por tanto a alcanzar su objetivo), pero algunas son más rápidas, más seguras, más fiables, o más baratas que otras. Las metas sólo proporcionan una cruda distinción binaria entre los estados de «felicidad» y «tristeza», mientras que una medida de eficiencia más general debería permitir una comparación entre estados del mundo diferentes de acuerdo al nivel exacto de felicidad que el agente alcance cuando se llegue a un estado u otro. Como el término «felicidad» no suena muy científico, la terminología tradicional utilizada en estos casos para indicar que se prefiere un estado del mundo a otro es que un estado tiene más **utilidad** que otro para el agente⁹.

Una **función de utilidad** proyecta un estado (o una secuencia de estados) en un número real, que representa un nivel de felicidad. La definición completa de una función de utilidad permite tomar decisiones racionales en dos tipos de casos en los que las metas son inadecuadas. Primero, cuando haya objetivos conflictivos, y sólo se puedan al-

UTILIDAD

FUNCIÓN DE UTILIDAD

⁹ La palabra «utilidad» aquí se refiere a «la cualidad de ser útil».

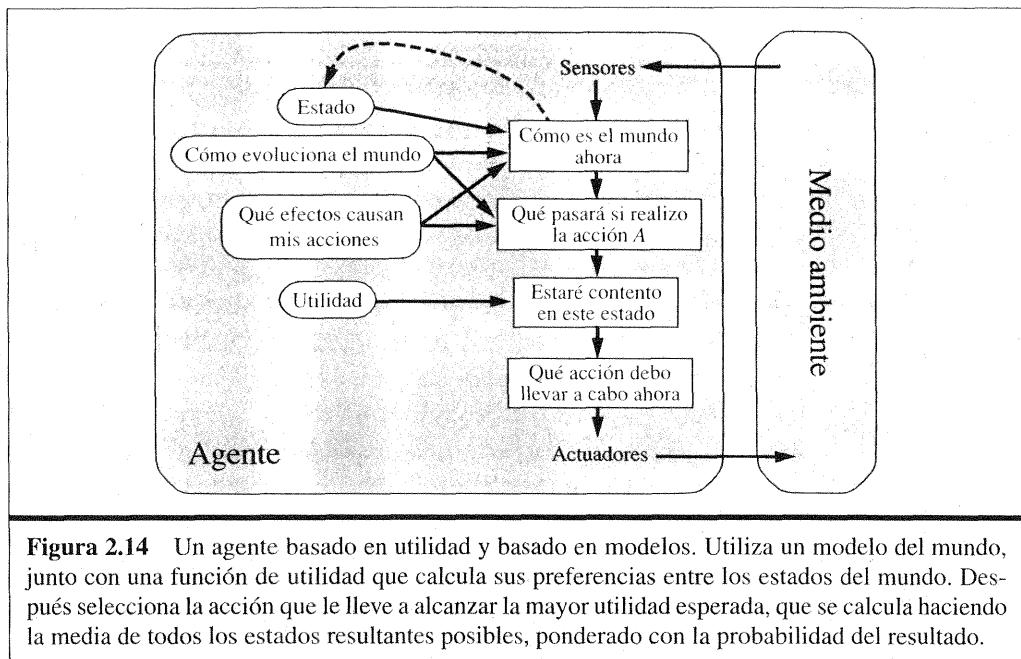


Figura 2.14 Un agente basado en utilidad y basado en modelos. Utiliza un modelo del mundo, junto con una función de utilidad que calcula sus preferencias entre los estados del mundo. Despues selecciona la acción que le lleve a alcanzar la mayor utilidad esperada, que se calcula haciendo la media de todos los estados resultantes posibles, ponderado con la probabilidad del resultado.

canzar algunos de ellos (por ejemplo, velocidad y seguridad), la función de utilidad determina el equilibrio adecuado. Segundo, cuando haya varios objetivos por los que se pueda guiar el agente, y ninguno de ellos se pueda alcanzar con certeza, la utilidad proporciona un mecanismo para ponderar la probabilidad de éxito en función de la importancia de los objetivos.

En el Capítulo 16, se mostrará cómo cualquier agente racional debe comportarse *como si* tuviese una función de utilidad cuyo valor esperado tiene que maximizar. Por tanto, un agente que posea una función de utilidad *explícita* puede tomar decisiones racionales, y lo puede hacer con la ayuda de un algoritmo de propósito general que no dependa de la función específica de utilidad a maximizar. De esta forma, la definición «global» de racionalidad (identificando como racionales aquellas funciones de los agentes que proporcionan el mayor rendimiento) se transforma en una restricción «local» en el diseño de agentes racionales que se puede expresar con un simple programa.

La Figura 2.14 muestra la estructura de un agente basado en utilidad. En la Parte IV aparecen programas de agentes basados en utilidad, donde se presentan agentes que toman decisiones y que deben trabajar con la incertidumbre inherente a los entornos parcialmente observables.

Agentes que aprenden

Se han descrito programas para agentes que poseen varios métodos para seleccionar acciones. Hasta ahora no se ha explicado cómo *poner en marcha* estos programas de agentes. Turing (1950), en su temprano y famoso artículo, consideró la idea de programar sus máquinas inteligentes a mano. Estimó cuánto tiempo podía llevar y concluyó que «Se-

ría deseable utilizar algún método más rápido». El método que propone es construir máquinas que aprendan y después enseñarlas. En muchas áreas de IA, éste es ahora el método más adecuado para crear sistemas novedosos. El aprendizaje tiene otras ventajas, como se ha explicado anteriormente: permite que el agente opere en medios inicialmente desconocidos y que sea más competente que si sólo utilizase un conocimiento inicial. En esta sección, se introducen brevemente las principales ideas en las que se basan los agentes que aprenden. En casi todos los capítulos de este libro se comentan las posibilidades y métodos de aprendizaje de tipos de agentes concretos. La Parte VI profundiza más en los algoritmos de aprendizaje en sí mismos.

Un agente que aprende se puede dividir en cuatro componentes conceptuales, tal y como se muestra en la Figura 2.15. La distinción más importante entre el **elemento de aprendizaje** y el **elemento de actuación** es que el primero está responsabilizado de hacer mejoras y el segundo se responsabiliza de la selección de acciones externas. El elemento de actuación es lo que anteriormente se había considerado como el agente completo: recibe estímulos y determina las acciones a realizar. El elemento de aprendizaje se realimenta con las **críticas** sobre la actuación del agente y determina cómo se debe modificar el elemento de actuación para proporcionar mejores resultados en el futuro.

El diseño del elemento de aprendizaje depende mucho del diseño del elemento de actuación. Cuando se intenta diseñar un agente que tenga capacidad de aprender, la primera cuestión a solucionar no es ¿cómo se puede enseñar a aprender?, sino ¿qué tipo de elemento de actuación necesita el agente para llevar a cabo su objetivo, cuando haya aprendido cómo hacerlo? Dado un diseño para un agente, se pueden construir los mecanismos de aprendizaje necesarios para mejorar cada una de las partes del agente.

La crítica indica al elemento de aprendizaje qué tal lo está haciendo el agente con respecto a un nivel de actuación fijo. La crítica es necesaria porque las percepciones por sí mismas no prevén una indicación del éxito del agente. Por ejemplo, un programa de

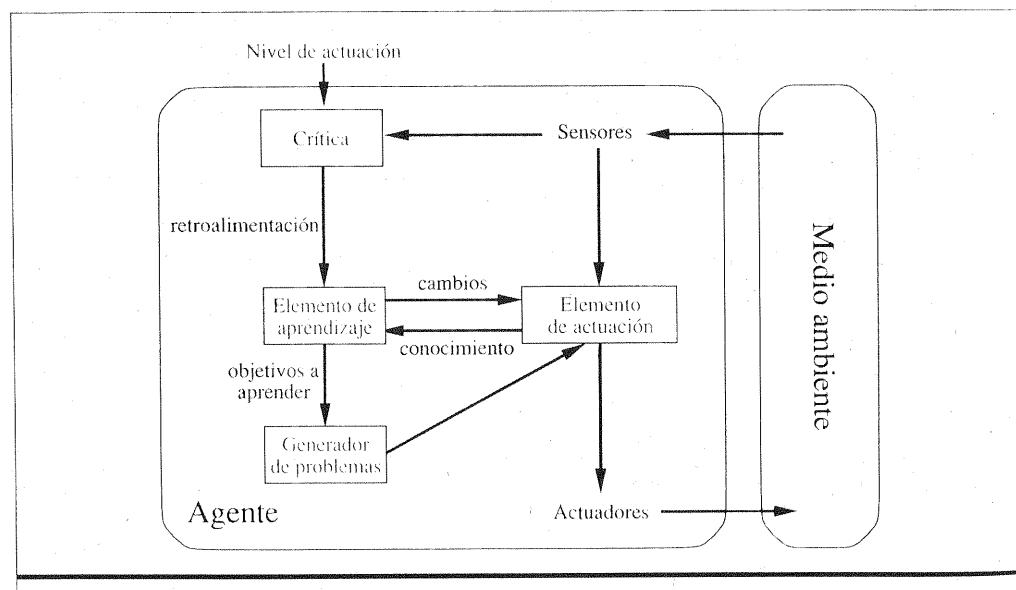


Figura 2.15 · Modelo general para agentes que aprenden.

ajedrez puede recibir una percepción indicando que ha dado jaque mate a su oponente, pero necesita tener un nivel de actuación que le indique que ello es bueno; la percepción por sí misma no lo indica. Es por tanto muy importante fijar el nivel de actuación. Conceptualmente, se debe tratar con él como si estuviese fuera del agente, ya que éste no debe modificarlo para satisfacer su propio interés.

GENERADOR DE PROBLEMAS

El último componente del agente con capacidad de aprendizaje es el **generador de problemas**. Es responsable de sugerir acciones que lo guiarán hacia experiencias nuevas e informativas. Lo interesante es que si el elemento de actuación sigue su camino, puede continuar llevando a cabo las acciones que sean mejores, dado su conocimiento. Pero si el agente está dispuesto a explorar un poco, y llevar a cabo algunas acciones que no sean totalmente óptimas a corto plazo, puede descubrir acciones mejores a largo plazo. El trabajo del generador de problemas es sugerir estas acciones exploratorias. Esto es lo que los científicos hacen cuando llevan a cabo experimentos. Galileo no pensaba que tirar piedras desde lo alto de una torre en Pisa tenía un valor por sí mismo. Él no trataba de romper piedras ni de cambiar la forma de pensar de transeúntes desafortunados que paseaban por el lugar. Su intención era adaptar su propia mente, para identificar una teoría que definiese mejor el movimiento de los objetos.

Para concretar el diseño total, se puede volver a utilizar el ejemplo del taxi automatizado. El elemento de actuación consiste en la colección de conocimientos y procedimientos que tiene el taxi para seleccionar sus acciones de conducción. El taxi se pone en marcha y circula utilizando este elemento de actuación. La crítica observa el mundo y proporciona información al elemento de aprendizaje. Por ejemplo, después de que el taxi se sitúe tres carriles hacia la izquierda de forma rápida, la crítica observa el lenguaje escandaloso que utilizan otros conductores. A partir de esta experiencia, el elemento de aprendizaje es capaz de formular una regla que indica que ésta fue una mala acción, y el elemento de actuación se modifica incorporando la nueva regla. El generador de problemas debe identificar ciertas áreas de comportamiento que deban mejorarse y sugerir experimentos, como probar los frenos en carreteras con tipos diferentes de superficies y bajo condiciones distintas.

El elemento de aprendizaje puede hacer cambios en cualquiera de los componentes de «conocimiento» que se muestran en los diagramas de agente (Figuras 2.9, 2.11, 2.13, y 2.14). Los casos más simples incluyen el aprendizaje directo a partir de la secuencia percibida. La observación de pares de estados sucesivos del entorno puede permitir que el agente aprenda «cómo evoluciona el mundo», y la observación de los resultados de sus acciones puede permitir que el agente aprenda «qué hacen sus acciones». Por ejemplo, si el taxi ejerce una cierta presión sobre los frenos cuando está circulando por una carretera mojada, acto seguido conocerá cómo decelera el coche. Claramente, estas dos tareas de aprendizaje son más difíciles si sólo existe una vista parcial del medio.

Las formas de aprendizaje mostradas en los párrafos precedentes no necesitan el acceso a niveles de actuación externo, de alguna forma, el nivel es el que se utiliza universalmente para hacer pronósticos de acuerdo con la experimentación. La situación es ligeramente más compleja para un agente basado en utilidad que deseé adquirir información para crear su función de utilidad. Por ejemplo, se supone que el agente conductor del taxi no recibe propina de los pasajeros que han recorrido un trayecto de forma incómoda debido a una mala conducción. El nivel de actuación externo debe informar

al agente de que la pérdida de propinas tiene una contribución negativa en su nivel de actuación medio; entonces el agente puede aprender que «maniobras violentas no contribuyen a su propia utilidad». De alguna manera, el nivel de actuación identifica parte de las percepciones entrantes como **recompensas** (o **penalizaciones**) que generan una respuesta directa en la calidad del comportamiento del agente. Niveles de actuación integrados como el dolor y el hambre en animales se pueden enmarcar en este contexto. El Capítulo 21 discute estos asuntos.

En resumen, los agentes tienen una gran variedad de componentes, y estos componentes se pueden representar de muchas formas en los programas de agentes, por lo que, parece haber una gran variedad de métodos de aprendizaje. Existe, sin embargo, una visión unificada sobre un tema fundamental. El aprendizaje en el campo de los agentes inteligentes puede definirse como el proceso de modificación de cada componente del agente, lo cual permite a cada componente comportarse más en consonancia con la información que se recibe, lo que por tanto permite mejorar el nivel medio de actuación del agente.

2.5 Resumen

En este capítulo se ha realizado un recorrido rápido por el campo de la IA, que se ha presentado como la ciencia del diseño de los agentes. Los puntos más importantes a tener en cuenta son:

- Un **agente** es algo que percibe y actúa en un medio. La **función del agente** para un agente especifica la acción que debe realizar un agente como respuesta a cualquier secuencia percibida.
- La **medida de rendimiento** evalúa el comportamiento del agente en un medio. Un **agente racional** actúa con la intención de maximizar el valor esperado de la medida de rendimiento, dada la secuencia de percepciones que ha observado hasta el momento.
- Las especificaciones del **entorno de trabajo** incluyen la medida de rendimiento, el medio externo, los actuadores y los sensores. El primer paso en el diseño de un agente debe ser siempre la especificación, tan completa como sea posible, del entorno de trabajo.
- El entorno de trabajo varía según distintos parámetros. Pueden ser total o parcialmente visibles, deterministas o estocásticos, episódicos o secuenciales, estáticos o dinámicos, discretos o continuos, y formados por un único agente o por varios agentes.
- El **programa del agente** implementa la función del agente. Existe una gran variedad de diseños de programas de agentes, y reflejan el tipo de información que se hace explícita y se utiliza en el proceso de decisión. Los diseños varían en eficiencia, solidez y flexibilidad. El diseño apropiado del programa del agente depende en gran medida de la naturaleza del medio.
- **Los agentes reactivos simples** responden directamente a las percepciones, mientras que los **agentes reactivos basados en modelos** mantienen un estado interno

que les permite seguir el rastro de aspectos del mundo que no son evidentes según las percepciones actuales. Los agentes basados en objetivos actúan con la intención de alcanzar sus metas, y los agentes basados en utilidad intentan maximizar su «felicidad» deseada.

- Todos los agentes pueden mejorar su eficacia con la ayuda de mecanismos de **aprendizaje**.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

CONTROLADOR

El papel central de la acción en la inteligencia (la noción del razonamiento práctico) se remonta por lo menos a la obra *Nicomachean Ethics* de Aristóteles. McCarthy (1958) trató también el tema del razonamiento práctico en su influyente artículo *Programs with Common Sense*. Los campos de la robótica y la teoría de control tienen interés, por su propia naturaleza, en la construcción de agentes físicos. El concepto de un **controlador**, en el ámbito de la teoría de control, es idéntico al de un agente en IA. Quizá sorprendentemente, la IA se ha concentrado durante la mayor parte de su historia en componentes aislados de agentes (sistemas que responden a preguntas, demostración de teoremas, sistemas de visión, y demás) en vez de en agentes completos. La discusión sobre agentes que se presenta en el libro de Genesereth y Nilsson (1987) fue una influyente excepción. El concepto de agente en sí está aceptado ampliamente ahora en el campo y es un tema central en libros recientes (Poole *et al.*, 1998; Nilsson, 1998).

El Capítulo 1 muestra las raíces del concepto de racionalidad en la Filosofía y la Economía. En la IA, el concepto tuvo un interés periférico hasta mediados de los 80, donde comenzó a suscitar muchas discusiones sobre los propios fundamentos técnicos del campo. Un artículo de Jon Doyle (1983) predijo que el diseño de agentes racionales podría llegar a ser la misión central de la IA, mientras otras áreas populares podrían separarse dando lugar a nuevas disciplinas.

Es muy importante tener muy en cuenta las propiedades del medio y sus consecuencias cuando se realiza el diseño de los agentes racionales ya que forma parte de la tradición ligada a la teoría de control [por ejemplo los sistemas de control clásicos (Dorf y Bishop, 1999) manejan medios deterministas y totalmente observables; el control óptimo estocástico (Kumar y Varaiya, 1986) maneja medios parcialmente observables y estocásticos y un control híbrido (Henzinger y Sastry, 1998) maneja entornos que contienen elementos discretos y continuos]. La distinción entre entornos totalmente y parcialmente observables es también central en la literatura sobre **programación dinámica** desarrollada en el campo de la investigación operativa (Puterman, 1994), como se comentará en el Capítulo 17.

Los agentes reactivos fueron los primeros modelos para psicólogos conductistas como Skinner (1953), que intentó reducir la psicología de los organismos estrictamente a correspondencias entrada/salida o estímulo/respuesta. La evolución del behaviourismo hacia el funcionalismo en el campo de la psicología, que estuvo, al menos de forma parcial, dirigida por la aplicación de la metáfora del computador a los agentes (Putnam, 1960; Lewis, 1966) introdujo el estado interno del agente en el nuevo escenario. La mayor par-

te del trabajo realizado en el campo de la IA considera que los agentes reactivos puros con estado interno son demasiado simples para ser muy influyentes, pero los trabajos de Rosenschein (1985) y Brooks (1986) cuestionan esta hipótesis (*véase* el Capítulo 25). En los últimos años, se ha trabajado intensamente para encontrar algoritmos eficientes capaces de hacer un buen seguimiento de entornos complejos (Hamscher *et al.*, 1992). El programa del Agente Remoto que controla la nave espacial Deep Space One (descrito en la página 27) es un admirable ejemplo concreto (Muscettola *et al.*, 1998; Jonsson *et al.*, 2000).

Los agentes basados en objetivos están presentes tanto en las referencias de Aristóteles sobre el razonamiento práctico como en los primeros artículos de McCarthy sobre IA lógica. El robot Shakey (Fikes y Nilsson, 1971; Nilsson, 1984) fue el primer robot construido como un agente basado en objetivos. El análisis lógico completo de un agente basado en objetivos aparece en Genesereth y Nilsson (1987), y Shoham (1993) ha desarrollado una metodología de programación basada en objetivos llamada programación orientada a agentes.

La perspectiva orientada a objetivos también predomina en la psicología cognitiva tradicional, concretamente en el área de la resolución de problemas, como se muestra tanto en el influyente *Human Problem Solving* (Newell y Simon, 1972) como en los últimos trabajos de Newell (1990). Los objetivos, posteriormente definidos como *deseos* (generales) y las *intenciones* (perseguidas en un momento dado), son fundamentales en la teoría de agentes desarrollada por Bratman (1987). Esta teoría ha sido muy influyente tanto en el entendimiento del lenguaje natural como en los sistemas multiagente.

Horvitz *et al.* (1988) sugieren específicamente el uso de la maximización de la utilidad esperada concebida racionalmente como la base de la IA. El texto de Pearl (1988) fue el primero en IA que cubrió las teorías de la probabilidad y la utilidad en profundidad; su exposición de métodos prácticos de razonamiento y toma de decisiones con incertidumbre fue, posiblemente, el factor individual que más influyó en el desarrollo de los agentes basados en utilidad en los 90 (*véase* la Parte V).

El diseño general de agentes que aprenden representado en la Figura 2.15 es un clásico de la literatura sobre aprendizaje automático (Buchanan *et al.*, 1978; Mitchell, 1997). Ejemplos de diseños, implementados en programas, se remontan, como poco, hasta los programas que aprendían a jugar al ajedrez de Arthur Samuel (1959, 1967). La Parte VI está dedicada al estudio en profundidad de los agentes que aprenden.

El interés en los agentes y en el diseño de agentes ha crecido rápidamente en los últimos años, en parte por la expansión de Internet y la necesidad observada de desarrollar **softbots** (robots software) automáticos y móviles (Etzioni y Weld, 1994). Artículos relevantes pueden encontrarse en *Readings in Agents* (Huhns y Singh, 1998) y *Fundations of Rational Agency* (Wooldridge y Rao, 1999). *Multiagent Systems* (Weiss, 1999) proporciona una base sólida para muchos aspectos del diseño de agentes. Conferencias dedicadas a agentes incluyen la International Conference on Autonomous Agents, la International Workshop on Agent Theories, Architectures, and Languages, y la International Conference on Multiagent Systems. Finalmente, *Dung Beetle Ecology* (Hanski y Cambefort, 1991) proporciona gran cantidad de información interesante sobre el comportamiento de los escarabajos esterceroleros.



EJERCICIOS

2.1 Defina con sus propias palabras los siguientes términos: agente, función de agente, programa de agente, racionalidad, autonomía, agente reactivo, agente basado en modelo, agente basado en objetivos, agente basado en utilidad, agente que aprende.

2.2 Tanto la medida de rendimiento como la función de utilidad miden la eficiencia del agente. Explique la diferencia entre los dos conceptos.

2.3 Este ejercicio explora las diferencias entre las funciones de los agentes y los programas de los agentes.

- a)** ¿Puede haber más de un programa de agente que implemente una función de agente dada? Proponga un ejemplo, o muestre por qué una no es posible.
- b)** ¿Hay funciones de agente que no se pueden implementar con algún programa de agente?
- c)** Dada una arquitectura máquina, ¿implementa cada programa de agente exactamente una función de agente?
- d)** Dada una arquitectura con n bits de almacenamiento, ¿cuántos posibles programas de agente diferentes puede almacenar?

2.4 Examíñese ahora la racionalidad de varias funciones de agentes aspiradora.

- a)** Muestre que la función de agente aspiradora descrita en la Figura 2.3 es realmente racional bajo la hipótesis presentada en la página 36.
- b)** Describa una función para un agente racional cuya medida de rendimiento modificada deduzca un punto por cada movimiento. ¿Requiere el correspondiente programa de agente estado interno?
- c)** Discuta posibles diseños de agentes para los casos en los que las cuadrículas limpias puedan ensuciarse y la geografía del medio sea desconocida. ¿Tiene sentido que el agente aprenda de su experiencia en estos casos? ¿Si es así, qué debe aprender?

2.5 Identifique la descripción REAS que define el entorno de trabajo para cada uno de los siguientes agentes:

- a)** Robot que juega al fútbol;
- b)** Agente para comprar libros en Internet;
- c)** Explorador autónomo de Marte;
- d)** Asistente matemático para la demostración de teoremas.

2.6 Para cada uno de los tipos de agente enumerados en el Ejercicio 2.5, caracterice el medio de acuerdo con las propiedades dadas en la Sección 2.3, y seleccione un diseño de agente adecuado.



Los siguientes ejercicios están relacionados con la implementación de entornos y agentes para el mundo de la aspiradora.

2.7 Implemente un simulador que determine la medida de rendimiento para el entorno del mundo de la aspiradora descrito en la Figura 2.2 y especificado en la página 36. La implementación debe ser modular, de forma que los sensores, actuadores, y las características del entorno (tamaño, forma, localización de la suciedad, etc.) puedan modificar-

se fácilmente. (*Nota:* hay implementaciones disponibles en el repositorio de Internet que pueden ayudar a decidir qué lenguaje de programación y sistema operativo seleccionar).

2.8 Implemente un agente reactivo simple para el entorno de la aspiradora del Ejercicio 2.7. Ejecute el simulador del entorno con este agente para todas las configuraciones iniciales posibles de suciedad y posiciones del agente. Almacene la puntuación de la actuación del agente para cada configuración y la puntuación media global.

2.9 Considere una versión modificada del entorno de la aspiradora del Ejercicio 2.7, en el que se penalice al agente con un punto en cada movimiento.

- a) ¿Puede un agente reactivo simple ser perfectamente racional en este medio? Explíquese.
- b) ¿Qué sucedería con un agente reactivo con estado? Diseñe este agente.
- c) ¿Cómo se responderían las preguntas a y b si las percepciones proporcionan al agente información sobre el nivel de suciedad/limpieza de todas las cuadrículas del entorno?

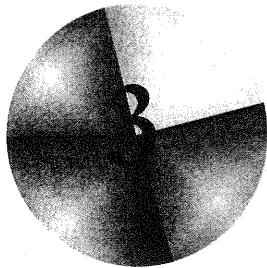
2.10 Considere una versión modificada del entorno de la aspiradora del Ejercicio 2.7, en el que la geografía del entorno (su extensión, límites, y obstáculos) sea desconocida, así como, la disposición inicial de la suciedad. (El agente puede ir hacia *arriba*, *abajo*, así como, hacia la *derecha* y a la *izquierda*.)

- a) ¿Puede un agente reactivo simple ser perfectamente racional en este medio? Explíquese.
- b) ¿Puede un agente reactivo simple con una función de agente aleatoria superar a un agente reactivo simple? Diseñe un agente de este tipo y medir su rendimiento en varios medios.
- c) ¿Se puede diseñar un entorno en el que el agente con la función aleatoria obtenga una actuación muy pobre? Muestre los resultados.
- d) ¿Puede un agente reactivo con estado mejorar los resultados de un agente reactivo simple? Diseñe un agente de este tipo y medir su eficiencia en distintos medios. ¿Se puede diseñar un agente racional de este tipo?

2.11 Repítase el Ejercicio 2.10 para el caso en el que el sensor de localización sea reemplazado por un sensor «de golpes» que detecte si el agente golpea un obstáculo o si se sale fuera de los límites del entorno. Supóngase que el sensor de golpes deja de funcionar. ¿Cómo debe comportarse el agente?

2.12 Los entornos de la aspiradora en los ejercicios anteriores han sido todos deterministas. Discuta posibles programas de agentes para cada una de las siguientes versiones estocásticas:

- a) Ley de Murphy: el 25 por ciento del tiempo, la acción de *Aspirar* falla en la limpieza del suelo si está sucio y deposita suciedad en el suelo si el suelo está limpio. ¿Cómo se ve afectado el agente si el sensor de suciedad da una respuesta incorrecta el diez por ciento de las veces?
- b) Niño pequeño: en cada lapso de tiempo, cada recuadro limpio tiene un diez por ciento de posibilidad de ensuciarse. ¿Puede identificar un diseño para un agente racional en este caso?



Resolver problemas mediante búsqueda

En donde veremos cómo un agente puede encontrar una secuencia de acciones que alcance sus objetivos, cuando ninguna acción simple lo hará.

Los agentes más simples discutidos en el Capítulo 2 fueron los agentes reactivos, los cuales basan sus acciones en una aplicación directa desde los estados a las acciones. Tales agentes no pueden funcionar bien en entornos en los que esta aplicación sea demasiado grande para almacenarla y que tarde mucho en aprenderla. Por otra parte, los agentes basados en objetivos pueden tener éxito considerando las acciones futuras y lo deseable de sus resultados.

AGENTE RESOLVENTE-PROBLEMAS

Este capítulo describe una clase de agente basado en objetivos llamado **agente resolvente-problemas**. Los agentes resolventes-problemas deciden qué hacer para encontrar secuencias de acciones que conduzcan a los estados deseables. Comenzamos definiendo con precisión los elementos que constituyen el «problema» y su «solución», y daremos diferentes ejemplos para ilustrar estas definiciones. Entonces, describimos diferentes algoritmos de propósito general que podamos utilizar para resolver estos problemas y así comparar las ventajas de cada algoritmo. Los algoritmos son **no informados**, en el sentido que no dan información sobre el problema salvo su definición. El Capítulo 4 se ocupa de los algoritmos de búsqueda **informada**, los que tengan cierta idea de dónde buscar las soluciones.

Este capítulo utiliza los conceptos de análisis de algoritmos. Los lectores no familiarizados con los conceptos de complejidad asintótica (es decir, notación $O()$) y la NP completitud, debería consultar el Apéndice A.

3.1 Agentes resolventes-problemas

Se supone que los agentes inteligentes deben maximizar su medida de rendimiento. Como mencionamos en el Capítulo 2, esto puede simplificarse algunas veces si el agente puede

elegir **un objetivo** y trata de satisfacerlo. Primero miraremos el porqué y cómo puede hacerlo.

Imagine un agente en la ciudad de Arad, Rumanía, disfrutando de un viaje de vacaciones. La medida de rendimiento del agente contiene muchos factores: desea mejorar su bronceado, mejorar su rumano, tomar fotos, disfrutar de la vida nocturna, evitar resacas, etcétera. El problema de decisión es complejo implicando muchos elementos y por eso, lee cuidadosamente las guías de viaje. Ahora, supongamos que el agente tiene un billete no reembolsable para volar a Bucarest al día siguiente. En este caso, tiene sentido que el agente elija **el objetivo** de conseguir Bucarest. Las acciones que no alcanzan Bucarest se pueden rechazar sin más consideraciones y el problema de decisión del agente se simplifica enormemente. Los objetivos ayudan a organizar su comportamiento limitando las metas que intenta alcanzar el agente. El primer paso para solucionar un problema es la **formulación del objetivo**, basado en la situación actual y la medida de rendimiento del agente.

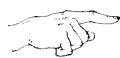
Consideraremos un objetivo como un conjunto de estados del mundo (exactamente aquellos estados que satisfacen el objetivo). La tarea del agente es encontrar qué secuencia de acciones permite obtener un estado objetivo. Para esto, necesitamos decidir qué acciones y estados considerar. Si se utilizaran acciones del tipo «mueve el pie izquierdo hacia delante» o «gira el volante un grado a la izquierda», probablemente el agente nunca encontraría la salida del aparcamiento, no digamos por tanto llegar a Bucarest, porque a ese nivel de detalle existe demasiada incertidumbre en el mundo y habría demasiados pasos en una solución. Dado un objetivo, la **formulación del problema** es el proceso de decidir qué acciones y estados tenemos que considerar. Discutiremos con más detalle este proceso. Por ahora, suponemos que el agente considerará acciones del tipo conducir de una ciudad grande a otra. Consideraremos los estados que corresponden a estar en una ciudad¹ determinada.

Ahora, nuestro agente ha adoptado el objetivo de conducir a Bucarest, y considera a dónde ir desde Arad. Existen tres carreteras desde Arad, una hacia Sibiu, una a Timisoara, y una a Zerind. Ninguna de éstas alcanza el objetivo, y, a menos que el agente esté familiarizado con la geografía de Rumanía, no sabrá qué carretera seguir². En otras palabras, el agente no sabrá cuál de las posibles acciones es mejor, porque no conoce lo suficiente los estados que resultan al tomar cada acción. Si el agente no tiene conocimiento adicional, entonces estará en un callejón sin salida. Lo mejor que puede hacer es escoger al azar una de las acciones.

Pero, supongamos que el agente tiene un mapa de Rumanía, en papel o en su memoria. El propósito del mapa es dotar al agente de información sobre los estados en los que podría encontrarse, así como las acciones que puede tomar. El agente puede usar esta información para considerar los siguientes estados de un viaje hipotético por cada una de las tres ciudades, intentando encontrar un viaje que llegue a Bucarest. Una vez que

¹ Observe que cada uno de estos «estados» se corresponde realmente a un conjunto de estados del mundo, porque un estado del mundo real especifica todos los aspectos de realidad. Es importante mantener en mente la distinción entre estados del problema a resolver y los estados del mundo.

² Suponemos que la mayoría de los lectores están en la misma situación y pueden fácilmente imaginarse cómo de desorientado está nuestro agente. Pedimos disculpas a los lectores rumanos quienes no pueden aprovecharse de este recurso pedagógico.



BÚSQUEDA

SOLUCIÓN

EJECUCIÓN

ha encontrado un camino en el mapa desde Arad a Bucarest, puede alcanzar su objetivo, tomando las acciones de conducir que corresponden con los tramos del viaje. En general, *un agente con distintas opciones inmediatas de valores desconocidos puede decidir qué hacer, examinando las diferentes secuencias posibles de acciones que le conduzcan a estados de valores conocidos, y entonces escoger la mejor secuencia.*

Este proceso de hallar esta secuencia se llama **búsqueda**. Un algoritmo de búsqueda toma como entrada un problema y devuelve una **solución** de la forma secuencia de acciones. Una vez que encontramos una solución, se procede a ejecutar las acciones que ésta recomienda. Esta es la llamada fase de **ejecución**. Así, tenemos un diseño simple de un agente «formular, buscar, ejecutar», como se muestra en la Figura 3.1. Después de formular un objetivo y un problema a resolver, el agente llama al procedimiento de búsqueda para resolverlo. Entonces, usa la solución para guiar sus acciones, haciendo lo que la solución le indica como siguiente paso a hacer —generalmente, primera acción de la secuencia— y procede a eliminar este paso de la secuencia. Una vez ejecutada la solución, el agente formula un nuevo objetivo.

Primero describimos el proceso de formulación del problema, y después dedicaremos la última parte del capítulo a diversos algoritmos para la función BÚSQUEDA. En este capítulo no discutiremos las funciones ACTUALIZAR-ESTADO y FORMULAR-OBJETIVO.

Antes de entrar en detalles, hagamos una breve pausa para ver dónde encajan los agentes resolventes de problemas en la discusión de agentes y entornos del Capítulo 2. El agente diseñado en la Figura 3.1 supone que el entorno es **estático**, porque la formulación y búsqueda del problema se hace sin prestar atención a cualquier cambio que puede ocurrir en el entorno. El agente diseñado también supone que se conoce el estado inicial; conocerlo es fácil si el entorno es **observable**. La idea de enumerar «las lí-

función AGENTE-SENCILLO-RESOLVENTE-PROBLEMAS(*percepción*) **devuelve** una acción

entradas: *percepción*, una percepción

estático: *sec*, una secuencia de acciones, vacía inicialmente
estado, una descripción del estado actual del mundo
objetivo, un objetivo, inicialmente nulo
problema, una formulación del problema

estado \leftarrow ACTUALIZAR-ESTADO(*estado*, *percepción*)

si *sec* está vacía **entonces** **hacer**

objetivo \leftarrow FORMULAR-OBJETIVO(*estado*)

problema \leftarrow FORMULAR-PROBLEMA(*estado*, *objetivo*)

sec \leftarrow BÚSQUEDA(*problema*)

acción \leftarrow PRIMERO(*secuencia*)

sec \leftarrow RESTO(*secuencia*)

devolver *acción*

Figura 3.1 Un sencillo agente resolvente de problemas. Primero formula un objetivo y un problema, busca una secuencia de acciones que deberían resolver el problema, y entonces ejecuta las acciones una cada vez. Cuando se ha completado, formula otro objetivo y comienza de nuevo. Notemos que cuando se ejecuta la secuencia, se ignoran sus percepciones: se supone que la solución encontrada trabajará bien.

LAZO ABIERTO

neas de acción alternativas» supone que el entorno puede verse como **discreto**. Finalmente, y más importante, el agente diseñado supone que el entorno es **determinista**. Las soluciones a los problemas son simples secuencias de acciones, así que no pueden manejar ningún acontecimiento inesperado; además, las soluciones se ejecutan sin prestar atención a las percepciones. Los agentes que realizan sus planes con los ojos cerrados, por así decirlo, deben estar absolutamente seguros de lo que pasa (los teóricos de control lo llaman sistema de **lazo abierto**, porque ignorar las percepciones rompe el lazo entre el agente y el entorno). Todas estas suposiciones significan que tratamos con las clases más fáciles de entornos, razón por la que este capítulo aparece tan pronto en el libro. La Sección 3.6 echa una breve ojeada sobre lo que sucede cuando relajamos las suposiciones de observancia y de determinismo. Los Capítulos 12 y 17 entran más en profundidad.

Problemas y soluciones bien definidos

PROBLEMA

Un **problema** puede definirse, formalmente, por cuatro componentes:

ESTADO INICIAL

- El **estado inicial** en el que comienza el agente. Por ejemplo, el estado inicial para nuestro agente en Rumanía se describe como *En(Arad)*.

FUCIÓN SUCESOR

- Una descripción de las posibles **acciones** disponibles por el agente. La formulación³ más común utiliza una **función sucesor**. Dado un estado particular *x*, *SUCESOR-FN(x)* devuelve un conjunto de pares ordenados *<acción, sucesor>*, donde cada acción es una de las acciones legales en el estado *x* y cada sucesor es un estado que puede alcanzarse desde *x*, aplicando la acción. Por ejemplo, desde el estado *En(Arad)*, la función sucesor para el problema de Rumanía devolverá

$$\{ \langle Ir(Sibiu), En(Sibiu) \rangle, \langle Ir(Timisoara), En(Timisoara) \rangle, \langle Ir(Zerind), En(Zerind) \rangle \}$$

ESPACIO DE ESTADOS

Implícitamente el estado inicial y la función sucesor definen el **espacio de estados** del problema (el conjunto de todos los estados alcanzables desde el estado inicial). El espacio de estados forma un grafo en el cual los nodos son estados y los arcos entre los nodos son acciones. (El mapa de Rumanía que se muestra en la Figura 3.2 puede interpretarse como un grafo del espacio de estados si vemos cada carretera como dos acciones de conducir, una en cada dirección). Un **camino** en el espacio de estados es una secuencia de estados conectados por una secuencia de acciones.

CAMINO

TEST OBJETIVO

- El **test objetivo**, el cual determina si un estado es un estado objetivo. Algunas veces existe un conjunto explícito de posibles estados objetivo, y el test simplemente comprueba si el estado es uno de ellos. El objetivo del agente en Rumanía es el conjunto *{En(Bucarest)}*. Algunas veces el objetivo se especifica como una propiedad abstracta más que como un conjunto de estados enumerados explícitamente. Por ejemplo, en el ajedrez, el objetivo es alcanzar un estado llamado «jaque mate», donde el rey del oponente es atacado y no tiene escapatoria.

³ Una formulación alternativa utiliza un conjunto de **operadores** que pueden aplicarse a un estado para generar así los sucesores.

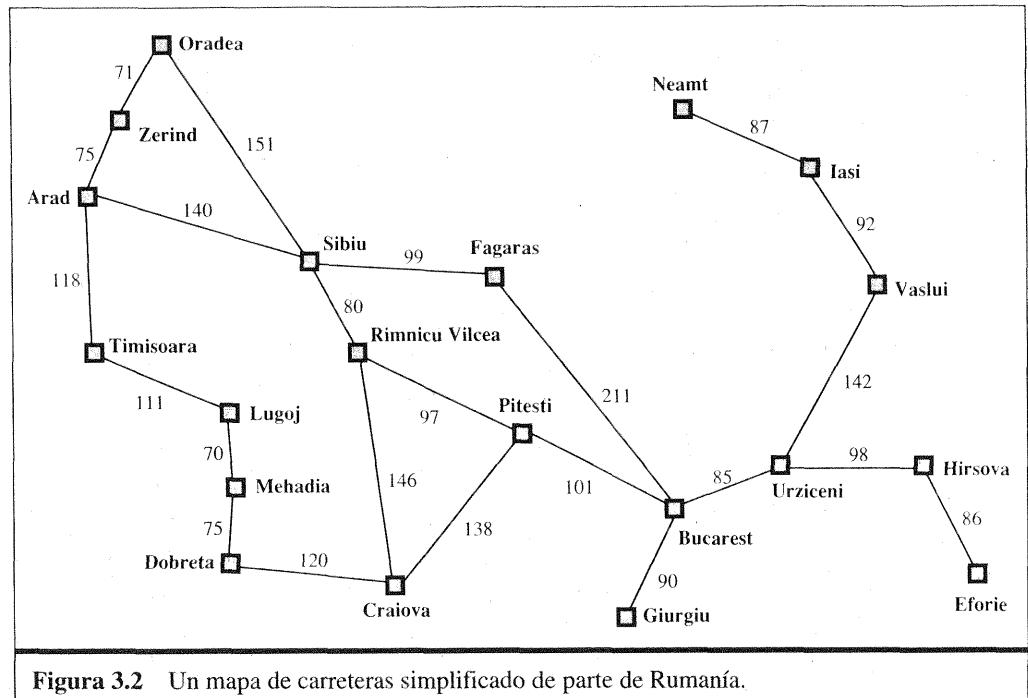


Figura 3.2 Un mapa de carreteras simplificado de parte de Rumanía.

COSTO DEL CAMINO

- Una función **costo del camino** que asigna un costo numérico a cada camino. El agente resolvente de problemas elige una función costo que refleje nuestra medida de rendimiento. Para el agente que intenta llegar a Bucarest, el tiempo es esencial, así que el costo del camino puede describirse como su longitud en kilómetros. En este capítulo, suponemos que el costo del camino puede describirse como la suma de los costos de las acciones individuales a lo largo del camino. El **costo individual** de una acción a que va desde un estado x al estado y se denota por $c(x,a,y)$. Los costos individuales para Rumanía se muestran en la Figura 3.2 como las distancias de las carreteras. Suponemos que los costos son no negativos⁴.

COSTO INDIVIDUAL

SOLUCIÓN ÓPTIMA

Los elementos anteriores definen un problema y pueden unirse en una estructura de datos simple que se dará como entrada al algoritmo resolvente del problema. Una **solución** de un problema es un camino desde el estado inicial a un estado objetivo. La calidad de la solución se mide por la función costo del camino, y una **solución óptima** tiene el costo más pequeño del camino entre todas las soluciones.

Formular los problemas

En la sección anterior propusimos una formulación del problema de ir a Bucarest en términos de estado inicial, función sucesor, test objetivo y costo del camino. Esta formulación parece razonable, a pesar de omitir muchos aspectos del mundo real. Para

⁴ Las implicaciones de costos negativos se exploran en el Ejercicio 3.17.

ABSTRACCIÓN

comparar la descripción de un estado simple, hemos escogido, *En(Arad)*, para un viaje real por el país, donde el estado del mundo incluye muchas cosas: los compañeros de viaje, lo que está en la radio, el paisaje, si hay algunos policías cerca, cómo está de lejos la parada siguiente, el estado de la carretera, el tiempo, etcétera. Todas estas consideraciones se dejan fuera de nuestras descripciones del estado porque son irrelevantes para el problema de encontrar una ruta a Bucarest. Al proceso de eliminar detalles de una representación se le llama **abstracción**.

Además de abstraer la descripción del estado, debemos abstraer sus acciones. Una acción de conducir tiene muchos efectos. Además de cambiar la localización del vehículo y de sus ocupantes, pasa el tiempo, consume combustible, genera contaminación, y cambia el agente (como dicen, el viaje ilustra). En nuestra formulación, tenemos en cuenta solamente el cambio en la localización. También, hay muchas acciones que omitiremos: encender la radio, mirar por la ventana, el retraso de los policías, etcétera. Y por supuesto, no especificamos acciones a nivel de «girar la rueda tres grados a la izquierda».

¿Podemos ser más precisos para definir los niveles apropiados de abstracción? Piense en los estados y las acciones abstractas que hemos elegido y que se corresponden con grandes conjuntos de estados detallados del mundo y de secuencias detalladas de acciones. Ahora considere una solución al problema abstracto: por ejemplo, la trayectoria de Arad a Sibiu a Rimnicu Vilcea a Pitesti a Bucarest. Esta solución abstracta corresponde a una gran cantidad de trayectorias más detalladas. Por ejemplo, podríamos conducir con la radio encendida entre Sibiu y Rimnicu Vilcea, y después lo apagamos para el resto del viaje. La abstracción es *válida* si podemos ampliar cualquier solución abstracta a una solución en el mundo más detallado; una condición suficiente es que para cada estado detallado de «en Arad», haya una trayectoria detallada a algún estado «en Sibiu», etcétera. La abstracción es útil si al realizar cada una de las acciones en la solución es más fácil que en el problema original; en este caso pueden ser realizadas por un agente que conduce sin búsqueda o planificación adicional. La elección de una buena abstracción implica quitar tantos detalles como sean posibles mientras que se conserve la validez y se asegure que las acciones abstractas son fáciles de realizar. Si no fuera por la capacidad de construir abstracciones útiles, los agentes inteligentes quedarían totalmente absorbidos por el mundo real.

3.2 Ejemplos de problemas

PROBLEMA DE JUGUETE

PROBLEMA DEL MUNDO REAL

La metodología para resolver problemas se ha aplicado a un conjunto amplio de entornos. Enumeramos aquí algunos de los más conocidos, distinguiendo entre problemas de *juguete* y del *mundo-real*. Un **problema de juguete** se utiliza para ilustrar o ejercitarse los métodos de resolución de problemas. Éstos se pueden describir de forma exacta y concisa. Esto significa que diferentes investigadores pueden utilizarlos fácilmente para comparar el funcionamiento de los algoritmos. Un **problema del mundo-real** es aquel en el que la gente se preocupa por sus soluciones. Ellos tienden a no tener una sola descripción, pero nosotros intentaremos dar la forma general de sus formulaciones.

Problemas de juguete

El primer ejemplo que examinaremos es el **mundo de la aspiradora**, introducido en el Capítulo 2. (Véase Figura 2.2.) Éste puede formularse como sigue:

- **Estados:** el agente está en una de dos localizaciones, cada una de las cuales puede o no contener suciedad. Así, hay $2 \times 2^2 = 8$ posibles estados del mundo.
 - **Estado inicial:** cualquier estado puede designarse como un estado inicial.
 - **Función sucesor:** ésta genera los estados legales que resultan al intentar las tres acciones (*Izquierda*, *Derecha* y *Aspirar*). En la Figura 3.3 se muestra el espacio de estados completo.
 - **Test objetivo:** comprueba si todos los cuadrados están limpios.
 - **Costo del camino:** cada costo individual es 1, así que el costo del camino es el número de pasos que lo compone.

Comparado con el mundo real, este problema de juguete tiene localizaciones discretas, suciedad discreta, limpieza fiable, y nunca se ensucia una vez que se ha limpiado. (En la Sección 3.6 relajaremos estas suposiciones). Una cosa a tener en cuenta es que el estado está determinado por la localización del agente y por las localizaciones de la suciedad. Un entorno grande con n localizaciones tiene $n \cdot 2^n$ estados.

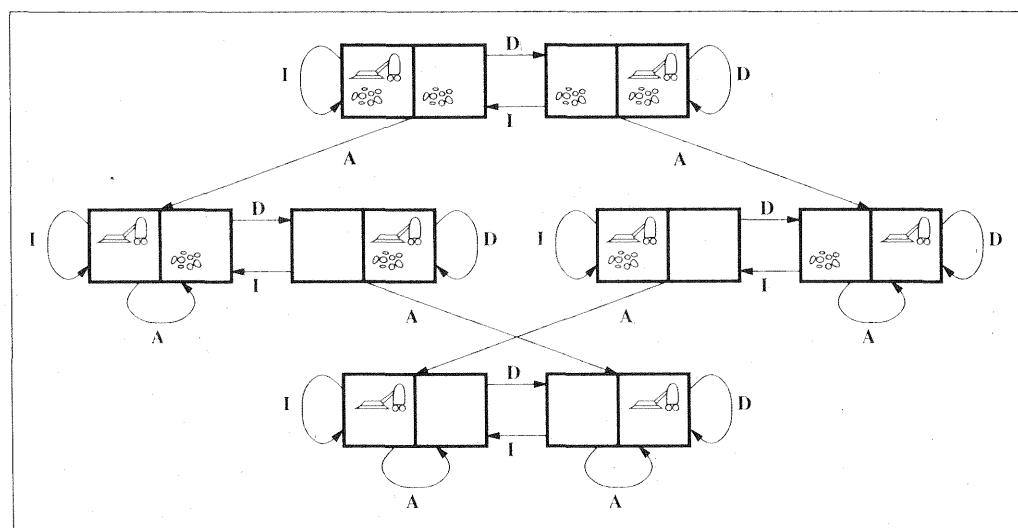


Figura 3.3 Espacio de estados para el mundo de la aspiradora. Los arcos denotan las acciones: I = Izquierda, D = Derecha, A = Aspirar.

8-PUZZLE

El **8-puzzle**, la Figura 3.4 muestra un ejemplo, consiste en un tablero de 3×3 con ocho fichas numeradas y un espacio en blanco. Una ficha adyacente al espacio en blanco puede deslizarse a éste. La meta es alcanzar el estado objetivo especificado, tal como se muestra a la derecha de la figura. La formulación estándar es como sigue:

- **Estados:** la descripción de un estado especifica la localización de cada una de las ocho fichas y el blanco en cada uno de los nueve cuadrados.

- **Estado inicial:** cualquier estado puede ser un estado inicial. Nótese que cualquier objetivo puede alcanzarse desde exactamente la mitad de los estados iniciales posibles (Ejercicio 3.4).
- **Función sucesor:** ésta genera los estados legales que resultan de aplicar las cuatro acciones (mover el blanco a la *Izquierda*, *Derecha*, *Arriba* y *Abajo*).
- **Test objetivo:** comprueba si el estado coincide con la configuración objetivo que se muestra en la Figura 3.4. (son posibles otras configuraciones objetivo).
- **Costo del camino:** el costo de cada paso del camino tiene valor 1, así que el costo del camino es el número de pasos.

¿Qué abstracciones se han incluido? Las acciones se han abstraído a los estados iniciales y finales, ignorando las localizaciones intermedias en donde se deslizan los bloques. Hemos abstraído acciones como la de sacudir el tablero cuando las piezas no se pueden mover, o extraer las piezas con un cuchillo y volverlas a poner. Nos dejan con una descripción de las reglas del puzzle que evitan todos los detalles de manipulaciones físicas.

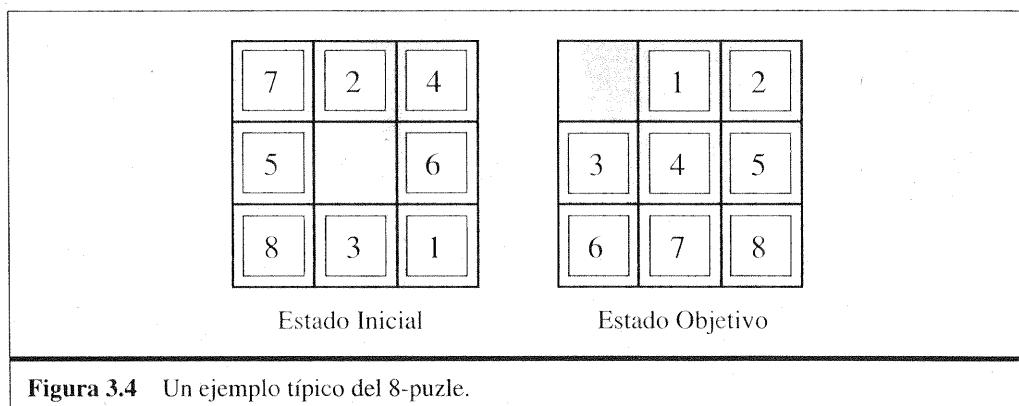


Figura 3.4 Un ejemplo típico del 8-puzzle.

PIEZAS DESLIZANTES

El 8-puzzle pertenece a la familia de puzzles con **piezas deslizantes**, los cuales a menudo se usan como problemas test para los nuevos algoritmos de IA. Esta clase general se conoce por ser NP completa, así que no esperamos encontrar métodos perceptiblemente mejores (en el caso peor) que los algoritmos de búsqueda descritos en este capítulo y en el siguiente. El 8-puzzle tiene $9!/2 = 181,440$ estados alcanzables y se resuelve fácilmente. El 15 puzzle (sobre un tablero de 4×4) tiene alrededor de 1,3 trillones de estados, y configuraciones aleatorias pueden resolverse óptimamente en pocos milisegundos por los mejores algoritmos de búsqueda. El 24 puzzle (sobre un tablero de 5×5) tiene alrededor de 10^{25} estados, y configuraciones aleatorias siguen siendo absolutamente difíciles de resolver de manera óptima con los computadores y algoritmos actuales.

PROBLEMA 8-REINAS

El objetivo del **problema de las 8-reinas** es colocar las ocho reinas en un tablero de ajedrez de manera que cada reina no ataque a ninguna otra. (Una reina ataca alguna pieza si está en la misma fila, columna o diagonal.) La Figura 3.5 muestra una configuración que no es solución: la reina en la columna de más a la derecha está atacando a la reina de arriba a la izquierda.

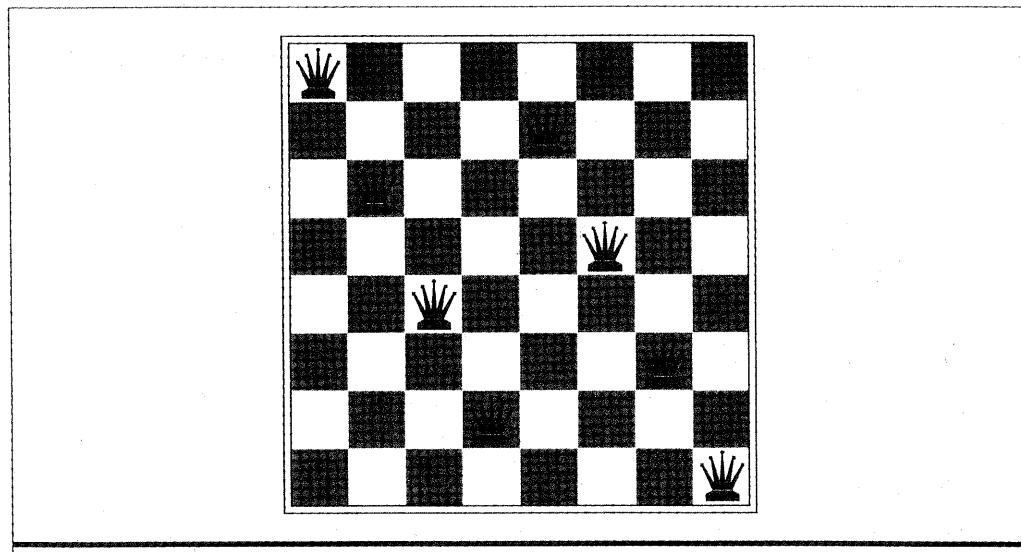


Figura 3.5 Casi una solución del problema de las 8-reinas. (La solución se deja como ejercicio.)

FORMULACIÓN INCREMENTAL

FORMULACIÓN COMPLETA DE ESTADOS

Aunque existen algoritmos eficientes específicos para este problema y para el problema general de las n reinas, sigue siendo un problema test interesante para los algoritmos de búsqueda. Existen dos principales formulaciones. Una **formulación incremental** que implica a operadores que aumentan la descripción del estado, comenzando con un estado vacío; para el problema de las 8-reinas, esto significa que cada acción añade una reina al estado. Una **formulación completa de estados** comienza con las ocho reinas en el tablero y las mueve. En cualquier caso, el coste del camino no tiene ningún interés porque solamente cuenta el estado final. La primera formulación incremental que se puede intentar es la siguiente:

- **Estados:** cualquier combinación de cero a ocho reinas en el tablero es un estado.
- **Estado inicial:** ninguna reina sobre el tablero.
- **Función sucesor:** añadir una reina a cualquier cuadrado vacío.
- **Test objetivo:** ocho reinas sobre el tablero, ninguna es atacada.

En esta formulación, tenemos $64 \cdot 63 \cdots 57 \approx 3 \times 10^{14}$ posibles combinaciones a investigar. Una mejor formulación deberá prohibir colocar una reina en cualquier cuadrado que esté realmente atacado:

- **Estados:** son estados, la combinación de n reinas ($0 \leq n \leq 8$), una por columna desde la columna más a la izquierda, sin que una reina ataque a otra.
- **Función sucesor:** añadir una reina en cualquier cuadrado en la columna más a la izquierda vacía tal que no sea atacada por cualquier otra reina.

Esta formulación reduce el espacio de estados de las 8-reinas de 3×10^{14} a 2.057, y las soluciones son fáciles de encontrar. Por otra parte, para 100 reinas la formulación inicial tiene 10^{400} estados mientras que las formulaciones mejoradas tienen cerca de 10^{52} estados (Ejercicio 3.5). Ésta es una reducción enorme, pero el espacio de estados mejorado sigue siendo demasiado grande para los algoritmos de este capítulo. El Capítulo 4

describe la formulación completa de estados y el Capítulo 5 nos da un algoritmo sencillo que hace el problema de un millón de reinas fácil de resolver.

Problemas del mundo real

PROBLEMA DE BÚSQUEDA DE UNA RUTA

Hemos visto cómo el **problema de búsqueda de una ruta** está definido en términos de posiciones y transiciones a lo largo de ellas. Los algoritmos de búsqueda de rutas se han utilizado en una variedad de aplicaciones, tales como rutas en redes de computadores, planificación de operaciones militares, y en sistemas de planificación de viajes de líneas aéreas. Estos problemas son complejos de especificar. Consideremos un ejemplo simplificado de un problema de viajes de líneas aéreas que especificamos como:

- **Estados:** cada estado está representado por una localización (por ejemplo, un aeropuerto) y la hora actual.
- **Estado inicial:** especificado por el problema.
- **Función sucesor:** devuelve los estados que resultan de tomar cualquier vuelo programado (quizá más especificado por la clase de asiento y su posición) desde el aeropuerto actual a otro, que salgan a la hora actual más el tiempo de tránsito del aeropuerto.
- **Test objetivo:** ¿tenemos nuestro destino para una cierta hora especificada?
- **Costo del camino:** esto depende del costo en dinero, tiempo de espera, tiempo del vuelo, costumbres y procedimientos de la inmigración, calidad del asiento, hora, tipo de avión, kilometraje del aviador experto, etcétera.

Los sistemas comerciales de viajes utilizan una formulación del problema de este tipo, con muchas complicaciones adicionales para manejar las estructuras bizantinas del precio que imponen las líneas aéreas. Cualquier viajero experto sabe, sin embargo, que no todo el transporte aéreo va según lo planificado. Un sistema realmente bueno debe incluir planes de contingencia (tales como reserva en vuelos alternativos) hasta el punto de que éstos estén justificados por el coste y la probabilidad de la falta del plan original.

PROBLEMAS TURÍSTICOS

Los **problemas turísticos** están estrechamente relacionados con los problemas de búsqueda de una ruta, pero con una importante diferencia. Consideremos, por ejemplo, el problema, «visitar cada ciudad de la Figura 3.2 al menos una vez, comenzando y finalizando en Bucarest». Como en la búsqueda de rutas, las acciones corresponden a viajes entre ciudades adyacentes. El espacio de estados, sin embargo, es absolutamente diferente. Cada estado debe incluir no sólo la localización actual sino también *las ciudades que el agente ha visitado*. El estado inicial sería «En Bucarest; visitado {Bucarest}», un estado intermedio típico sería «En Vaslui; visitado {Bucarest, Urziceni, Vaslui}», y el test objetivo comprobaría si el agente está en Bucarest y ha visitado las 20 ciudades.

PROBLEMA DEL VIAJANTE DE COMERCIO

El **problema del viajante de comercio** (PVC) es un problema de ruta en la que cada ciudad es visitada exactamente una vez. La tarea principal es encontrar el viaje *más corto*. El problema es de tipo NP duro, pero se ha hecho un gran esfuerzo para mejorar las capacidades de los algoritmos del PVC. Además de planificación de los viajes del viajante de comercio, estos algoritmos se han utilizado para tareas tales como la plani-

ficación de los movimientos de los taladros de un circuito impreso y para abastecer de máquinas a las tiendas.

DISTRIBUCIÓN VLSI

Un problema de **distribución VLSI** requiere la colocación de millones de componentes y de conexiones en un chip verificando que el área es mínima, que se reduce al mínimo el circuito, que se reduce al mínimo las capacitaciones, y se maximiza la producción de fabricación. El problema de la distribución viene después de la fase de diseño lógico, y está dividido generalmente en dos partes: **distribución de las celdas** y **dirección del canal**. En la distribución de las celdas, los componentes primitivos del circuito se agrupan en las celdas, cada una de las cuales realiza una cierta función. Cada celda tiene una característica fija (el tamaño y la forma) y requiere un cierto número de conexiones a cada una de las otras celdas. El objetivo principal es colocar las celdas en el chip de manera que no se superpongan y que quede espacio para que los alambres que conectan celdas puedan colocarse entre ellas. La dirección del canal encuentra una ruta específica para cada alambre por los espacios entre las celdas. Estos problemas de búsqueda son extremadamente complejos, pero definitivamente dignos de resolver. En el Capítulo 4, veremos algunos algoritmos capaces de resolverlos.

NAVEGACIÓN DE UN ROBOT

La **navegación de un robot** es una generalización del problema de encontrar una ruta descrito anteriormente. Más que un conjunto discreto de rutas, un robot puede moverse en un espacio continuo con (en principio) un conjunto infinito de acciones y estados posibles. Para un robot circular que se mueve en una superficie plana, el espacio es esencialmente de dos dimensiones. Cuando el robot tiene manos y piernas o ruedas que se deben controlar también, el espacio de búsqueda llega a ser de muchas dimensiones. Lo que se requiere es que las técnicas avanzadas hagan el espacio de búsqueda finito. Examinaremos algunos de estos métodos en el Capítulo 25. Además de la complejidad del problema, los robots reales también deben tratar con errores en las lecturas de los sensores y controles del motor.

SECUENCIACIÓN PARA EL ENSAMBLAJE AUTOMÁTICO

La **secuenciación para el ensamblaje automático** por un robot de objetos complejos fue demostrado inicialmente por FREDDY (Michie, 1972). Los progresos desde entonces han sido lentos pero seguros, hasta el punto de que el ensamblaje de objetos tales como motores eléctricos son económicamente factibles. En los problemas de ensamblaje, lo principal es encontrar un orden en los objetos a ensamblar. Si se elige un orden equivocado, no habrá forma de añadir posteriormente una parte de la secuencia sin deshacer el trabajo ya hecho. Verificar un paso para la viabilidad de la sucesión es un problema de búsqueda geométrico difícil muy relacionado con la navegación del robot. Así, la generación de sucesores legales es la parte costosa de la secuenciación para el ensamblaje. Cualquier algoritmo práctico debe evitar explorar todo, excepto una fracción pequeña del espacio de estados. Otro problema de ensamblaje importante es el **diseño de proteínas**, en el que el objetivo es encontrar una secuencia de aminoácidos que se plegarán en una proteína de tres dimensiones con las propiedades adecuadas para curar alguna enfermedad.

DISEÑO DE PROTEÍNAS

En la actualidad, se ha incrementado la demanda de robots *software* que realicen la **búsqueda en Internet**, la búsqueda de respuestas a preguntas, de información relacionada o para compras. Esto es una buena aplicación para las técnicas de búsqueda, porque es fácil concebir Internet como un grafo de nodos (páginas) conectadas por arcos. Una descripción completa de búsqueda en Internet se realiza en el Capítulo 10.

BÚSQUEDA EN INTERNET

3.3 Búsqueda de soluciones

ÁRBOL DE BÚSQUEDA

NODO DE BÚSQUEDA

EXPANDIR

GENERAR

ESTRATEGIA DE BÚSQUEDA

Hemos formulado algunos problemas, ahora necesitamos resolverlos. Esto se hace mediante búsqueda a través del espacio de estados. Este capítulo se ocupa de las técnicas de búsqueda que utilizan un **árbol de búsqueda** explícito generado por el estado inicial y la función sucesor, definiendo así el espacio de estados. En general, podemos tener un grafo de búsqueda más que un árbol, cuando el mismo estado puede alcanzarse desde varios caminos. Aplazamos, hasta la Sección 3.5, el tratar estas complicaciones importantes.

La Figura 3.6 muestra algunas de las expansiones en el árbol de búsqueda para encontrar una camino desde Arad a Bucarest. La raíz del árbol de búsqueda es el **nodo de búsqueda** que corresponde al estado inicial, *En(Arad)*. El primer paso es comprobar si éste es un estado objetivo. Claramente es que no, pero es importante comprobarlo de modo que podamos resolver problemas como «comenzar en Arad, consigue Arad». Como no estamos en un estado objetivo, tenemos que considerar otros estados. Esto se hace **expandiendo** el estado actual; es decir aplicando la función sucesor al estado actual y **generar** así un nuevo conjunto de estados. En este caso, conseguimos tres nuevos estados: *En(Sibiu)*, *En(Timisoara)* y *En(Zerind)*. Ahora debemos escoger cuál de estas tres posibilidades podemos considerar.

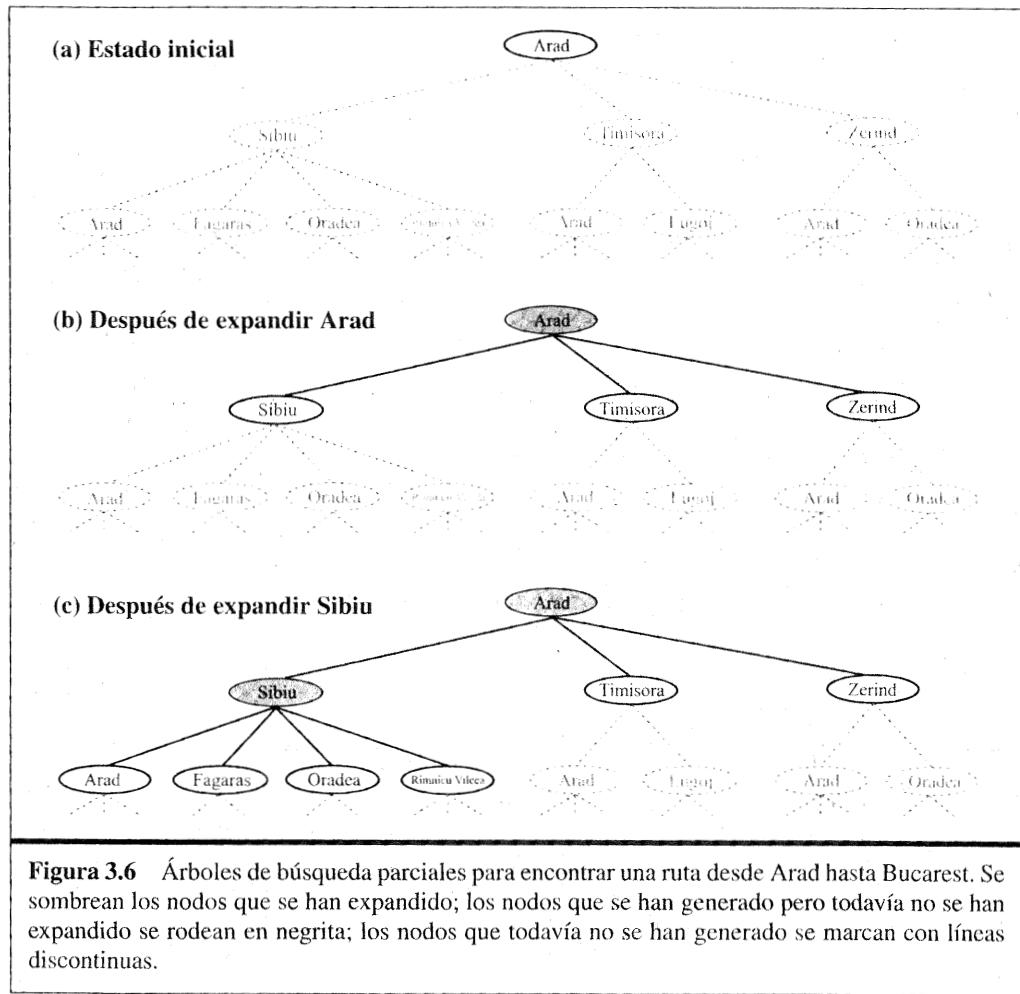
Esto es la esencia de la búsqueda, llevamos a cabo una opción y dejamos de lado las demás para más tarde, en caso de que la primera opción no conduzca a una solución. Supongamos que primero elegimos Sibiu. Comprobamos si es un estado objetivo (que no lo es) y entonces expandimos para conseguir *En(Arad)*, *En(Fagaras)*, *En(Oradea)* y *En(Rimnicu Vilcea)*. Entonces podemos escoger cualquiera de estas cuatro o volver atrás y escoger Timisoara o Zerind. Continuamos escogiendo, comprobando y expandiendo hasta que se encuentra una solución o no existen más estados para expandir. El estado a expandir está determinado por la **estrategia de búsqueda**. La Figura 3.7 describe informalmente el algoritmo general de búsqueda en árboles.

Es importante distinguir entre el espacio de estados y el árbol de búsqueda. Para el problema de búsqueda de un ruta, hay solamente 20 estados en el espacio de estados, uno por cada ciudad. Pero hay un número infinito de caminos en este espacio de estados, así que el árbol de búsqueda tiene un número infinito de nodos. Por ejemplo, los tres caminos Arad-Sibiu, Arad-Sibiu-Arad, Arad-Sibiu-Arad-Sibiu son los tres primeros caminos de una secuencia infinita de caminos. (Obviamente, un buen algoritmo de búsqueda evita seguir tales trayectorias; La Sección 3.5 nos muestra cómo hacerlo).

Hay muchas formas de representar los nodos, pero vamos a suponer que un nodo es una estructura de datos con cinco componentes:

- ESTADO: el estado, del espacio de estados, que corresponde con el nodo;
- NODO PADRE: el nodo en el árbol de búsqueda que ha generado este nodo;
- ACCIÓN: la acción que se aplicará al padre para generar el nodo;
- COSTO DEL CAMINO: el costo, tradicionalmente denotado por $g(n)$, de un camino desde el estado inicial al nodo, indicado por los punteros a los padres; y
- PROFUNDIDAD: el número de pasos a lo largo del camino desde el estado inicial.

Es importante recordar la distinción entre nodos y estados. Un nodo es una estructura de datos usada para representar el árbol de búsqueda. Un estado corresponde a una



función BÚSQUEDA-ÁRBOLES(*problema, estrategia*) **devuelve** una solución o fallo
 initializa el árbol de búsqueda usando el estado inicial del *problema*
 bucle hacer

si no hay candidatos para expandir **entonces devolver** fallo
 escoger, de acuerdo a la *estrategia*, un nodo hoja para expandir
 si el nodo contiene un estado objetivo **entonces devolver** la correspondiente solución
 en otro caso expandir el nodo y añadir los nodos resultado al árbol de búsqueda

Figura 3.7 Descripción informal del algoritmo general de búsqueda en árboles.

configuración del mundo. Así, los nodos están en caminos particulares, según lo definido por los punteros del nodo padre, mientras que los estados no lo están. En la Figura 3.8 se representa la estructura de datos del nodo.

También necesitamos representar la colección de nodos que se han generado pero todavía no se han expandido – a esta colección se le llama **frontera**. Cada elemento de

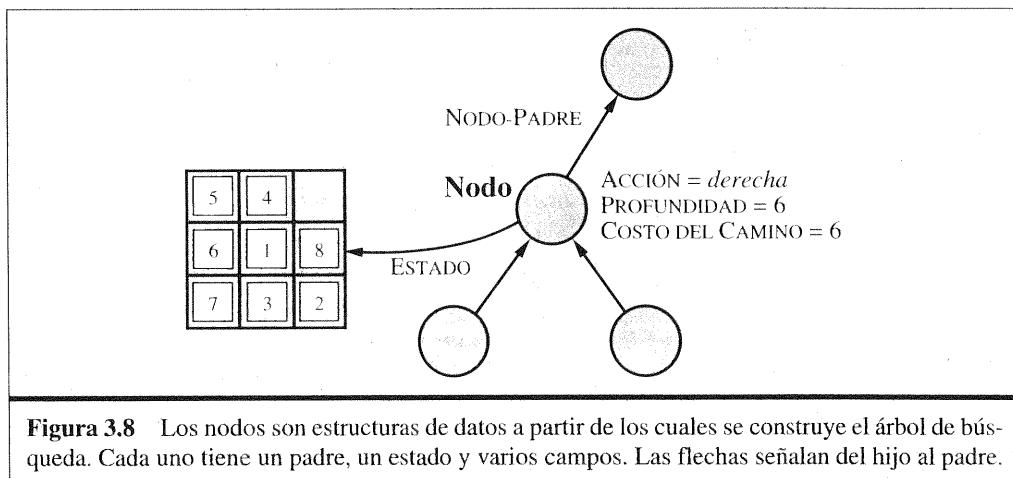


Figura 3.8 Los nodos son estructuras de datos a partir de los cuales se construye el árbol de búsqueda. Cada uno tiene un parente, un estado y varios campos. Las flechas señalan del hijo al parente.

NODO HOJA

la frontera es un **nodo hoja**, es decir, un nodo sin sucesores en el árbol. En la Figura 3.6, la frontera de cada árbol consiste en los nodos dibujados con líneas discontinuas. La representación más simple de la frontera sería como un conjunto de nodos. La estrategia de búsqueda será una función que seleccione de este conjunto el siguiente nodo a expandir. Aunque esto sea conceptualmente sencillo, podría ser computacionalmente costoso, porque la función estrategia quizás tenga que mirar cada elemento del conjunto para escoger el mejor. Por lo tanto, nosotros asumiremos que la colección de nodos se implementa como una **cola**. Las operaciones en una cola son como siguen:

- HACER-COLA(*elemento*, ...) crea una cola con el(s) elemento(s) dado(s).
- VACIA?(*cola*) devuelve verdadero si no hay ningún elemento en la cola.
- PRIMERO(*cola*) devuelve el primer elemento de la cola.
- BORRAR-PRIMERO(*cola*) devuelve PRIMERO(*cola*) y lo borra de la cola.
- INSERTA(*elemento*, *cola*) inserta un elemento en la cola y devuelve la cola resultado. (Veremos que tipos diferentes de colas insertan los elementos en órdenes diferentes.)
- INSERTAR-TODO(*elementos*, *cola*) inserta un conjunto de elementos en la cola y devuelve la cola resultado.

Con estas definiciones, podemos escribir una versión más formal del algoritmo general de búsqueda en árboles. Se muestra en la Figura 3.9.

Medir el rendimiento de la resolución del problema

La salida del algoritmo de resolución de problemas es *fallo* o una solución. (Algunos algoritmos podrían caer en un bucle infinito y nunca devolver una salida.) Evaluaremos el rendimiento de un algoritmo de cuatro formas:

COMPLETITUD

- **Completitud:** ¿está garantizado que el algoritmo encuentre una solución cuando esta existe?
- **Optimización:** ¿encuentra la estrategia la solución óptima, según lo definido en la página 62?

OPTIMIZACIÓN

función BÚSQUEDA-ÁRBOLES(*problema,frontera*) **devuelve** una solución o fallo

frontera \leftarrow INSERTA(HACER-NODO(ESTADO-INICIAL[*problema*]), *frontera*)

hacer bucle

si VACIA?(*frontera*) **entonces devolver** fallo.

nodo \leftarrow BORRAR-PRIMERO(*frontera*)

si TEST-OBJETIVO[*problema*] aplicado al ESTADO[*nodo*] es cierto

entonces devolver SOLUCIÓN(*nodo*)

frontera \leftarrow INSERTAR-TODO(EXPANDIR(*nodo,problema*), *frontera*)

función EXPANDIR(*nodo,problema*) **devuelve** un conjunto de nodos

sucesores \leftarrow conjunto vacío

para cada (*acción,resultado*) **en** SUCESOR-FN[*problema*](ESTADO[*nodo*]) **hacer**

s \leftarrow un nuevo Nodo

 ESTADO[*s*] \leftarrow *resultado*

 NODO-PADRE[*s*] \leftarrow *nodo*

 ACCIÓN[*s*] \leftarrow *acción*

 COSTO-CAMINO[*s*] \leftarrow COSTO-CAMINO[*nodo*] + COSTO-INDIVIDUAL(*nodo,acción,s*)

 PROFUNDIDAD[*s*] \leftarrow PROFUNDIDAD[*nodo*] + 1

 añadir *s* a *sucesores*

devolver *sucesores*

Figura 3.9 Algoritmo general de búsqueda en árboles. (Notemos que el argumento *frontera* puede ser una cola vacía, y el tipo de cola afectará al orden de la búsqueda.) La función SOLUCIÓN devuelve la secuencia de acciones obtenida de la forma punteros al padre hasta la raíz.

COMPLEJIDAD EN TIEMPO

COMPLEJIDAD EN ESPACIO

FACTOR DE RAMIFICACIÓN

COSTO DE LA BÚSQUEDA

• **Complejidad en tiempo:** ¿cuánto tarda en encontrar una solución?

• **Complejidad en espacio:** ¿cuánta memoria se necesita para el funcionamiento de la búsqueda?

La complejidad en tiempo y espacio siempre se considera con respecto a alguna medida de la dificultad del problema. En informática teórica, la medida es el tamaño del grafo del espacio de estados, porque el grafo se ve como una estructura de datos explícita que se introduce al programa de búsqueda. (El mapa de Rumanía es un ejemplo de esto.) En IA, donde el grafo está representado de forma implícita por el estado inicial y la función sucesor y frecuentemente es infinito, la complejidad se expresa en términos de tres cantidades: *b*, el **factor de ramificación** o el máximo número de sucesores de cualquier nodo; *d*, la profundidad del nodo objetivo más superficial; y *m*, la longitud máxima de cualquier camino en el espacio de estados.

El tiempo a menudo se mide en términos de número de nodos generados⁵ durante la búsqueda, y el espacio en términos de máximo número de nodos que se almacena en memoria.

Para valorar la eficacia de un algoritmo de búsqueda, podemos considerar el **costo de la búsqueda** (que depende típicamente de la complejidad en tiempo pero puede in-

⁵ Algunos textos miden el tiempo en términos del número de las expansiones del nodo. Las dos medidas se diferencian como mucho en un factor *b*. A nosotros nos parece que el tiempo de ejecución de la expansión del nodo aumenta con el número de nodos generados en esa expansión.

COSTE TOTAL

cluir también un término para el uso de la memoria) o podemos utilizar el **coste total**, que combina el costo de la búsqueda y el costo del camino solución encontrado. Para el problema de encontrar una ruta desde Arad hasta Bucarest, el costo de la búsqueda es la cantidad de tiempo que ha necesitado la búsqueda y el costo de la solución es la longitud total en kilómetros del camino. Así, para el cálculo del coste total, tenemos que sumar kilómetros y milisegundos. No hay ninguna conversión entre los dos, pero quizás sea razonable, en este caso, convertir kilómetros en milisegundos utilizando una estimación de la velocidad media de un coche (debido a que el tiempo es lo que cuida el agente.) Esto permite al agente encontrar un punto óptimo de intercambio en el cual el cálculo adicional para encontrar que un camino más corto llegue a ser contraproducente. El problema más general de intercambios entre bienes diferentes será tratado en el Capítulo 16.

3.4 Estrategias de búsqueda no informada

BÚSQUEDA NO INFORMADA

Esta sección trata cinco estrategias de búsqueda englobadas bajo el nombre de **búsqueda no informada** (llamada también **búsqueda a ciegas**). El término significa que ellas no tienen información adicional acerca de los estados más allá de la que proporciona la definición del problema. Todo lo que ellas pueden hacer es generar los sucesores y distinguir entre un estado objetivo de uno que no lo es. Las estrategias que saben si un estado no objetivo es «más prometedor» que otro se llaman **búsqueda informada** o **búsqueda heurística**; éstas serán tratadas en el Capítulo 4. Todas las estrategias se distinguen por el *orden* de expansión de los nodos.

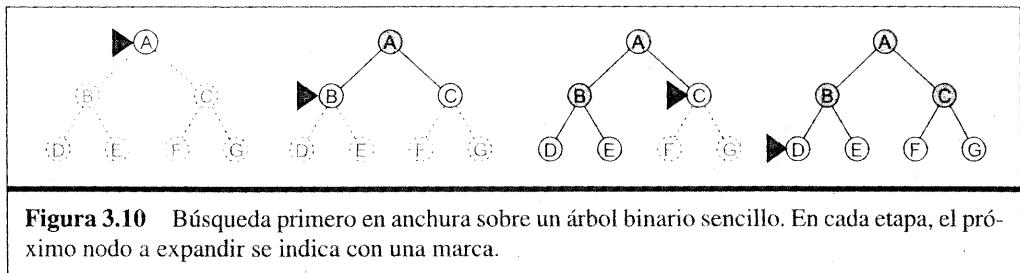
BÚSQUEDA INFORMADA**BÚSQUEDA HEURÍSTICA****BÚSQUEDA PRIMERO EN ANCHURA**

Búsqueda primero en anchura

La **búsqueda primero en anchura** es una estrategia sencilla en la que se expande primero el nodo raíz, a continuación se expanden todos los sucesores del nodo raíz, después sus sucesores, etc. En general, se expanden todos los nodos a una profundidad en el árbol de búsqueda antes de expandir cualquier nodo del próximo nivel.

La búsqueda primero en anchura se puede implementar llamando a la BÚSQUEDA-ÁRBOLES con una frontera vacía que sea una cola primero en entrar primero en salir (FIFO), asegurando que los nodos primeros visitados serán los primeros expandidos. En otras palabras, llamando a la BÚSQUEDA-ÁRBOLES(*problema*,COLA-FIFO()) resulta una búsqueda primero en anchura. La cola FIFO pone todos los nuevos sucesores generados al final de la cola, lo que significa que los nodos más superficiales se expanden antes que los nodos más profundos. La Figura 3.10 muestra el progreso de la búsqueda en un árbol binario sencillo.

Evaluemos la búsqueda primero en anchura usando los cuatro criterios de la sección anterior. Podemos ver fácilmente que es *completa* (si el nodo objetivo más superficial está en una cierta profundidad finita d , la búsqueda primero en anchura lo encontrará después de expandir todos los nodos más superficiales, con tal que el factor de ramificación b sea finito). El nodo objetivo más superficial no es necesariamente el óptimo:



técnicamente, la búsqueda primero en anchura es óptima si el costo del camino es una función no decreciente de la profundidad del nodo (por ejemplo, cuando todas las acciones tienen el mismo coste).

Hasta ahora, la información sobre la búsqueda primero en anchura ha sido buena. Para ver por qué no es siempre la estrategia a elegir, tenemos que considerar la cantidad de tiempo y memoria que utiliza para completar una búsqueda. Para hacer esto, consideramos un espacio de estados hipotético donde cada estado tiene b sucesores. La raíz del árbol de búsqueda genera b nodos en el primer nivel, cada uno de ellos genera b nodos más, teniendo un total de b^2 en el segundo nivel. Cada uno de estos genera b nodos más, teniendo b^3 nodos en el tercer nivel, etcétera. Ahora supongamos que la solución está a una profundidad d . En el peor caso, expandiremos todos excepto el último nodo en el nivel d (ya que el objetivo no se expande), generando $b^{d+1} - b$ nodos en el nivel $d + 1$. Entonces el número total de nodos generados es

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

Cada nodo generado debe permanecer en la memoria, porque o es parte de la frontera o es un antepasado de un nodo de la frontera. La complejidad en espacio es, por lo tanto, la misma que la complejidad en tiempo (más un nodo para la raíz).

Los que hacen análisis de complejidad están preocupados (o emocionados, si les gusta el desafío) por las cotas de complejidad exponencial como $O(b^{d+1})$. La Figura 3.11 muestra por qué. Se enumera el tiempo y la memoria requerida para una búsqueda primero en anchura con el factor de ramificación $b = 10$, para varios valores de profundidad.

Profundidad	Nodos	Tiempo	Memoria
2	1.100	11 segundos	1 megabyte
4	111.100	11 segundos	106 megabytes
6	10^7	19 minutos	10 gigabytes
8	10^9	31 horas	1 terabytes
10	10^{11}	129 días	101 terabytes
12	10^{13}	35 años	10 petabytes
14	10^{15}	3.523 años	1 exabyte

Figura 3.11 Requisitos de tiempo y espacio para la búsqueda primero en anchura. Los números que se muestran suponen un factor de ramificación $b = 10$; 10.000 nodos/segundo; 1.000 bytes/nodo.

dad d de la solución. La tabla supone que se pueden generar 10.000 nodos por segundo y que un nodo requiere 1.000 bytes para almacenarlo. Muchos problemas de búsqueda quedan aproximadamente dentro de estas suposiciones (más o menos un factor de 100) cuando se ejecuta en un computador personal moderno.

 Hay dos lecciones que debemos aprender de la Figura 3.11. Primero, *son un problema más grande los requisitos de memoria para la búsqueda primero en anchura que el tiempo de ejecución*. 31 horas no sería demasiado esperar para la solución de un problema importante a profundidad ocho, pero pocos computadores tienen suficientes terabytes de memoria principal que lo admitieran. Afortunadamente, hay otras estrategias de búsqueda que requieren menos memoria.

 La segunda lección es que los requisitos de tiempo son todavía un factor importante. Si su problema tiene una solución a profundidad 12, entonces (dadas nuestras suposiciones) llevará 35 años encontrarla por la búsqueda primero en anchura (o realmente alguna búsqueda sin información). En general, *los problemas de búsqueda de complejidad-exponencial no pueden resolverse por métodos sin información, salvo casos pequeños*.

Búsqueda de costo uniforme

BÚSQUEDA DE COSTO UNIFORME

La búsqueda primero en anchura es óptima cuando todos los costos son iguales, porque siempre expande el nodo no expandido más superficial. Con una extensión sencilla, podemos encontrar un algoritmo que es óptimo con cualquier función costo. En vez de expandir el nodo más superficial, la **búsqueda de costo uniforme** expande el nodo n con el camino de costo más pequeño. Notemos que si todos los costos son iguales, es idéntico a la búsqueda primero en anchura.

La búsqueda de costo uniforme no se preocupa por el número de pasos que tiene un camino, pero sí sobre su coste total. Por lo tanto, éste se meterá en un bucle infinito si expande un nodo que tiene una acción de coste cero que conduzca de nuevo al mismo estado (por ejemplo, una acción *NoOp*). Podemos garantizar completitud si el costo de cada paso es mayor o igual a alguna constante positiva pequeña ϵ . Esta condición es también suficiente para asegurar optimización. Significa que el costo de un camino siempre aumenta cuando vamos por él. De esta propiedad, es fácil ver que el algoritmo expande nodos que incrementan el coste del camino. Por lo tanto, el primer nodo objetivo seleccionado para la expansión es la solución óptima. (Recuerde que la búsqueda en árboles aplica el test objetivo sólo a los nodos que son seleccionados para la expansión.) Recomendamos probar el algoritmo para encontrar el camino más corto a Bucarest.

La búsqueda de costo uniforme está dirigida por los costos de los caminos más que por las profundidades, entonces su complejidad no puede ser fácilmente caracterizada en términos de b y d . En su lugar, C^* es el costo de la solución óptima, y se supone que cada acción cuesta al menos ϵ . Entonces la complejidad en tiempo y espacio del peor caso del algoritmo es $O(b^{[C^*/\epsilon]})$, la cual puede ser mucho mayor que b^d . Esto es porque la búsqueda de costo uniforme, y a menudo lo hace, explora los árboles grandes en pequeños pasos antes de explorar caminos que implican pasos grandes y quizás útiles. Cuando todos los costos son iguales, desde luego, la $b^{[C^*/\epsilon]}$ es justamente b^d .

Búsqueda primero en profundidad

BÚSQUEDA PRIMERO EN PROFUNDIDAD

La **búsqueda primero en profundidad** siempre expande el nodo *más profundo* en la frontera actual del árbol de búsqueda. El progreso de la búsqueda se ilustra en la Figura 3.12. La búsqueda procede inmediatamente al nivel más profundo del árbol de búsqueda, donde los nodos no tienen ningún sucesor. Cuando esos nodos se expanden, son quitados de la frontera, así entonces la búsqueda «retrocede» al siguiente nodo más superficial que todavía tenga sucesores inexplorados.

Esta estrategia puede implementarse por la **BÚSQUEDA-ÁRBOLES** con una cola último en entrar primero en salir (LIFO), también conocida como una pila. Como una alternativa a la implementación de la **BÚSQUEDA-ÁRBOLES**, es común aplicar la búsqueda primero en profundidad con una función recursiva que se llama en cada uno de sus hijos. (Un algoritmo primero en profundidad recursivo incorporando un límite de profundidad se muestra en la Figura 3.13.)

La búsqueda primero en profundidad tiene unos requisitos muy modestos de memoria. Necesita almacenar sólo un camino desde la raíz a un nodo hoja, junto con los nodos hermanos restantes no expandidos para cada nodo del camino. Una vez que un nodo se

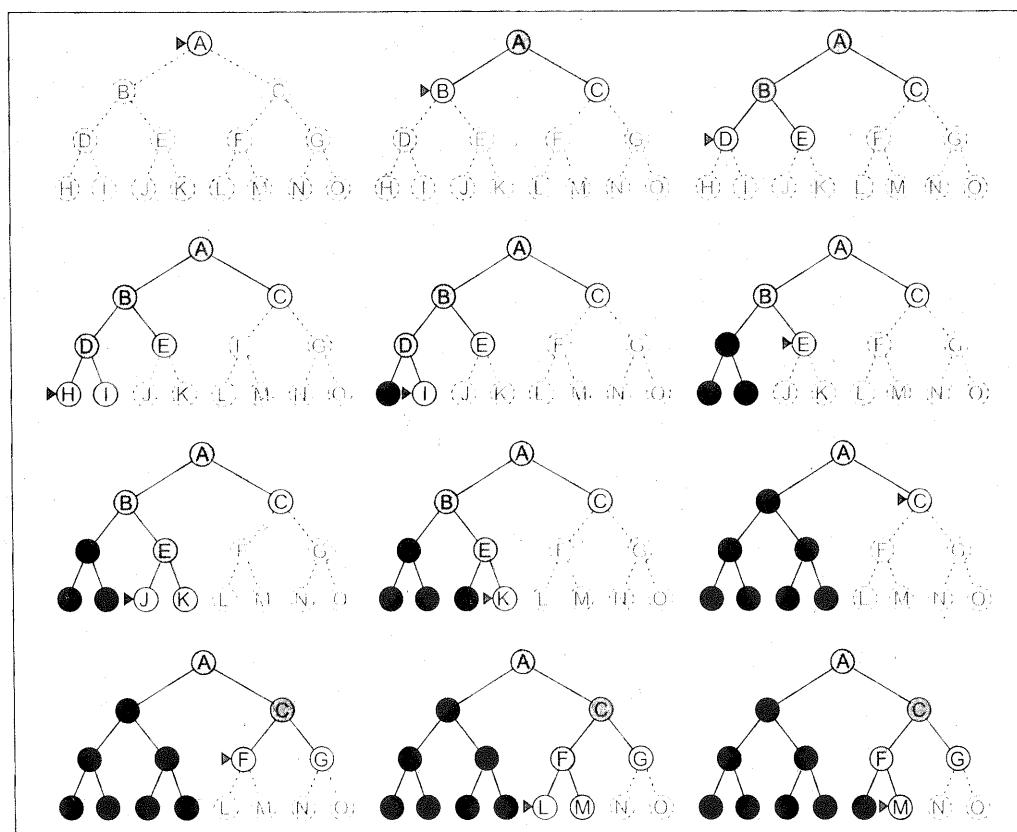


Figura 3.12 Búsqueda primero en profundidad sobre un árbol binario. Los nodos que se han expandido y no tienen descendientes en la frontera se pueden quitar de la memoria; estos se muestran en negro. Los nodos a profundidad 3 se suponen que no tienen sucesores y *M* es el nodo objetivo.

```

función BÚSQUEDA-PROFUNDIDAD-LIMITADA(problema,límite) devuelve una solución, o
    fallo/corte
devolver BPL-RECURSIVO(HACER-NODO(ESTADO-INICIAL[problema]).
    problema,límite)

función BPL-RECURSIVO(nodo,problema,límite) devuelve una solución, o fallo/corte
    ocurrió un corte  $\leftarrow$  falso
    si TEST-OBJETIVO[problema](ESTADO[nodo]) entonces devolver SOLUCIÓN(nodo)
    en caso contrario si PROFUNDIDAD[nodo] = límite entonces devolver corte
    en caso contrario para cada sucesor en EXPANDIR(nodo,problema) hacer
        resultado  $\leftarrow$  BPL-RECURSIVO(sucesor,problema,límite)
        si resultado = corte entonces ocurrió un corte  $\leftarrow$  verdadero
        en otro caso si resultado  $\neq$  fallo entonces devolver resultado
    si ocurrió un corte? entonces devolver corte en caso contrario devolver fallo

```

Figura 3.13 Implementación recursiva de la búsqueda primero en profundidad.

ha expandido, se puede quitar de la memoria tan pronto como todos sus descendientes han sido explorados. (Véase Figura 3.12.) Para un espacio de estados con factor de ramificación b y máxima profundidad m , la búsqueda primero en profundidad requiere almacenar sólo $bm + 1$ nodos. Utilizando las mismas suposiciones que con la Figura 3.11, y suponiendo que los nodos a la misma profundidad que el nodo objetivo no tienen ningún sucesor, nos encontramos que la búsqueda primero en profundidad requeriría 118 kilobytes en vez de diez petabytes a profundidad $d = 12$, un factor de diez billones de veces menos de espacio.

Una variante de la búsqueda primero en profundidad, llamada **búsqueda hacia atrás**, utiliza todavía menos memoria. En la búsqueda hacia atrás, sólo se genera un sucesor a la vez; cada nodo parcialmente expandido recuerda qué sucesor se expande a continuación. De esta manera, sólo se necesita $O(m)$ memoria más que el $O(bm)$ anterior. La búsqueda hacia atrás facilita aún otro ahorro de memoria (y ahorro de tiempo): la idea de generar un sucesor modificando directamente la descripción actual del estado más que copiarlo. Esto reduce los requerimientos de memoria a solamente una descripción del estado y $O(m)$ acciones. Para hacer esto, debemos ser capaces de deshacer cada modificación cuando volvemos hacia atrás para generar el siguiente sucesor. Para problemas con grandes descripciones de estados, como el ensamblaje en robótica, estas técnicas son críticas para tener éxito.

El inconveniente de la búsqueda primero en profundidad es que puede hacer una elección equivocada y obtener un camino muy largo (o infinito) aun cuando una elección diferente llevaría a una solución cerca de la raíz del árbol de búsqueda. Por ejemplo, en la Figura 3.12, la búsqueda primero en profundidad explorará el subárbol izquierdo entero incluso si el nodo *C* es un nodo objetivo. Si el nodo *J* fuera también un nodo objetivo, entonces la búsqueda primero en profundidad lo devolvería como una solución; de ahí, que la búsqueda primero en profundidad no es óptima. Si el subárbol izquierdo fuera de profundidad ilimitada y no contuviera ninguna solución, la búsqueda primero en profundidad nunca terminaría; de ahí, que no es completo. En el caso peor, la búsqueda primero en profundidad generará todos los nodos $O(b^m)$ del árbol de búsqueda, don-

de m es la profundidad máxima de cualquier nodo. Nótese que m puede ser mucho más grande que d (la profundidad de la solución más superficial), y es infinito si el árbol es ilimitado.

Búsqueda de profundidad limitada

Se puede aliviar el problema de árboles ilimitados aplicando la búsqueda primero en profundidad con un límite de profundidad ℓ predeterminado. Es decir, los nodos a profundidad ℓ se tratan como si no tuvieran ningún sucesor. A esta aproximación se le llama **búsqueda de profundidad limitada**. El límite de profundidad resuelve el problema del camino infinito. Lamentablemente, también introduce una fuente adicional de incompletitud si escogemos $\ell < d$, es decir, el objetivo está fuera del límite de profundidad. (Esto no es improbable cuando d es desconocido.) La búsqueda de profundidad limitada también será no óptima si escogemos $\ell > d$. Su complejidad en tiempo es $O(b^\ell)$ y su complejidad en espacio es $O(b\ell)$. La búsqueda primero en profundidad puede verse como un caso especial de búsqueda de profundidad limitada con $\ell = \infty$.

A veces, los límites de profundidad pueden estar basados en el conocimiento del problema. Por ejemplo, en el mapa de Rumanía hay 20 ciudades. Por lo tanto, sabemos que si hay una solución, debe ser de longitud 19 como mucho, entonces $\ell = 19$ es una opción posible. Pero de hecho si estudiáramos el mapa con cuidado, descubriríamos que cualquier ciudad puede alcanzarse desde otra como mucho en nueve pasos. Este número, conocido como el **diámetro** del espacio de estados, nos da un mejor límite de profundidad, que conduce a una búsqueda con profundidad limitada más eficiente. Para la mayor parte de problemas, sin embargo, no conoceremos un límite de profundidad bueno hasta que hayamos resuelto el problema.

La búsqueda de profundidad limitada puede implementarse con una simple modificación del algoritmo general de búsqueda en árboles o del algoritmo recursivo de búsqueda primero en profundidad. En la Figura 3.13 se muestra el pseudocódigo de la búsqueda recursiva de profundidad limitada. Notemos que la búsqueda de profundidad limitada puede terminar con dos clases de fracaso: el valor de *fracaso* estándar indicando que no hay ninguna solución; el valor de *corte* indicando que no hay solución dentro del límite de profundidad.

Búsqueda primero en profundidad con profundidad iterativa

La **búsqueda con profundidad iterativa** (o búsqueda primero en profundidad con profundidad iterativa) es una estrategia general, usada a menudo en combinación con la búsqueda primero en profundidad, la cual encuentra el mejor límite de profundidad. Esto se hace aumentando gradualmente el límite (primero 0, después 1, después 2, etcétera) hasta que encontramos un objetivo. Esto ocurrirá cuando el límite de profundidad alcanza d , profundidad del nodo objetivo. Se muestra en la Figura 3.14 el algoritmo. La profundidad iterativa combina las ventajas de la búsqueda primero en profundidad y primero en anchura. En la búsqueda primero en profundidad, sus exigencias de memoria

función BÚSQUEDA-PROFUNDIDAD-ITERATIVA(*problema*) **devuelve** una solución, o fallo
entradas: *problema*, un problema

```
para profundidad ← 0 a  $\infty$  hacer
    resultado ← BÚSQUEDA-PROFUNDIDAD-LIMITADA(problema, profundidad)
    si resultado ≠ corte entonces devolver resultado
```

Figura 3.14 Algoritmo de búsqueda de profundidad iterativa, el cual aplica repetidamente la búsqueda de profundidad limitada incrementando el límite. Termina cuando se encuentra una solución o si la búsqueda de profundidad limitada devuelve *fracaso*, significando que no existe solución.

son muy modestas: $O(bd)$ para ser exacto. La búsqueda primero en anchura, es completa cuando el factor de ramificación es finito y óptima cuando el coste del camino es una función que no disminuye con la profundidad del nodo. La Figura 3.15 muestra cuatro iteraciones de la BÚSQUEDA-PROFUNDIDAD-ITERATIVA sobre un árbol binario de búsqueda, donde la solución se encuentra en la cuarta iteración.

La búsqueda de profundidad iterativa puede parecer derrochadora, porque los estados se generan múltiples veces. Pero esto no es muy costoso. La razón es que en un árbol de búsqueda con el mismo (o casi el mismo) factor de ramificación en cada nivel, la mayor parte de los nodos está en el nivel inferior, entonces no importa mucho que los niveles superiores se generen múltiples veces. En una búsqueda de profundidad iterativa, los nodos sobre el nivel inferior (profundidad d) son generados una vez, los anteriores al nivel inferior son generados dos veces, etc., hasta los hijos de la raíz, que son generados d veces. Entonces el número total de nodos generados es

$$N(BPI) = (d)b + (d - 1)b^2 + \dots + (1)b^d,$$

que da una complejidad en tiempo de $O(b^d)$. Podemos compararlo con los nodos generados por una búsqueda primero en anchura:

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

Notemos que la búsqueda primero en anchura genera algunos nodos en profundidad $d + 1$, mientras que la profundidad iterativa no lo hace. El resultado es que la profundidad iterativa es en realidad más rápida que la búsqueda primero en anchura, a pesar de la generación repetida de estados. Por ejemplo, si $b = 10$ y $d = 5$, los números son

$$N(BPI) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$

$$N(BPA) = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100$$



En general, la profundidad iterativa es el método de búsqueda no informada preferido cuando hay un espacio grande de búsqueda y no se conoce la profundidad de la solución.

La búsqueda de profundidad iterativa es análoga a la búsqueda primero en anchura en la cual se explora, en cada iteración, una capa completa de nuevos nodos antes de continuar con la siguiente capa. Parecería que vale la pena desarrollar una búsqueda iterativa análoga a la búsqueda de coste uniforme, heredando las garantías de optimización del algoritmo evitando sus exigencias de memoria. La idea es usar límites crecientes de costo del camino en vez de aumentar límites de profundidad. El algoritmo que resulta.

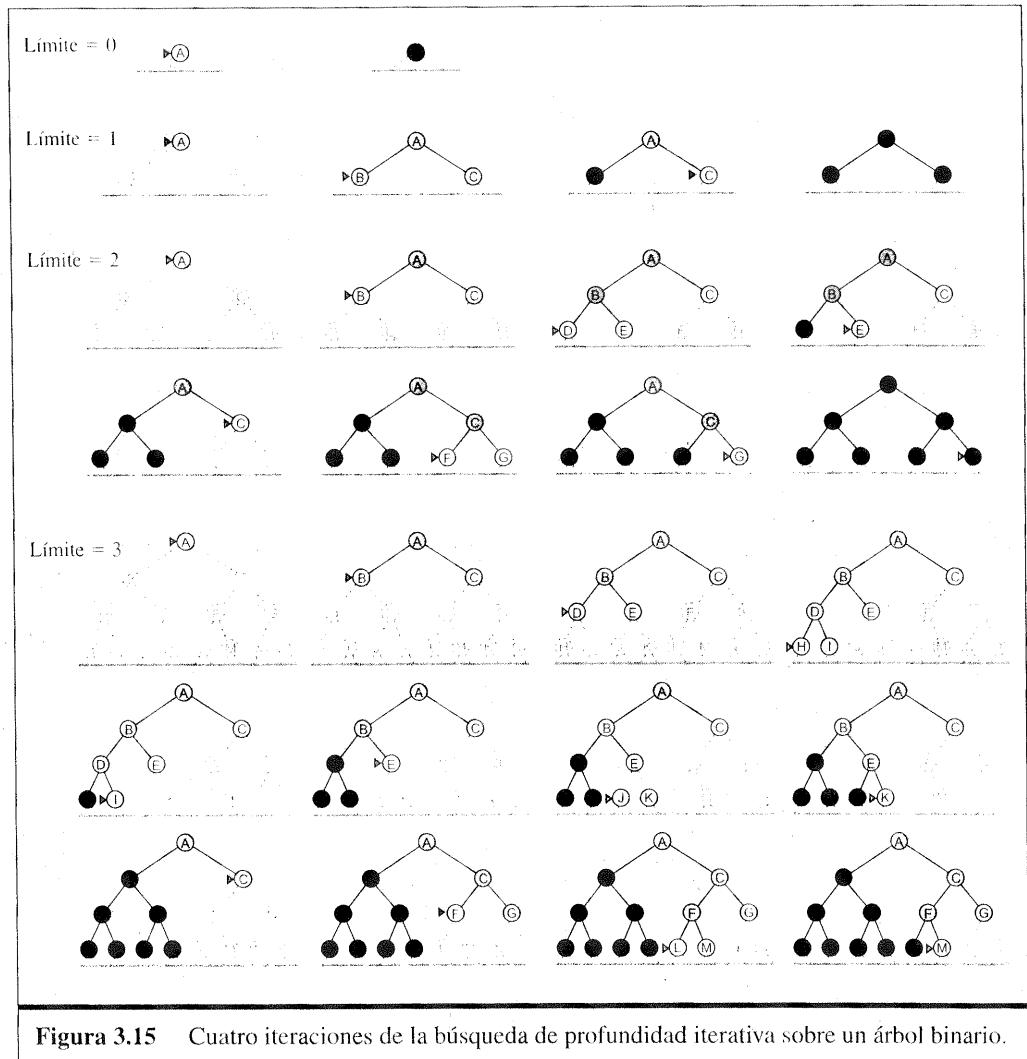


Figura 3.15 Cuatro iteraciones de la búsqueda de profundidad iterativa sobre un árbol binario.

BÚSQUEDA DE LONGITUD ITERATIVA

Llamado **búsqueda de longitud iterativa**, se explora en el Ejercicio 3.11. Resulta, lamentablemente, que la longitud iterativa incurre en gastos indirectos sustanciales comparado con la búsqueda de coste uniforme.

Búsqueda bidireccional

La idea de la búsqueda bidireccional es ejecutar dos búsquedas simultáneas: una hacia delante desde el estado inicial y la otra hacia atrás desde el objetivo, parando cuando las dos búsquedas se encuentren en el centro (Figura 3.16). La motivación es que $b^{d/2} + b^{d/2}$ es mucho menor que b^d , o en la figura, el área de los dos círculos pequeños es menor que el área de un círculo grande centrado en el inicio y que alcance al objetivo.

La búsqueda bidireccional se implementa teniendo una o dos búsquedas que comprobarán antes de ser expandido si cada nodo está en la frontera del otro árbol de bús-

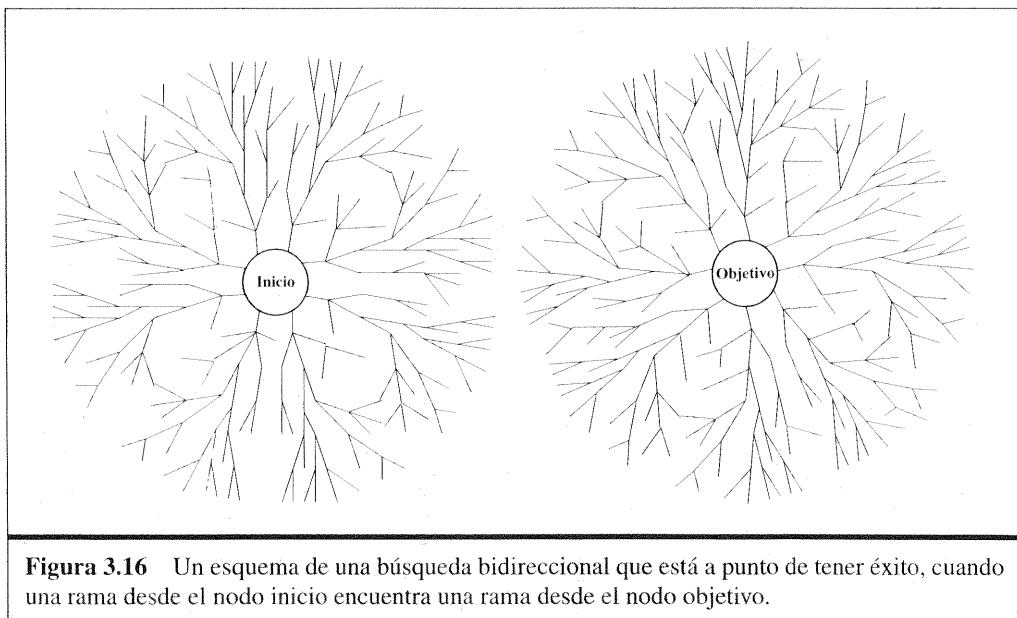


Figura 3.16 Un esquema de una búsqueda bidireccional que está a punto de tener éxito, cuando una rama desde el nodo inicio encuentra una rama desde el nodo objetivo.

queda; si esto ocurre, se ha encontrado una solución. Por ejemplo, si un problema tiene una solución a profundidad $d = 6$, y en cada dirección se ejecuta la búsqueda primero en anchura, entonces, en el caso peor, las dos búsquedas se encuentran cuando se han expandido todos los nodos excepto uno a profundidad 3. Para $b = 10$, esto significa un total de 22.200 nodos generados, comparado con 11.111.100 para una búsqueda primero en anchura estándar. Verificar que un nodo pertenece al otro árbol de búsqueda se puede hacer en un tiempo constante con una tabla *hash*, así que la complejidad en tiempo de la búsqueda bidireccional es $O(b^{d/2})$. Por lo menos uno de los árboles de búsqueda se debe mantener en memoria para que se pueda hacer la comprobación de pertenencia, de ahí que la complejidad en espacio es también $O(b^{d/2})$. Este requerimiento de espacio es la debilidad más significativa de la búsqueda bidireccional. El algoritmo es completo y óptimo (para costos uniformes) si las búsquedas son primero en anchura; otras combinaciones pueden sacrificar la completitud, optimización, o ambas.

La reducción de complejidad en tiempo hace a la búsqueda bidireccional atractiva, pero ¿cómo busca hacia atrás? Esto no es tan fácil como suena. Sean los **predecesores** de un nodo n , $Pred(n)$, todos los nodos que tienen como un sucesor a n . La búsqueda bidireccional requiere que $Pred(n)$ se calcule eficientemente. El caso más fácil es cuando todas las acciones en el espacio de estados son reversibles, así que $Pred(n) = Succ(n)$. Otro caso puede requerir ser ingenioso.

Consideremos la pregunta de qué queremos decir con «el objetivo» en la búsqueda «hacia atrás». Para el 8-puzzle y para encontrar un camino en Rumanía, hay solamente un estado objetivo, entonces la búsqueda hacia atrás se parece muchísimo a la búsqueda hacia delante. Si hay varios estados objetivo explícitamente catalogados (por ejemplo, los dos estados objetivo sin suciedad de la Figura 3.3) podemos construir un nuevo estado objetivo ficticio cuyos predecesores inmediatos son todos los estados objetivo reales. Alternativamente, algunos nodos generados redundantes se pueden evitar vien-

do el conjunto de estados objetivo como uno solo, cada uno de cuyos predecesores es también un conjunto de estados (específicamente, el conjunto de estados que tienen a un sucesor en el conjunto de estados objetivo. Véase también la Sección 3.6).

El caso más difícil para la búsqueda bidireccional es cuando el test objetivo da sólo una descripción implícita de algún conjunto posiblemente grande de estados objetivo, por ejemplo, todos los estados que satisfacen el test objetivo «jaque mate» en el ajedrez. Una búsqueda hacia atrás necesitaría construir las descripciones de «todos los estados que llevan al jaque mate al mover m_1 », etcétera; y esas descripciones tendrían que ser probadas de nuevo con los estados generados en la búsqueda hacia delante. No hay ninguna manera general de hacer esto eficientemente.

Comparación de las estrategias de búsqueda no informada

La Figura 3.17 compara las estrategias de búsqueda en términos de los cuatro criterios de evaluación expuestos en la Sección 3.4.

Criterio	Primero en anchura	Costo uniforme	Primero en profundidad	Profundidad limitada	Profundidad iterativa	Bidireccional (si aplicable)
¿Completa?	Sí ^a	Sí ^{a,b}	No	No	Sí ^a	Sí ^{a,d}
Tiempo	$O(b^{d+1})$	$O(b^{\lceil C^*/\ell \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Espacio	$O(b^{d+1})$	$O(b^{\lceil C^*/\ell \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
¿Optimal?	Sí	Sí	No	No	Sí ^c	Sí ^{c,d}

Figura 3.17 Evaluación de estrategias de búsqueda. b es el factor de ramificación; d es la profundidad de la solución más superficial; m es la máxima profundidad del árbol de búsqueda; ℓ es el límite de profundidad. Los superíndice significan lo siguiente: ^a completa si b es finita; ^b completa si los costos son $\geq \epsilon$ para ϵ positivo; ^c óptima si los costos son iguales; ^d si en ambas direcciones se utiliza la búsqueda primero en anchura.

3.5 Evitar estados repetidos

Hasta este punto, casi hemos ignorado una de las complicaciones más importantes al proceso de búsqueda: la posibilidad de perder tiempo expandiendo estados que ya han sido visitados y expandidos. Para algunos problemas, esta posibilidad nunca aparece; el espacio de estados es un árbol y hay sólo un camino a cada estado. La formulación eficiente del problema de las ocho reinas (donde cada nueva reina se coloca en la columna vacía de más a la izquierda) es eficiente en gran parte a causa de esto (cada estado se puede alcanzar sólo por un camino). Si formulamos el problema de las ocho reinas para poder colocar una reina en cualquier columna, entonces cada estado con n reinas se puede alcanzar por $n!$ caminos diferentes.

Para algunos problemas, la repetición de estados es inevitable. Esto incluye todos los problemas donde las acciones son reversibles, como son los problemas de búsque-

REJILLA
RECTANGULAR

da de rutas y los puzzles que deslizan sus piezas. Los árboles de la búsqueda para estos problemas son infinitos, pero si podamos parte de los estados repetidos, podemos cortar el árbol de búsqueda en un tamaño finito, generando sólo la parte del árbol que atraviesa el grafo del espacio de estados. Considerando solamente el árbol de búsqueda hasta una profundidad fija, es fácil encontrar casos donde la eliminación de estados repetidos produce una reducción exponencial del coste de la búsqueda. En el caso extremo, un espacio de estados de tamaño $d + 1$ (Figura 3.18(a)) se convierte en un árbol con 2^d hojas (Figura 3.18(b)). Un ejemplo más realista es la **rejilla rectangular**, como se ilustra en la Figura 3.18(c). Sobre una rejilla, cada estado tiene cuatro sucesores, entonces el árbol de búsqueda, incluyendo estados repetidos, tiene 4^d hojas; pero hay sólo $2d^2$ estados distintos en d pasos desde cualquier estado. Para $d = 20$, significa aproximadamente un billón de nodos, pero aproximadamente 800 estados distintos.

Entonces, si el algoritmo no detecta los estados repetidos, éstos pueden provocar que un problema resoluble llegue a ser irresoluble. La detección por lo general significa la comparación del nodo a expandir con aquellos que han sido ya expandidos; si se encuentra un empate, entonces el algoritmo ha descubierto dos caminos al mismo estado y puede desechar uno de ellos.

Para la búsqueda primero en profundidad, los únicos nodos en memoria son aquellos del camino desde la raíz hasta el nodo actual. La comparación de estos nodos permite al algoritmo descubrir los caminos que forman ciclos y que pueden eliminarse inmediatamente. Esto está bien para asegurar que espacios de estados finitos no hagan árboles de búsqueda infinitos debido a los ciclos; lamentablemente, esto no evita la proliferación exponencial de caminos que no forman ciclos, en problemas como los de la Figura 3.18. El único modo de evitar éstos es guardar más nodos en la memoria. Hay una compensación fundamental entre el espacio y el tiempo. *Los algoritmos que olvidan su historia están condenados a repetirla.*

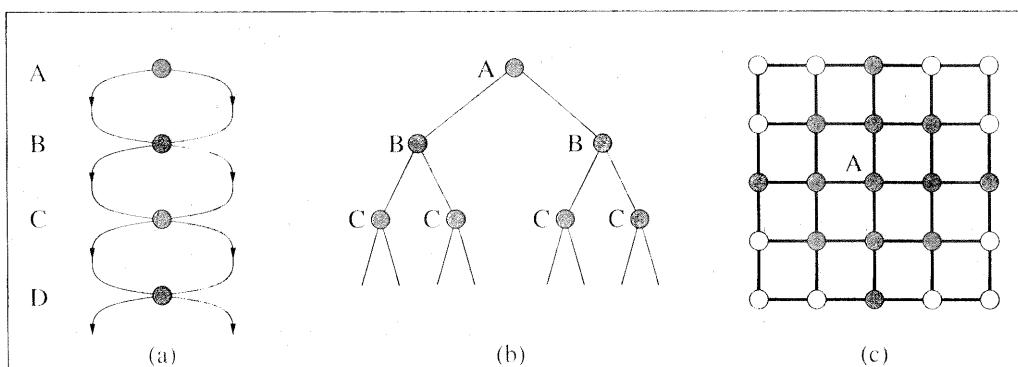


Figura 3.18 Espacios de estados que generan un árbol de búsqueda exponencialmente más grande. (a) Un espacio de estados en el cual hay dos acciones posibles que conducen de A a B, dos de B a C, etcétera. El espacio de estados contiene $d + 1$ estados, donde d es la profundidad máxima. (b) Correspondiente árbol de búsqueda, que tiene 2^d ramas correspondientes a 2^d caminos en el espacio. (c) Un espacio rejilla rectangular. Se muestran en gris los estados dentro de dos pasos desde el estado inicial (A).

LISTA CERRADA

LISTA ABIERTA

Si un algoritmo recuerda cada estado que ha visitado, entonces puede verse como la exploración directamente del grafo de espacio de estados. Podemos modificar el algoritmo general de BÚSQUEDA-ÁRBOLES para incluir una estructura de datos llamada **lista cerrada**, que almacene cada nodo expandido. (A veces se llama a la frontera de nodos no expandidos **lista abierta**.) Si el nodo actual se empareja con un nodo de la lista cerrada, se elimina en vez de expandirlo. Al nuevo algoritmo se le llama BÚSQUEDA-GRAFOS. Sobre problemas con muchos estados repetidos, la BÚSQUEDA-GRAFOS es mucho más eficiente que la BÚSQUEDA-ÁRBOLES. Los requerimientos en tiempo y espacio, en el caso peor, son proporcionales al tamaño del espacio de estados. Esto puede ser mucho más pequeño que $O(b^d)$.

La optimización para la búsqueda en grafos es una cuestión difícil. Dijimos antes que cuando se detecta un estado repetido, el algoritmo ha encontrado dos caminos al mismo estado. El algoritmo de BÚSQUEDA-GRAFOS de la Figura 3.19 siempre desecha el camino recién descubierto; obviamente, si el camino recién descubierto es más corto que el original, la BÚSQUEDA-GRAFOS podría omitir una solución óptima. Afortunadamente, podemos mostrar (Ejercicio 3.12) que esto no puede pasar cuando utilizamos la búsqueda de coste uniforme o la búsqueda primero en anchura con costos constantes; de ahí, que estas dos estrategias óptimas de búsqueda en árboles son también estrategias óptimas de búsqueda en grafos. La búsqueda con profundidad iterativa, por otra parte, utiliza la expansión primero en profundidad y fácilmente puede seguir un camino subóptimo a un nodo antes de encontrar el óptimo. De ahí, la búsqueda en grafos de profundidad iterativa tiene que comprobar si un camino recién descubierto a un nodo es mejor que el original, y si es así, podría tener que revisar las profundidades y los costos del camino de los descendientes de ese nodo.

Notemos que el uso de una lista cerrada significa que la búsqueda primero en profundidad y la búsqueda en profundidad iterativa tienen unos requerimientos lineales en espacio. Como el algoritmo de BÚSQUEDA-GRAFOS mantiene cada nodo en memoria, algunas búsquedas son irrealizables debido a limitaciones de memoria.

función BÚSQUEDA-GRAFOS(*problema*, *frontera*) **devuelve** una solución, o fallo

```

cerrado  $\leftarrow$  conjunto vacío
frontera  $\leftarrow$  INSERTAR(HACER-NODO(ESTADO-INICIAL[problema]), frontera)
bucle hacer
  si VACIA?(frontera) entonces devolver fallo
  nodo  $\leftarrow$  BORRAR-PRIMERO(frontera)
  si TEST-OBJETIVO[problema](ESTADO[nodo]) entonces devolver SOLUCIÓN(nodo)
  si ESTADO[nodo] no está en cerrado entonces
    añadir ESTADO[nodo] a cerrado
    frontera  $\leftarrow$  INSERTAR-TODO(EXPANDIR(nodo, problema), frontera)

```

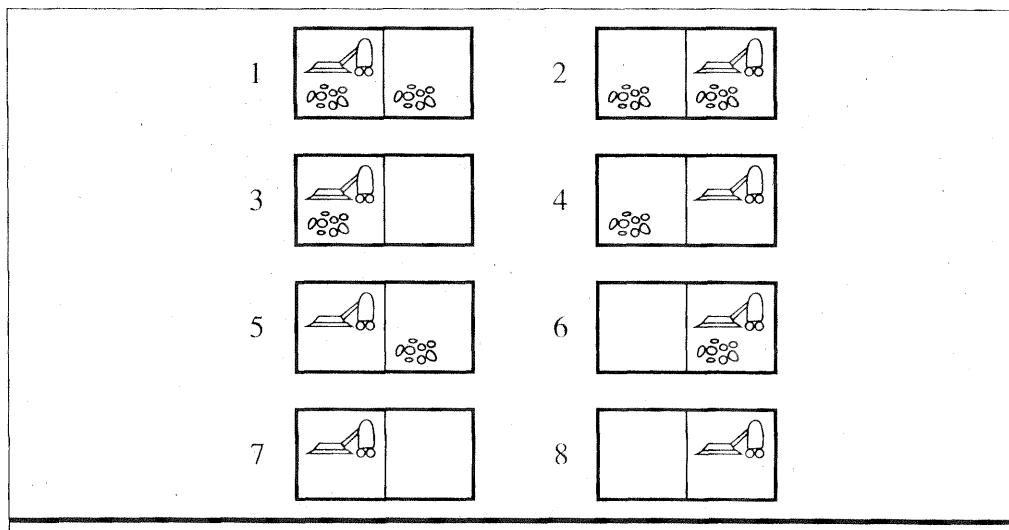
Figura 3.19 Algoritmo general de búsqueda en grafos. El conjunto cerrado puede implementarse como una tabla *hash* para permitir la comprobación eficiente de estados repetidos. Este algoritmo supone que el primer camino a un estado *s* es el más barato (véase el texto).

3.6 Búsqueda con información parcial

En la Sección 3.3 asumimos que el entorno es totalmente observable y determinista y que el agente conoce cuáles son los efectos de cada acción. Por lo tanto, el agente puede calcular exactamente cuál es el estado resultado de cualquier secuencia de acciones y siempre sabe en qué estado está. Su percepción no proporciona ninguna nueva información después de cada acción. ¿Qué pasa cuando el conocimiento de los estados o acciones es incompleto? Encontramos que diversos tipos de incompletitud conducen a tres tipos de problemas distintos:

1. **Problemas sin sensores** (también llamados **problemas conformados**): si el agente no tiene ningún sensor, entonces (por lo que sabe) podría estar en uno de los posibles estados iniciales, y cada acción por lo tanto podría conducir a uno de los posibles estados sucesores.
2. **Problemas de contingencia**: si el entorno es parcialmente observable o si las acciones son inciertas, entonces las percepciones del agente proporcionan *nueva* información después de cada acción. Cada percepción posible define una contingencia que debe de planearse. A un problema se le llama entre **adversarios** si la incertidumbre está causada por las acciones de otro agente.
3. **Problemas de exploración**: cuando se desconocen los estados y las acciones del entorno, el agente debe actuar para descubrirlos. Los problemas de exploración pueden verse como un caso extremo de problemas de contingencia.

Como ejemplo, utilizaremos el entorno del mundo de la aspiradora. Recuerde que el espacio de estados tiene ocho estados, como se muestra en la Figura 3.20. Hay tres acciones (*Izquierda*, *Derecha* y *Aspirar*) y el objetivo es limpiar toda la suciedad (estados 7 y 8). Si el entorno es observable, determinista, y completamente conocido, entonces el problema es trivialmente resoluble por cualquiera de los algoritmos que hemos descrito. Por



ejemplo, si el estado inicial es 5, entonces la secuencia de acciones [*Derecha, Aspirar*] alcanzará un estado objetivo, 8. El resto de esta sección trata con las versiones sin sensores y de contingencia del problema. Los problemas de exploración se tratan en la Sección 4.5, los problemas entre adversarios en el Capítulo 6.

Problemas sin sensores

Supongamos que el agente de la aspiradora conoce todos los efectos de sus acciones, pero no tiene ningún sensor. Entonces sólo sabe que su estado inicial es uno del conjunto $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Quizá supongamos que el agente está desesperado, pero de hecho puede hacerlo bastante bien. Como conoce lo que hacen sus acciones, puede, por ejemplo, calcular que la acción *Derecha* produce uno de los estados $\{2, 4, 6, 8\}$, y la secuencia de acción [*Derecha, Aspirar*] siempre terminará en uno de los estados $\{4, 8\}$. Finalmente, la secuencia [*Derecha, Aspirar, Izquierda, Aspirar*] garantiza alcanzar el estado objetivo 7 sea cual sea el estado inicio. Decimos que el agente puede **coaccionar** al mundo en el estado 7, incluso cuando no sepa dónde comenzó. Resumiendo: cuando el mundo no es completamente observable, el agente debe decidir sobre los *conjuntos* de estados que podría poner, más que por estados simples. Llamamos a cada conjunto de estados un **estado de creencia**, representando la creencia actual del agente con los estados posibles físicos en los que podría estar. (En un ambiente totalmente observable, cada estado de creencia contiene un estado físico.)

Para resolver problemas sin sensores, buscamos en el espacio de estados de creencia más que en los estados físicos. El estado inicial es un estado de creencia, y cada acción aplica un estado de creencia en otro estado de creencia. Una acción se aplica a un estado de creencia uniendo los resultados de aplicar la acción a cada estado físico del estado de creencia. Un camino une varios estados de creencia, y una solución es ahora un camino que conduce a un estado de creencia, *todos de cuyos miembros* son estados objetivo. La Figura 3.21 muestra el espacio de estados de creencia accesible para el mundo determinista de la aspiradora sin sensores. Hay sólo 12 estados de creencia accesibles, pero el espacio de estados de creencia entero contiene todo conjunto posible de estados físicos, por ejemplo, $2^8 = 256$ estados de creencia. En general, si el espacio de estados físico tiene S estados, el espacio de estados de creencia tiene 2^S estados de creencia.

Nuestra discusión hasta ahora de problemas sin sensores ha supuesto acciones deterministas, pero el análisis es esencialmente el mismo si el entorno es no determinista, es decir, si las acciones pueden tener varios resultados posibles. La razón es que, en ausencia de sensores, el agente no tiene ningún modo de decir qué resultado ocurrió en realidad, así que varios resultados posibles son estados físicos adicionales en el estado de creencia sucesor. Por ejemplo, supongamos que el entorno obedece a la ley de Murphy: la llamada acción de *Aspirar a veces* deposita suciedad en la alfombra *pero sólo si no ha ninguna suciedad allí*⁶. Entonces, si *Aspirar* se aplica al estado físico 4 (mirar la Figura 3.20), hay dos resultados posibles: los estados 2 y 4. Aplicado al estado de

⁶ Suponemos que la mayoría de los lectores afrontan problemas similares y pueden simpatizar con nuestro agente. Nos disculpamos a los dueños de los aparatos electrodomésticos modernos y eficientes que no pueden aprovecharse de este dispositivo pedagógico.

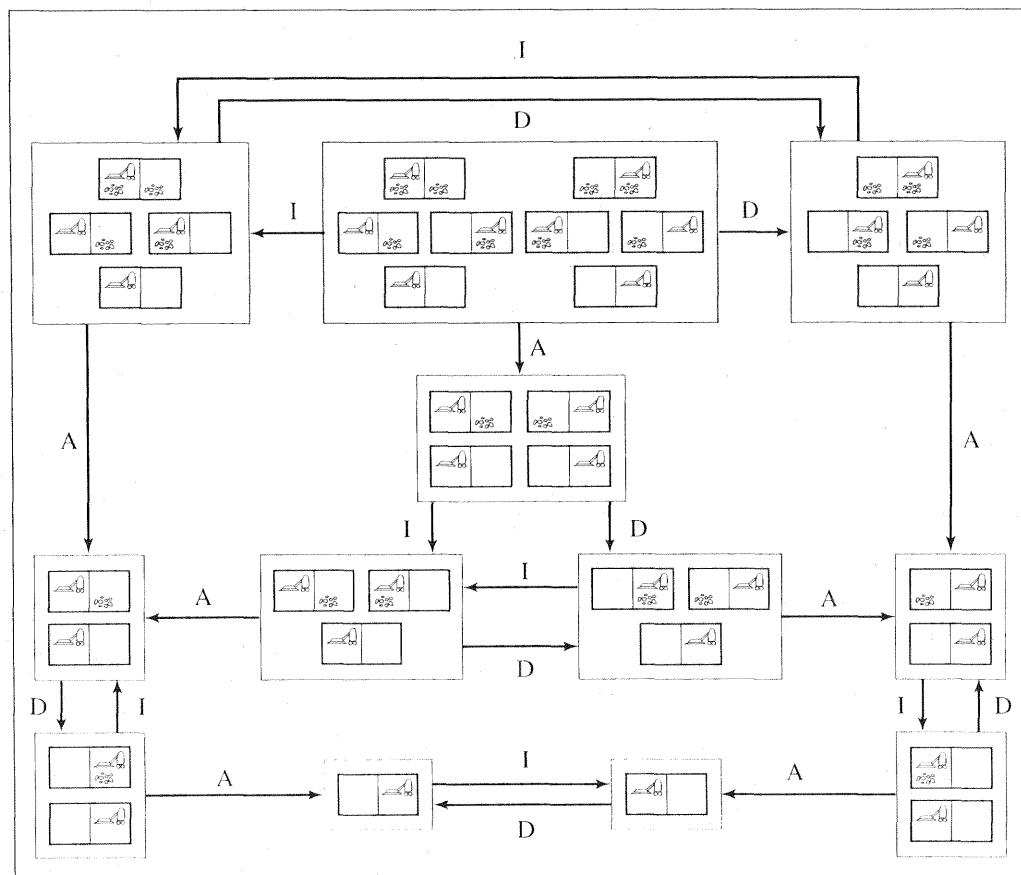


Figura 3.21 La parte accesible del espacio de estados de creencia para el mundo determinista de la aspiradora sin sensores. Cada caja sombreada corresponde a un estado de creencia simple. Desde cualquier punto, el agente está en un estado de creencia particular pero no sabe en qué estado físico está. El estado de creencia inicial (con ignorancia completa) es la caja central superior. Las acciones se representan por arcos etiquetados. Los autoarcos se omiten por claridad.

creencia inicial, $\{1, 2, 3, 4, 5, 6, 7, 8\}$. *Aspirar* conduce al estado de creencia que es la unión de los conjuntos resultados para los ocho estados físicos. Calculándolos, encontramos que el nuevo estado de creencia es $\{1, 2, 3, 4, 5, 6, 7, 8\}$. ¡Así, para un agente sin sensores en el mundo de la ley de Murphy, la acción *Aspirar* deja el estado de creencia inalterado! De hecho, el problema es no resoluble. (Véase el Ejercicio 3.18.) Por intuición, la razón es que el agente no puede distinguir si el cuadrado actual está sucio y de ahí que no puede distinguir si la acción *Aspirar* lo limpiará o creará más suciedad.

Problemas de contingencia

Cuando el entorno es tal que el agente puede obtener nueva información de sus sensores después de su actuación, el agente afronta **problemas de contingencia**. La solución en un problema de contingencia a menudo toma la forma de un árbol, donde cada rama se

puede seleccionar según la percepción recibida en ese punto del árbol. Por ejemplo, supongamos que el agente está en el mundo de la ley de Murphy y que tiene un sensor de posición y un sensor de suciedad local, pero no tiene ningún sensor capaz de detectar la suciedad en otros cuadrados. Así, la percepción $[I, Sucio]$ significa que el agente está en uno de los estados $\{1, 3\}$. El agente podría formular la secuencia de acciones $[Aspirar, Derecha, Aspirar]$. *Aspirar* cambiaría el estado al $\{5, 7\}$, y el mover a la derecha debería entonces cambiar el estado al $\{6, 8\}$. La ejecución de la acción final de *Aspirar* en el estado 6 nos lleva al estado 8, un objetivo, pero la ejecución en el estado 8 podría llevarnos para atrás al estado 6 (según la ley de Murphy), en el caso de que el plan falle.

Examinando el espacio de estados de creencia para esta versión del problema, fácilmente puede determinarse que ninguna secuencia de acciones rígida garantiza una solución a este problema. Hay, sin embargo, una solución si no insistimos en una secuencia de acciones *rígida*:

[Aspirar, Derecha, si [D, Suciedad] entonces Aspirar].

Esto amplía el espacio de soluciones para incluir la posibilidad de seleccionar acciones basadas en contingencias que surgen durante la ejecución. Muchos problemas en el mundo real, físico, son problemas de contingencia, porque la predicción exacta es imposible. Por esta razón, todas las personas mantienen sus ojos abiertos mientras andan o conducen.

Los problemas de contingencia *a veces* permiten soluciones puramente secuenciales. Por ejemplo, considere el mundo de la ley de Murphy *totalmente observable*. Las contingencias surgen si el agente realiza una acción *Aspirar* en un cuadrado limpio, porque la suciedad podría o no ser depositada en el cuadrado. Mientras el agente nunca haga esto, no surge ninguna contingencia y hay una solución secuencial desde cada estado inicial (Ejercicio 3.18).

Los algoritmos para problemas de contingencia son más complejos que los algoritmos estándar de búsqueda de este capítulo; ellos serán tratados en el Capítulo 12. Los problemas de contingencia también se prestan a un diseño de agente algo diferente, en el cual el agente puede actuar *antes* de que haya encontrado un plan garantizado. Esto es útil porque más que considerar por adelantado cada posible contingencia que *podría* surgir durante la ejecución, es a menudo mejor comenzar a actuar y ver qué contingencias surgen realmente. Entonces el agente puede seguir resolviendo el problema, teniendo en cuenta la información adicional. Este tipo de **intercalar** la búsqueda y la ejecución es también útil para problemas de exploración (véase la Sección 4.5) y para juegos (véase el Capítulo 6).

INTERCALAR

3.7 Resumen

Este capítulo ha introducido métodos en los que un agente puede seleccionar acciones en los ambientes deterministas, observables, estáticos y completamente conocidos. En tales casos, el agente puede construir secuencias de acciones que alcanzan sus objetivos; a este proceso se le llama **búsqueda**.

- Antes de que un agente pueda comenzar la búsqueda de soluciones, debe formular un objetivo y luego usar dicho objetivo para formular un **problema**.

- Un problema consiste en cuatro partes: el **estado inicial**, un conjunto de **acciones**, una función para el **test objetivo**, y una función de **costo del camino**. El entorno del problema se representa por un **espacio de estados**. Un **camino** por el espacio de estados desde el estado inicial a un estado objetivo es una **solución**.
- Un algoritmo sencillo y general de BÚSQUEDA-ÁRBOL puede usarse para resolver cualquier problema; las variantes específicas del algoritmo incorporan estrategias diferentes.
- Los algoritmos de búsqueda se juzgan sobre la base de **completitud, optimización, complejidad en tiempo y complejidad en espacio**. La complejidad depende de b , factor de ramificación en el espacio de estados, y d , profundidad de la solución más superficial.
- La **búsqueda primero en anchura** selecciona para su expansión el nodo no expandido más superficial en el árbol de búsqueda. Es completo, óptimo para costos unidad, y tiene la complejidad en tiempo y en espacio de $O(b^d)$. La complejidad en espacio lo hace poco práctico en la mayor parte de casos. La **búsqueda de coste uniforme** es similar a la búsqueda primero en anchura pero expande el nodo con el costo más pequeño del camino, $g(n)$. Es completo y óptimo si el costo de cada paso excede de una cota positiva ϵ .
- La **búsqueda primero en profundidad** selecciona para la expansión el nodo no expandido más profundo en el árbol de búsqueda. No es ni completo, ni óptimo, y tiene la complejidad en tiempo de $O(b^m)$ y la complejidad en espacio de $O(bm)$, donde m es la profundidad máxima de cualquier camino en el espacio de estados.
- La **búsqueda de profundidad limitada** impone un límite de profundidad fijo a una búsqueda primero en profundidad.
- La **búsqueda de profundidad iterativa** llama a la búsqueda de profundidad limitada aumentando este límite hasta que se encuentre un objetivo. Es completo, óptimo para costos unidad, y tiene la complejidad en tiempo de $O(b^d)$ y la complejidad en espacio de $O(bd)$.
- La **búsqueda bidireccional** puede reducir enormemente la complejidad en tiempo, pero no es siempre aplicable y puede requerir demasiado espacio.
- Cuando el espacio de estados es un grafo más que un árbol, puede valer la pena comprobar si hay estados repetidos en el árbol de búsqueda. El algoritmo de BÚSQUEDA-GRAFOS elimina todos los estados duplicados.
- Cuando el ambiente es parcialmente observable, el agente puede aplicar algoritmos de búsqueda en el espacio de **estados de creencia**, o los conjuntos de estados posibles en los cuales el agente podría estar. En algunos casos, se puede construir una sencilla secuencia solución; en otros casos, el agente necesita de un **plan de contingencia** para manejar las circunstancias desconocidas que puedan surgir.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

Casi todos los problemas de búsqueda en espacio de estados analizados en este capítulo tienen una larga historia en la literatura y son menos triviales de lo que parecían. El problema de los misioneros y caníbales, utilizado en el ejercicio 3.9, fue analizado con

detalle por Amarel (1968). Antes fue considerado en IA por Simon y Newell (1961), y en Investigación Operativa por Bellman y Dreyfus (1962). Estudios como estos y el trabajo de Newell y Simon sobre el Lógico Teórico (1957) y GPS (1961) provocaron el establecimiento de los algoritmos de búsqueda como las armas principales de los investigadores de IA en 1960 y el establecimiento de la resolución de problemas como la tarea principal de la IA. Desafortunadamente, muy pocos trabajos se han hecho para automatizar los pasos para la formulación de los problemas. Un tratamiento más reciente de la representación y abstracción del problema, incluyendo programas de IA que los llevan a cabo (en parte), está descrito en Knoblock (1990).

El 8-puzzle es el primo más pequeño del 15-puzzle, que fue inventado por el famoso americano diseñador de juegos Sam Loyd (1959) en la década de 1870. El 15-puzzle rápidamente alcanzó una inmensa popularidad en Estados Unidos, comparable con la más reciente causada por el Cubo de Rubik. También rápidamente atrajo la atención de matemáticos (Johnson y Story, 1879; Tait, 1880). Los editores de la *Revista Americana de Matemáticas* indicaron que «durante las últimas semanas el 15-puzzle ha ido creciendo en interés ante el público americano, y puede decirse con seguridad haber captado la atención de nueve de cada diez personas, de todas las edades y condiciones de la comunidad. Pero esto no ha inducido a los editores a incluir artículos sobre tal tema en la *Revista Americana de Matemáticas*, pero para el hecho que...» (sigue un resumen de interés matemático del 15-puzzle). Un análisis exhaustivo del 8-puzzle fue realizado por Schofield (1967) con la ayuda de un computador. Ratner y Warmuth (1986) mostraron que la versión general $n \times n$ del 15-puzzle pertenece a la clase de problemas NP-completos.

El problema de las 8-reinas fue publicado de manera anónima en la revista alemana de ajedrez *Schach* en 1848; más tarde fue atribuido a Max Bezzel. Fue republicado en 1850 y en aquel tiempo llamó la atención del matemático eminentemente Carl Friedrich Gauss, que intentó enumerar todas las soluciones posibles, pero encontró sólo 72. Nauck publicó más tarde en 1850 las 92 soluciones. Netto (1901) generalizó el problema a n reinas, y Abramson y Yung (1989) encontraron un algoritmo de orden $O(n)$.

Cada uno de los problemas de búsqueda del mundo real, catalogados en el capítulo, han sido el tema de mucho esfuerzo de investigación. Los métodos para seleccionar vuelos óptimos de líneas aéreas siguen estando mayoritariamente patentados, pero Carl de Marcken (personal de comunicaciones) ha demostrado que las restricciones y el precio de los billetes de las líneas aéreas lo convierten en algo tan enrevesado que el problema de seleccionar un vuelo óptimo es formalmente *indecidable*. El problema del viajante de comercio es un problema combinatorio estándar en informática teórica (Lawler, 1985; Lawler *et al.*, 1992). Karp (1972) demostró que el PVC es NP-duro, pero se desarrollaron métodos aproximados heurísticos efectivos (Lin y Kernighan, 1973). Arora (1998) inventó un esquema aproximado polinomial completo para los PVC Euclídeos. Shahookar y Mazumder (1991) inspeccionaron los métodos para la distribución VLSI, y muchos trabajos de optimización de distribuciones aparecen en las revistas de VLSI. La navegación robótica y problemas de ensamblaje se discuten en el Capítulo 25.

Los algoritmos de búsqueda no informada para resolver un problema son un tema central de la informática clásica (Horowitz y Sahni, 1978) y de la investigación operativa (Dreyfus, 1969); Deo y Pang (1984) y Gallo y Pallottino (1988) dan revisiones más

recientes. La búsqueda primero en anchura fue formulada por Moore (1959) para resolver laberintos. El método de **programación dinámica** (Bellman y Dreyfus, 1962), que sistemáticamente registra soluciones para todos los sub-problemas de longitudes crecientes, puede verse como una forma de búsqueda primero en anchura sobre grafos. El algoritmo de camino más corto entre dos puntos de Dijkstra (1959) es el origen de búsqueda de coste uniforme.

Una versión de profundidad iterativa, diseñada para hacer eficiente el uso del reloj en el ajedrez, fue utilizada por Slate y Atkin (1977) en el programa de juego AJEDREZ 4.5, pero la aplicación a la búsqueda del camino más corto en grafos se debe a Korf (1985a). La búsqueda bidireccional, que fue presentada por Pohl (1969, 1971), también puede ser muy eficaz en algunos casos.

Ambientes parcialmente observables y no deterministas no han sido estudiados en gran profundidad dentro de la resolución de problemas. Algunas cuestiones de eficacia en la búsqueda en estados de creencia han sido investigadas por Genesereth y Nourbakhsh (1993). Koenig y Simmons (1998) estudió la navegación del robot desde una posición desconocida inicial, y Erdmann y Mason (1988) estudió el problema de la manipulación robótica sin sensores, utilizando una forma continua de búsqueda en estados de creencia. La búsqueda de contingencia ha sido estudiada dentro del subcampo de la planificación (véase el Capítulo 12). Principalmente, planificar y actuar con información incierta se ha manejado utilizando las herramientas de probabilidad y la teoría de decisión (véase el Capítulo 17).

Los libros de texto de Nilsson (1971, 1980) son buenas fuentes bibliográficas generales sobre algoritmos clásicos de búsqueda. Una revisión comprensiva y más actualizada puede encontrarse en Korf (1988). Los artículos sobre nuevos algoritmos de búsqueda (que, notablemente, se siguen descubriendo) aparecen en revistas como *Artificial Intelligence*.



EJERCICIOS

- 3.1** Defina con sus propias palabras los siguientes términos: estado, espacio de estados, árbol de búsqueda, nodo de búsqueda, objetivo, acción, función sucesor, y factor de ramificación.
- 3.2** Explique por qué la formulación del problema debe seguir a la formulación del objetivo.
- 3.3** Supongamos que **ACCIONES-LEGALES**(*s*) denota el conjunto de acciones que son legales en el estado *s*, y **RESULTADO**(*a,s*) denota el estado que resulta de la realización de una acción legal *a* a un estado *s*. Defina **FUNCIÓN-SUCESOR** en términos **ACCIONES-LEGALES** y **RESULTADO**, y viceversa.
- 3.4** Demuestre que los estados del 8-puzzle están divididos en dos conjuntos disjuntos, tales que ningún estado en un conjunto puede transformarse en un estado del otro conjunto por medio de un número de movimientos. (*Consejo:* véase Berlekamp *et al.* (1982)). Invente un procedimiento que nos diga en qué clase está un estado dado, y explique por qué esto es bueno cuando generamos estados aleatorios.

3.5 Consideremos el problema de las 8-reinas usando la formulación «eficiente» incremental de la página 75. Explique por qué el tamaño del espacio de estados es al menos $\sqrt{n!}$ y estime el valor más grande para n para el cual es factible la exploración exhaustiva. (*Consejo:* saque una cota inferior para el factor de ramificación considerando el número máximo de cuadrados que una reina puede atacar en cualquier columna.)

3.6 ¿Conduce siempre un espacio de estados finito a un árbol de búsqueda finito? ¿Cómo un espacio de estados finito es un árbol? ¿Puede ser más preciso sobre qué tipos de espacios de estados siempre conducen a árboles de búsqueda finito? (Adaptado de Bender, 1996.)

3.7 Defina el estado inicial, test objetivo, función sucesor, y función costo para cada uno de los siguientes casos. Escoja una formulación que sea suficientemente precisa para ser implementada.

- a) Coloree un mapa plano utilizando sólo cuatro colores, de tal modo que dos regiones adyacentes no tengan el mismo color.
- b) Un mono de tres pies de alto está en una habitación en donde algunos plátanos están suspendidos del techo de ocho pies de alto. Le gustaría conseguir los plátanos. La habitación contiene dos cajas apilables, móviles y escalables de tres pies de alto.
- c) Tiene un programa que da como salida el mensaje «registro de entrada ilegal» cuando introducimos un cierto archivo de registros de entrada. Sabe que el tratamiento de cada registro es independiente de otros registros. Quiere descubrir que es ilegal.
- d) Tiene tres jarros, con capacidades 12 galones, ocho galones, y tres galones, y un grifo de agua. Usted puede llenar los jarros o vaciarlos de uno a otro o en el suelo. Tiene que obtener exactamente un galón.

3.8 Considere un espacio de estados donde el estado comienzo es el número 1 y la función sucesor para el estado n devuelve 2 estados, los números $2n$ y $2n + 1$.

- a) Dibuje el trozo del espacio de estados para los estados del 1 al 15.
- b) Supongamos que el estado objetivo es el 11. Enumere el orden en el que serán visitados los nodos por la búsqueda primero en anchura, búsqueda primero en profundidad con límite tres, y la búsqueda de profundidad iterativa.
- c) ¿Será apropiada la búsqueda bidireccional para este problema? Si es así, describa con detalle cómo trabajaría.
- d) ¿Qué es el factor de ramificación en cada dirección de la búsqueda bidireccional?
- e) ¿La respuesta (c) sugiere una nueva formulación del problema que permitiría resolver el problema de salir del estado 1 para conseguir un estado objetivo dado, con casi ninguna búsqueda?

3.9 El problema de los **misioneros y caníbales** en general se forma como sigue. tres misioneros y tres caníbales están en un lado de un río, con un barco que puede sostener a una o dos personas. Encuentre un modo de conseguir que todos estén en el otro lado, sin dejar alguna vez a un grupo de misioneros en un lugar excedido en número por los



caníbales. Este problema es famoso en IA porque fue el tema del primer trabajo que aproximó una formulación de problema de un punto de vista analítico (Amarel, 1968).

- a) Formule el problema de forma precisa, haciendo sólo las distinciones necesarias para asegurar una solución válida. Dibujar un diagrama del espacio de estados completo.
- b) Implemente y resuelva el problema de manera óptima utilizando un algoritmo apropiado de búsqueda. ¿Es una buena idea comprobar los estados repetidos?
- c) ¿Por qué cree que la gente utiliza mucho tiempo para resolver este puzzle, dado que el espacio de estados es tan simple?



3.10 Implemente dos versiones de la función sucesor para el 8-puzzle: uno que genere todos los sucesores a la vez copiando y editando la estructura de datos del 8-puzzle, y otro que genere un nuevo sucesor cada vez, llamando y modificando el estado padre directamente (haciendo las modificaciones necesarias). Escriba versiones de la búsqueda primero en profundidad con profundidad iterativa que use estas funciones y compare sus rendimientos.



3.11 En la página 89, mencionamos la **búsqueda de longitud iterativa**, una búsqueda análoga a la de costo uniforme iterativa. La idea es usar incrementos de los límites en el costo del camino. Si se genera un nodo cuyo costo del camino excede el límite actual, se descarta inmediatamente. Para cada nueva iteración, el límite se pone al coste más bajo del camino de cualquier nodo descartado en la iteración anterior.

- a) Muestre que este algoritmo es óptimo para costos de camino generales.
- b) Considere un árbol uniforme con factor de ramificación b , profundidad de solución d , y costos unidad. ¿Cuántas iteraciones requerirá la longitud iterativa?
- c) Ahora considere los costos en el rango continuo $[0,1]$ con un coste mínimo positivo ϵ . ¿Cuántas iteraciones requieren en el peor caso?
- d) Implemente el algoritmo y aplíquelo a los casos de los problemas del 8-puzzle y del viajante de comercio. Compare el funcionamiento del algoritmo con la búsqueda de costo uniforme, y comente sus resultados.



3.12 Demuestre que la búsqueda de costo uniforme y la búsqueda primero en anchura con costos constantes son óptimas cuando se utiliza con el algoritmo de BÚSQUEDA-GRAFOS. Muestre un espacio de estados con costos constantes en el cual la BÚSQUEDA-GRAFOS, utilizando profundidad iterativa, encuentre una solución subóptima.



3.13 Describa un espacio de estados en el cual la búsqueda de profundidad iterativa funcione mucho peor que la búsqueda primero en profundidad (por ejemplo, $O(n^2)$ contra $O(n)$).

3.14 Escriba un programa que tome como entrada dos URLs de páginas Web y encuentre un camino de links de una a la otra. ¿Cuál es una estrategia apropiada de búsqueda? ¿La búsqueda bidireccional es una idea buena? ¿Podría usarse un motor de búsqueda para implementar una función predecesor?

3.15 Considere el problema de encontrar el camino más corto entre dos puntos sobre un plano que tiene obstáculos poligonales convexos como los que se muestran en la Figura 3.22. Esto es una idealización del problema que tiene un robot para navegar en un entorno muy concurrido.

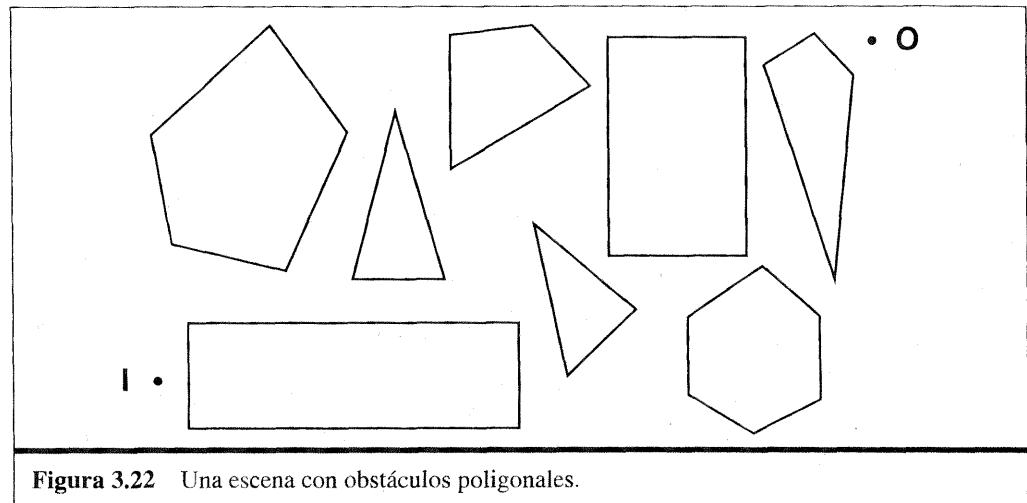


Figura 3.22 Una escena con obstáculos poligonales.

- Suponga que el espacio de estados consiste en todas las posiciones (x, y) en el plano. ¿Cuántos estados hay? ¿Cuántos caminos hay al objetivo?
- Explique brevemente por qué el camino más corto desde un vértice de un polígono a cualquier otro debe consistir en segmentos en línea recta que unen algunos vértices de los polígonos. Defina un espacio de estados bueno. ¿Cómo de grande es este espacio de estados?
- Defina las funciones necesarias para implementar el problema de búsqueda, incluyendo una función sucesor que toma un vértice como entrada y devuelve el conjunto de vértices que pueden alcanzarse en línea recta desde el vértice dado. (No olvide los vecinos sobre el mismo polígono.) Utilice la distancia en línea recta para la función heurística.
- Aplique uno o varios de los algoritmos de este capítulo para resolver un conjunto de problemas en el dominio, y comente su funcionamiento.

3.16 Podemos girar el problema de la navegación del Ejercicio 3.15 en un entorno como el siguiente:

- La percepción será una lista de posiciones, *relativas al agente*, de vértices visibles. ¡La percepción *no* incluye la posición del robot! El robot debe aprender su propia posición en el mapa; por ahora, puede suponer que cada posición tiene una «*visión*» diferente.
 - Cada acción será un vector describiendo un camino en línea recta a seguir. Si el camino está libre, la acción tiene éxito; en otro caso, el robot se para en el punto donde el camino cruza un obstáculo. Si el agente devuelve un vector de movimiento cero y está en el objetivo (que está fijado y conocido), entonces el entorno debería colocar al agente en una posición aleatoria (no dentro de un obstáculo).
 - La medida de rendimiento carga al agente, un punto por cada unidad de distancia atravesada y concede 1.000 puntos cada vez que se alcance el objetivo.
- Implemente este entorno y un agente de resolución de problemas. El agente tendrá que formular un nuevo problema después de la colocación en otro lugar, que implicará el descubrimiento de su posición actual.

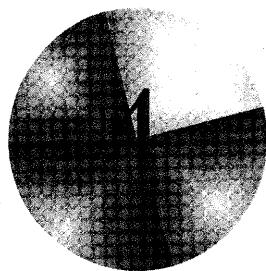
- b)** Documente el funcionamiento de su agente (teniendo que generar su agente el comentario conveniente de cómo se mueve a su alrededor) y el informe de su funcionamiento después de más de 100 episodios.
- c)** Modifique el entorno de modo que el 30 por ciento de veces el agente termine en un destino no planeado (escogido al azar de otros vértices visibles, o que no haya ningún movimiento). Esto es un modelo ordinario de los errores de movimiento de un robot real. Modifique al agente de modo que cuando se descubra tal error, averigüe dónde está y luego construya un plan para regresar donde estaba y resumir el viejo plan. ¡Recuerde que a veces recuperarse hacia donde estaba también podría fallar! Muestre un ejemplo satisfactorio del agente de modo que después de dos errores de movimientos sucesivos, todavía alcance el objetivo.
- d)** Ahora intente dos esquemas de recuperación diferentes después de un error: (1) diríjase al vértice más cercano sobre la ruta original; y (2) plantee una nueva ruta al objetivo desde la nueva posición. Compare el funcionamiento de estos tres esquemas de recuperación. ¿La inclusión de costos de la búsqueda afecta la comparación?
- e)** Ahora suponga que hay posiciones de las cuales la vista es idéntica (por ejemplo, suponga que el mundo es una rejilla con obstáculos cuadrados). ¿Qué tipo de problema afronta ahora el agente? ¿A qué se parecen las soluciones?
- 3.17** En la página 71, dijimos que no consideraríamos problemas con caminos de costos negativos. En este ejercicio, lo exploramos con más profundidad.
- a)** Suponga que las acciones pueden tener costos negativos arbitrariamente grandes; explique por qué esta posibilidad forzaría a cualquier algoritmo óptimo a explorar entero el espacio de estados.
- b)** ¿Ayuda si insistimos que los costos puedan ser mayores o iguales a alguna constante negativa c ? Considere tanto árboles como grafos.
- c)** Suponga que hay un conjunto de operadores que forman un ciclo, de modo que la ejecución del conjunto en algún orden no cause ningún cambio neto al estado. Si todos estos operadores tienen el coste negativo, ¿qué implica sobre el comportamiento óptimo para un agente en tal entorno?
- d)** Fácilmente pueden imaginarse operadores con alto coste negativo, incluso sobre dominios como la búsqueda de rutas. Por ejemplo, algunos caminos podrían tener un paisaje hermoso cuanto más lejano esté y pese más que los gastos normales en términos de tiempo y combustible. Explique, en términos exactos, dentro del contexto del espacio de estados de búsqueda, por qué la gente no conduce sobre ciclos pintorescos indefinidamente, y explique cómo definir el espacio de estados y operadores para la búsqueda de rutas de modo que agentes artificiales también puedan evitar la formación de ciclos.
- e)** ¿Puede usted pensar en un dominio real en el cual los costos son la causa de la formación de ciclos?
- 3.18** Considere el mundo de la aspiradora de dos posiciones sin sensores conforme a la ley de Murphy. Dibuje el espacio de estados de creencia accesible desde el estado de creencia inicial $\{1, 2, 3, 4, 5, 6, 7, 8\}$, y explique por qué el problema es irresoluble. Mues-

tre también que si el mundo es totalmente observable, entonces hay una secuencia solución para cada estado inicial posible.



3.19 Considere el problema del mundo de la aspiradora definido en la Figura 2.2

- a) ¿Cuál de los algoritmos definidos en este capítulo parece apropiado para este problema? ¿Debería el algoritmo comprobar los estados repetidos?
- b) Aplique el algoritmo escogido para obtener una secuencia de acciones óptima para un mundo de 3×3 cuyo estado inicial tiene la suciedad en los tres cuadrados superiores y el agente está en el centro.
- c) Construya un agente de búsqueda para el mundo de la aspiradora, y evalúe su rendimiento en un conjunto de 3×3 con probabilidad de 0,2 de suciedad en cada cuadrado.
- d) Compare su mejor agente de búsqueda con un agente reactivo sencillo aleatorio que aspira si hay suciedad y en otro caso se mueve aleatoriamente.
- e) Considere lo que sucedería si el mundo se ampliase a $n \times n$. ¿Cómo se modificaría el rendimiento del agente de búsqueda y del agente reactivo variando el n ?



Búsqueda informada y exploración

En donde veremos cómo la información sobre el espacio de estados puede impedir a los algoritmos cometer un error en la oscuridad.

El Capítulo 3 mostró que las estrategias de búsqueda no informadas pueden encontrar soluciones en problemas generando sistemáticamente nuevos estados y probándolos con el objetivo. Lamentablemente, estas estrategias son increíblemente ineficientes en la mayoría de casos. Este capítulo muestra cómo una estrategia de búsqueda informada (la que utiliza el conocimiento específico del problema) puede encontrar soluciones de una manera más eficiente. La Sección 4.1 describe las versiones informadas de los algoritmos del Capítulo 3, y la Sección 4.2 explica cómo se puede obtener la información específica necesaria del problema. Las Secciones 4.3 y 4.4 cubren los algoritmos que realizan la **búsqueda** puramente **local** en el espacio de estados, evaluando y modificando uno o varios estados más que explorando sistemáticamente los caminos desde un estado inicial. Estos algoritmos son adecuados para problemas en los cuales el coste del camino es irrelevante y todo lo que importa es el estado solución en sí mismo. La familia de algoritmos de búsqueda locales incluye métodos inspirados por la física estadística (**temple simulado**) y la biología evolutiva (**algoritmos genéticos**). Finalmente, la Sección 4.5 investiga la **búsqueda en línea**, en la cual el agente se enfrenta con un espacio de estados que es completamente desconocido.

4.1 Estrategias de búsqueda informada (heurísticas)

BÚSQUEDA PRIMERO
EL MEJOR

FUNCIÓN DE
EVALUACIÓN

FUNCIÓN HEURÍSTICA

A la aproximación general que consideraremos se le llamará **búsqueda primero el mejor**. La búsqueda primero el mejor es un caso particular del algoritmo general de BÚSQUEDA-ÁRBOLES o de BÚSQUEDA-GRAFOS en el cual se selecciona un nodo para la expansión basada en una **función de evaluación**, $f(n)$. Tradicionalmente, se selecciona para la expansión el nodo con la evaluación más baja, porque la evaluación mide la distancia al objetivo. La búsqueda primero el mejor puede implementarse dentro de nuestro marco general de búsqueda con una cola con prioridad, una estructura de datos que mantendrá la frontera en orden ascendente de f -valores.

El nombre de «búsqueda primero el mejor» es venerable pero inexacto. A fin de cuentas, si nosotros *realmente* pudiéramos expandir primero el mejor nodo, esto no sería una búsqueda en absoluto; sería una marcha directa al objetivo. Todo lo que podemos hacer es escoger el nodo que *parece* ser el mejor según la función de evaluación. Si la función de evaluación es exacta, entonces de verdad sería el mejor nodo; en realidad, la función de evaluación no será así, y puede dirigir la búsqueda por mal camino. No obstante, nos quedaremos con el nombre «búsqueda primero el mejor», porque «búsqueda aparentemente primero el mejor» es un poco incómodo.

Hay una familia entera de algoritmos de BÚSQUEDA-PRIMERO-MEJOR con funciones¹ de evaluación diferentes. Una componente clave de estos algoritmos es una **función heurística**², denotada $h(n)$:

$h(n)$ = coste estimado del camino más barato desde el nodo n a un nodo objetivo.

Por ejemplo, en Rumanía, podríamos estimar el coste del camino más barato desde Arad a Bucarest con la distancia en línea recta desde Arad a Bucarest.

Las funciones heurísticas son la forma más común de transmitir el conocimiento adicional del problema al algoritmo de búsqueda. Estudiaremos heurísticas con más profundidad en la Sección 4.2. Por ahora, las consideraremos funciones arbitrarias específicas del problema, con una restricción: si n es un nodo objetivo, entonces $h(n) = 0$. El resto de esta sección trata dos modos de usar la información heurística para dirigir la búsqueda.

Búsqueda voraz primero el mejor

BÚSQUEDA VORAZ
PRIMERO EL MEJOR

DISTANCIA EN LÍNEA
RECTA

La **búsqueda voraz primero el mejor**³ trata de expandir el nodo más cercano al objetivo, alegando que probablemente conduzca rápidamente a una solución. Así, evalúa los nodos utilizando solamente la función heurística: $f(n) = h(n)$.

Veamos cómo trabaja para los problemas de encontrar una ruta en Rumanía utilizando la heurística **distancia en línea recta**, que llamaremos h_{DLR} . Si el objetivo es Bucarest, tendremos que conocer las distancias en línea recta a Bucarest, que se muestran en la Figura 4.1. Por ejemplo, $h_{DLR}(En(Arad)) = 366$. Notemos que los valores de h_{DLR} no pueden calcularse de la descripción de problema en sí mismo. Además, debemos tener una

¹ El Ejercicio 4.3 le pide demostrar que esta familia incluye varios algoritmos familiares no informados.

² Una función heurística $h(n)$ toma un *nodo* como entrada, pero depende sólo del *estado* en ese nodo.

³ Nuestra primera edición la llamó **búsqueda avara (voraz)**; otros autores la han llamado **búsqueda primero el mejor**. Nuestro uso más general del término se sigue de Pearl (1984).

cierta cantidad de experiencia para saber que h_{DLR} está correlacionada con las distancias reales del camino y es, por lo tanto, una heurística útil.

Arad	366	Mehadia	241
Bucarest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	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 Valores de h_{DLR} . Distancias en línea recta a Bucarest.

La Figura 4.2 muestra el progreso de una búsqueda primero el mejor avara con h_{DLR} para encontrar un camino desde Arad a Bucarest. El primer nodo a expandir desde Arad será Sibiu, porque está más cerca de Bucarest que Zerind o que Timisoara. El siguiente nodo a expandir será Fagaras, porque es la más cercana. Fagaras en su turno genera Bucarest, que es el objetivo. Para este problema particular, la búsqueda primero el mejor avara usando h_{DLR} encuentra una solución sin expandir un nodo que no esté sobre el camino solución; de ahí, que su coste de búsqueda es mínimo. Sin embargo, no es óptimo: el camino vía Sibiu y Fagaras a Bucarest es 32 kilómetros más largo que el camino por Rimnicu Vilcea y Pitesti. Esto muestra por qué se llama algoritmo «avaro» (en cada paso trata de ponerse tan cerca del objetivo como pueda).

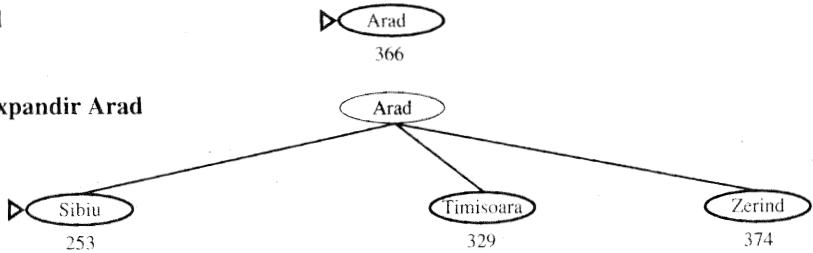
La minimización de $h(n)$ es susceptible de ventajas falsas. Considere el problema de ir de Iasi a Fagaras. La heurística sugiere que Neamt sea expandido primero, porque es la más cercana a Fagaras, pero esto es un callejón sin salida. La solución es ir primero a Vaslui (un paso que en realidad está más lejano del objetivo según la heurística) y luego seguir a Urziceni, Bucarest y Fagaras. En este caso, entonces, la heurística provoca nodos innecesarios para expandir. Además, si no somos cuidadosos en descubrir estados repetidos, la solución nunca se encontrará, la búsqueda oscilará entre Neamt e Iasi.

La búsqueda voraz primero el mejor se parece a la búsqueda primero en profundidad en el modo que prefiere seguir un camino hacia el objetivo, pero volverá atrás cuando llegue a un callejón sin salida. Sufre los mismos defectos que la búsqueda primero en profundidad, no es óptima, y es incompleta (porque puede ir hacia abajo en un camino infinito y nunca volver para intentar otras posibilidades). La complejidad en tiempo y espacio, del caso peor, es $O(b^m)$, donde m es la profundidad máxima del espacio de búsqueda. Con una buena función, sin embargo, pueden reducir la complejidad considerablemente. La cantidad de la reducción depende del problema particular y de la calidad de la heurística.

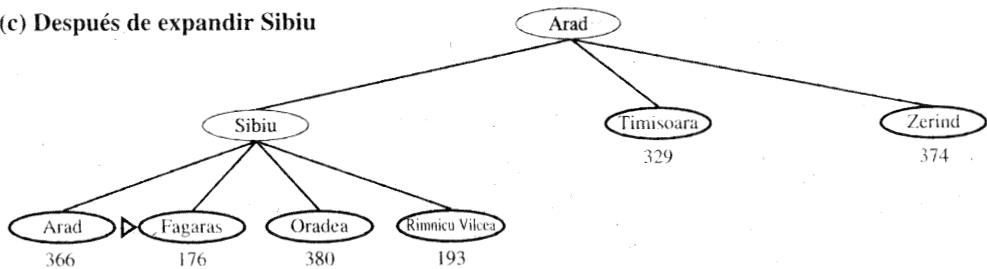
(a) Estado inicial



(b) Despues de expandir Arad



(c) Despues de expandir Sibiu



(d) Despues de expandir Fagaras

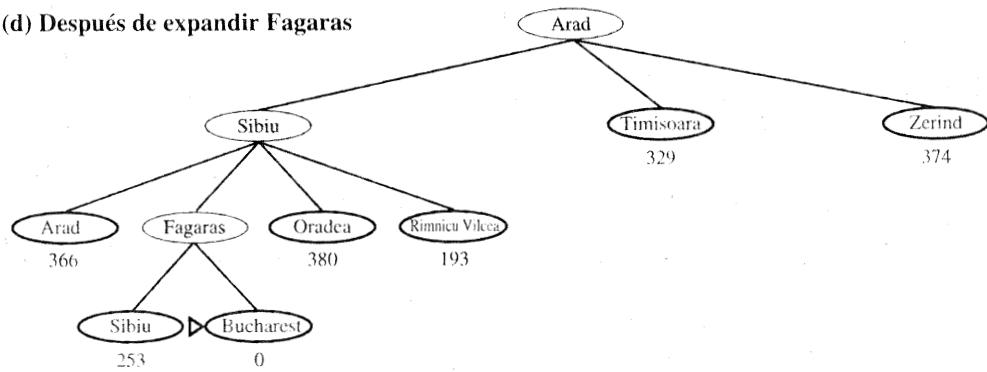


Figura 4.2 Etapas en una búsqueda primero el mejor avara para Bucarest utilizando la heurística distancia en línea recta h_{DLR} . Etiquetamos los nodos con sus h -valores.

Búsqueda A*: minimizar el costo estimado total de la solución

BÚSQUEDA A*

A la forma más ampliamente conocida de la búsqueda primero el mejor se le llama **búsqueda A*** (pronunciada «búsqueda A-estrella»). Evalúa los nodos combinando $g(n)$, el coste para alcanzar el nodo, y $h(n)$, el coste de ir al nodo objetivo:

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

Ya que la $g(n)$ nos da el coste del camino desde el nodo inicio al nodo n , y la $h(n)$ el coste estimado del camino más barato desde n al objetivo, tenemos:

$$f(n) = \text{coste más barato estimado de la solución a través de } n.$$

Así, si tratamos de encontrar la solución más barata, es razonable intentar primero el nodo con el valor más bajo de $g(n) + h(n)$. Resulta que esta estrategia es más que razonable: con tal de que la función heurística $h(n)$ satisfaga ciertas condiciones, la búsqueda A* es tanto completa como óptima.

La optimalidad de A* es sencilla de analizar si se usa con la BÚSQUEDA-ÁRBOLES. En este caso, A* es óptima si $h(n)$ es una **heurística admisible**, es decir, con tal de que la $h(n)$ nunca sobresteime el coste de alcanzar el objetivo. Las heurísticas admisibles son por naturaleza optimistas, porque piensan que el coste de resolver el problema es menor que el que es en realidad. Ya que $g(n)$ es el coste exacto para alcanzar n , tenemos como consecuencia inmediata que la $f(n)$ nunca sobresteime el coste verdadero de una solución a través de n .

Un ejemplo obvio de una heurística admisible es la distancia en línea recta h_{DLR} que usamos para ir a Bucarest. La distancia en línea recta es admisible porque el camino más corto entre dos puntos cualquiera es una línea recta, entonces la línea recta no puede ser una sobreestimación. En la Figura 4.3, mostramos el progreso de un árbol de búsqueda A* para Bucarest. Los valores de g se calculan desde los costos de la Figura 3.2, y los valores de h_{DLR} son los de la Figura 4.1. Notemos en particular que Bucarest aparece primero sobre la frontera en el paso (e), pero no se selecciona para la expansión porque su coste de $f(450)$ es más alto que el de Pitesti (417). Otro modo de decir esto consiste en que podría haber una solución por Pitesti cuyo coste es tan bajo como 417, entonces el algoritmo no se conformará con una solución que cuesta 450. De este ejemplo, podemos extraer una demostración general de que A*, *utilizando la BÚSQUEDA-ÁRBOLES, es óptimo si la $h(n)$ es admisible*. Supongamos que aparece en la frontera un nodo objetivo subóptimo G_2 , y que el coste de la solución óptima es C^* . Entonces, como G_2 es subóptimo y $h(G_2) = 0$ (cierto para cualquier nodo objetivo), sabemos que

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

Ahora considere un nodo n de la frontera que esté sobre un camino solución óptimo, por ejemplo, Pitesti en el ejemplo del párrafo anterior. (Siempre debe de haber ese nodo si existe una solución.) Si la $h(n)$ no sobreestima el coste de completar el camino solución, entonces sabemos que

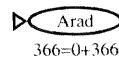
$$f(n) = g(n) + h(n) \leq C^*$$

Hemos demostrado que $f(n) \leq C^* < f(G_2)$, así que G_2 no será expandido y A* debe devolver una solución óptima.

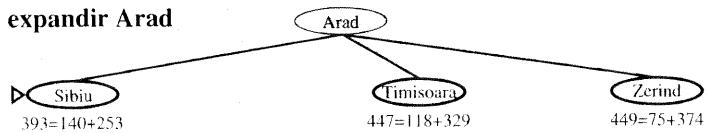
Si utilizamos el algoritmo de BÚSQUEDA-GRAFOS de la Figura 3.19 en vez de la BÚSQUEDA-ÁRBOLES, entonces esta demostración se estropea. Soluciones subóptimas pueden devolverse porque la BÚSQUEDA-GRAFOS puede desechar el camino óptimo en un estado repetido si éste no se genera primero (véase el Ejercicio 4.4). Hay dos modos de arreglar este problema. La primera solución es extender la BÚSQUEDA-GRAFOS de modo que deseche el camino más caro de dos caminos cualquiera encontrando al mismo nodo (véase la discusión de la Sección 3.5). El cálculo complementario es complicado, pero realmente garantiza la optimalidad. La segunda solución es asegurar que el camino óptimo a cualquier estado repetido es siempre el que primero seguimos (como en el caso de la búsqueda de costo uniforme). Esta propiedad se mantiene si imponemos una nueva condición a $h(n)$,



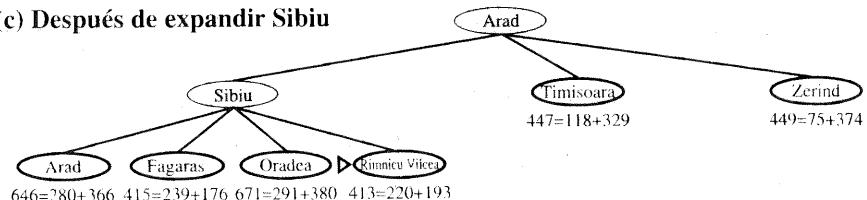
(a) Estado inicial



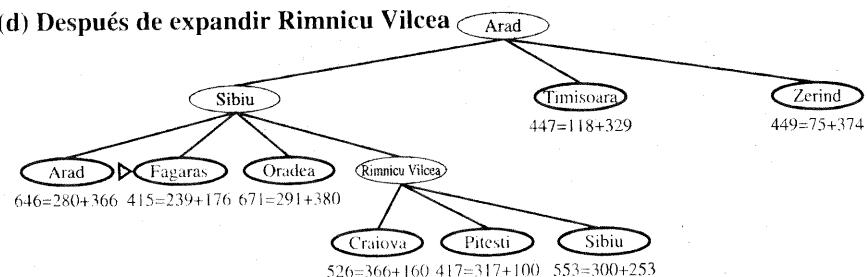
(b) Después de expandir Arad



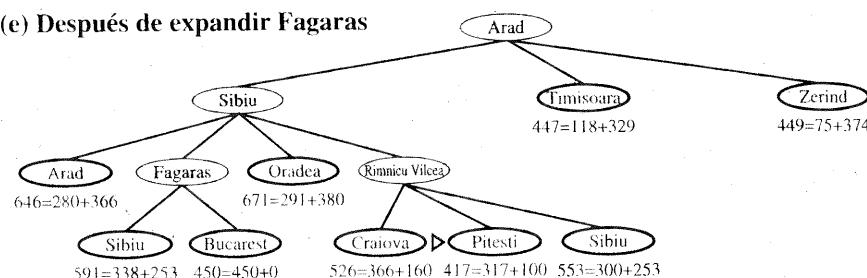
(c) Después de expandir Sibiu



(d) Después de expandir Rimnicu Vilcea



(e) Después de expandir Fagaras



(f) Después de expandir Pitesti

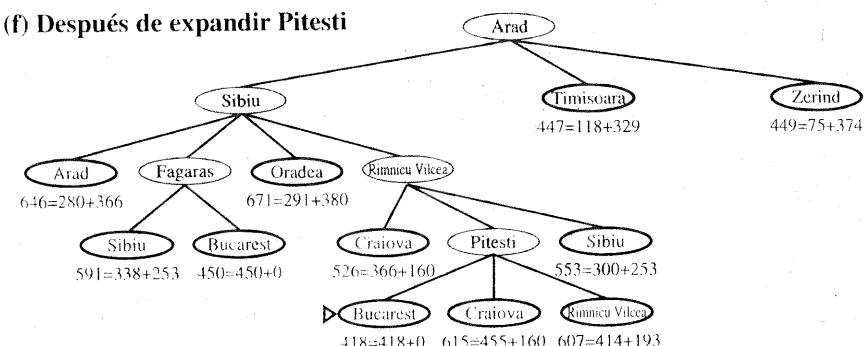


Figura 4.3 Etapas en una búsqueda A* para Bucarest. Etiquetamos los nodos con $f = g + h$. Los valores h son las distancias en línea recta a Bucarest tomadas de la Figura 4.1.

CONSISTENCIA

MONOTONÍA

DESIGUALDAD
TRIANGULAR

CURVAS DE NIVEL

concretamente condición de **consistencia** (también llamada **monotonía**). Una heurística $h(n)$ es consistente si, para cada nodo n y cada sucesor n' de n generado por cualquier acción a , el coste estimado de alcanzar el objetivo desde n no es mayor que el coste de alcanzar n' más el coste estimado de alcanzar el objetivo desde n' :

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

Esto es una forma de la **desigualdad triangular** general, que especifica que cada lado de un triángulo no puede ser más largo que la suma de los otros dos lados. En nuestro caso, el triángulo está formado por n , n' , y el objetivo más cercano a n . Es fácil demostrar (Ejercicio 4.7) que toda heurística consistente es también admisible. La consecuencia más importante de la consistencia es la siguiente: A* utilizando la BÚSQUEDA-GRAFOS es óptimo si la $h(n)$ es consistente.

Aunque la consistencia sea una exigencia más estricta que la admisibilidad, uno tiene que trabajar bastante para inventar heurísticas que sean admisibles, pero no consistentes. Todas las heurísticas admisibles de las que hablamos en este capítulo también son consistentes. Consideremos, por ejemplo, h_{DLR} . Sabemos que la desigualdad triangular general se satisface cuando cada lado se mide por la distancia en línea recta, y que la distancia en línea recta entre n y n' no es mayor que $c(n, a, n')$. De ahí que, h_{DLR} es una heurística consistente.

Otra consecuencia importante de la consistencia es la siguiente: *si $h(n)$ es consistente, entonces los valores de $f(n)$, a lo largo de cualquier camino, no disminuyen*. La demostración se sigue directamente de la definición de consistencia. Supongamos que n' es un sucesor de n ; entonces $g(n') = g(n) + c(n, a, n')$ para alguna a , y tenemos

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

Se sigue que la secuencia de nodos expandidos por A* utilizando la BÚSQUEDA-GRAFOS están en orden no decreciente de $f(n)$. De ahí que, el primer nodo objetivo seleccionado para la expansión debe ser una solución óptima, ya que todos los nodos posteriores serán al menos tan costosos.

El hecho de que los f-costos no disminuyan a lo largo de cualquier camino significa que podemos dibujar **curvas de nivel** en el espacio de estados, como las curvas de nivel en un mapa topográfico. La Figura 4.4 muestra un ejemplo. Dentro de la curva de nivel etiquetada con 400, todos los nodos tienen la $f(n)$ menor o igual a 400, etcétera. Entonces, debido a que A* expande el nodo de frontera de f-coste más bajo, podemos ver que A* busca hacia fuera desde el nodo inicial, añadiendo nodos en bandas concéntricas de f-coste creciente.

Con la búsqueda de coste uniforme (búsqueda A* utilizando $h(n) = 0$), las bandas serán «circulares» alrededor del estado inicial. Con heurísticas más precisas, las bandas se estirarán hacia el estado objetivo y se harán más concéntricas alrededor del camino óptimo. Si C^* es el coste del camino de solución óptima, entonces podemos decir lo siguiente:

- A* expande todos los nodos con $f(n) < C^*$.
- A* entonces podría expandir algunos nodos directamente sobre «la curva de nivel objetivo» (donde la $f(n) = C^*$) antes de seleccionar un nodo objetivo.

Por intuición, es obvio que la primera solución encontrada debe ser óptima, porque los nodos objetivos en todas las curvas de nivel siguientes tendrán el f-coste más alto, y así

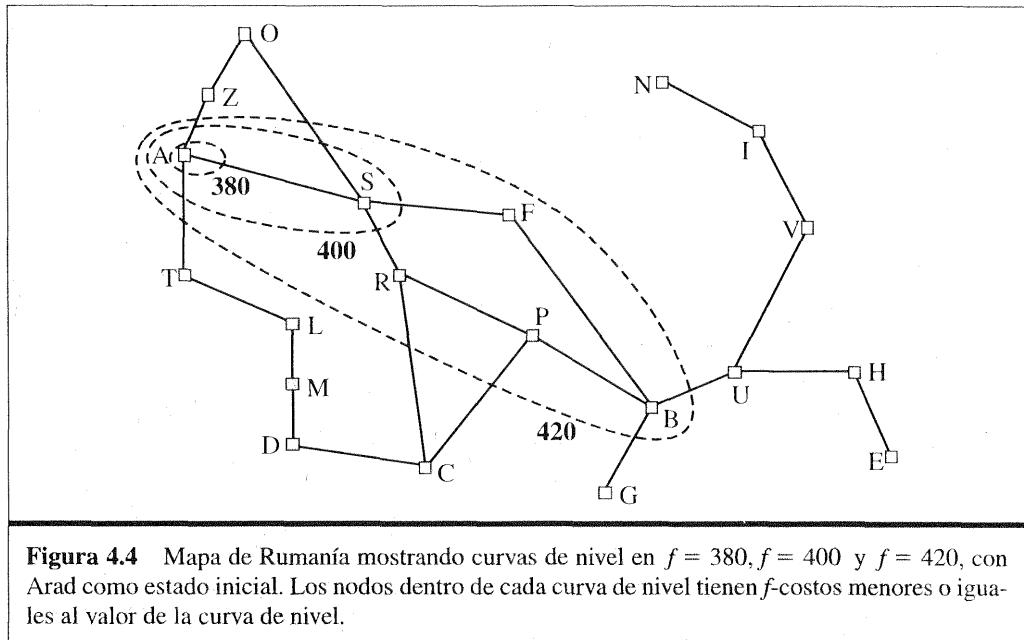


Figura 4.4 Mapa de Rumanía mostrando curvas de nivel en $f = 380$, $f = 400$ y $f = 420$, con Arad como estado inicial. Los nodos dentro de cada curva de nivel tienen f -costos menores o iguales al valor de la curva de nivel.

el g -coste más alto (porque todos los nodos de objetivo tienen $h(n) = 0$). Por intuición, es también obvio que la búsqueda A^* es completa. Como añadimos las bandas de f creciente, al final debemos alcanzar una banda donde f sea igual al coste del camino a un estado objetivo⁴.

Note que A^* no expande ningún nodo con $f(n) > C^*$ (por ejemplo, Timisoara no se expande en la Figura 4.3 aun cuando sea un hijo de la raíz). Decimos que el subárbol debajo de Timisoara está **podado**; como h_{DLR} es admisible, el algoritmo seguramente no hará caso de este subárbol mientras todavía se garantiza la optimalidad. El concepto de poda (eliminación de posibilidades a considerar sin necesidad de examinarlas) es importante para muchas áreas de IA.

Una observación final es que entre los algoritmos óptimos de este tipo (los algoritmos que extienden los caminos de búsqueda desde la raíz) A^* es **óptimamente eficiente** para cualquier función heurística. Es decir, ningún otro algoritmo óptimo garantiza expandir menos nodos que A^* (excepto posiblemente por los desempates entre los nodos con $f(n) = C^*$). Esto es porque cualquier algoritmo que no expanda todos los nodos con $f(n) < C^*$ corre el riesgo de omitir la solución óptima.

Nos satisface bastante la búsqueda A^* ya que es completa, óptima, y óptimamente eficiente entre todos los algoritmos. Lamentablemente, no significa que A^* sea la respuesta a todas nuestras necesidades de búsqueda. La dificultad es que, para la mayoría de los problemas, el número de nodos dentro de la curva de nivel del objetivo en el espacio de búsqueda es todavía exponencial en la longitud de la solución. Aunque la demostración del resultado está fuera del alcance de este libro, se ha mostrado que el

PODA

ÓPTIMAMENTE
EFICIENTE

⁴ La completitud requiere que haya sólo un número finito de nodos con el coste menor o igual a C^* , una condición que es cierta si todos los costos exceden algún número ϵ finito y si b es finito.

crecimiento exponencial ocurrirá a no ser que el error en la función heurística no crezca más rápido que el logaritmo del coste de camino real. En notación matemática, la condición para el crecimiento subexponencial es

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

donde $h^*(n)$ es el coste real de alcanzar el objetivo desde n . En la práctica, para casi todas las heurísticas, el error es al menos proporcional al coste del camino, y el crecimiento exponencial que resulta alcanza la capacidad de cualquier computador. Por esta razón, es a menudo poco práctico insistir en encontrar una solución óptima. Uno puede usar las variantes de A* que encuentran rápidamente soluciones subóptimas, o uno a veces puede diseñar heurísticas que sean más exactas, pero no estrictamente admisibles. En cualquier caso, el empleo de buenas heurísticas proporciona enormes ahorros comparados con el empleo de una búsqueda no informada. En la Sección 4.2, veremos el diseño de heurísticas buenas.

El tiempo computacional no es, sin embargo, la desventaja principal de A*. Como mantiene todos los nodos generados en memoria (como hacen todos los algoritmos de BÚSQUEDA-GRAFOS), A*, por lo general, se queda sin mucho espacio antes de que se quede sin tiempo. Por esta razón, A* no es práctico para problemas grandes. Los algoritmos recientemente desarrollados han vencido el problema de espacio sin sacrificar la optimalidad o la completitud, con un pequeño coste en el tiempo de ejecución. Éstos se discutirán a continuación.

Búsqueda heurística con memoria acotada

La forma más simple de reducir la exigencia de memoria para A* es adaptar la idea de profundizar iterativamente al contexto de búsqueda heurística, resultando así el algoritmo A* de profundidad iterativa (A*PI). La diferencia principal entre A*PI y la profundidad iterativa estándar es que el corte utilizado es el f -coste ($g + h$) más que la profundidad; en cada iteración, el valor del corte es el f -coste más pequeño de cualquier nodo que excedió el corte de la iteración anterior. A*PI es práctico para muchos problemas con costos unidad y evita el trabajo asociado con el mantenimiento de una cola ordenada de nodos. Lamentablemente, esto sufre de las mismas dificultades con costos de valores reales como hace la versión iterativa de búsqueda de coste uniforme descrita en el Ejercicio 3.11. Esta sección brevemente examina dos algoritmos más recientes con memoria acotada, llamados BRPM y A*M.

La **búsqueda recursiva del primero mejor** (BRPM) es un algoritmo sencillo recursivo que intenta imitar la operación de la búsqueda primero el mejor estándar, pero utilizando sólo un espacio lineal. En la Figura 4.5 se muestra el algoritmo. Su estructura es similar a la búsqueda primero en profundidad recursiva, pero más que seguir indefinidamente hacia abajo el camino actual, mantiene la pista del f -valor del mejor camino alternativo disponible desde cualquier antepasado del nodo actual. Si el nodo actual excede este límite, la recursividad vuelve atrás al camino alternativo. Como la recursividad vuelve atrás, la BRPM sustituye los f -valores de cada nodo a lo largo del camino con el mejor f -valor de su hijo. De este modo, la BRPM recuerda el f -valor de la mejor hoja en el subárbol olvidado y por lo tanto puede decidir si merece la pena expandir el subárbol más tarde. La Figura 4.6 muestra cómo la BRPM alcanza Bucarest.

```

función BÚSQUEDA-RECURSIVA-PRIMERO-MEJOR(problema) devuelve una solución, o fallo
  BRPM(problema, HACER-NODO(ESTADO-INICIAL[problema] ),∞)

función BRPM(problema,nodo,f_límite) devuelve una solución, o fallo y un nuevo límite
f-costo
  si TEST-OBJETIVO[problema](estado) entonces devolver nodo
  sucesores ← EXPANDIR(nodo, problema)
  si sucesores está vacío entonces devolver fallo, ∞
  para cada s en sucesores hacer
    f[s] ← max(g(s) + h(s), f[nodo])
  repetir
    mejor ← nodo con f-valor más pequeño de sucesores
    si f[mejor] > f_límite entonces devolver fallo, f[mejor]
    alternativa ← nodo con el segundo f-valor más pequeño entre los sucesores
    resultado,f[mejor] ← BRPM(problema,mejor,min(f_límite,alternativa))
    si resultado ≠ fallo entonces devolver resultado

```

Figura 4.5 Algoritmo para la búsqueda primero el mejor recursiva.

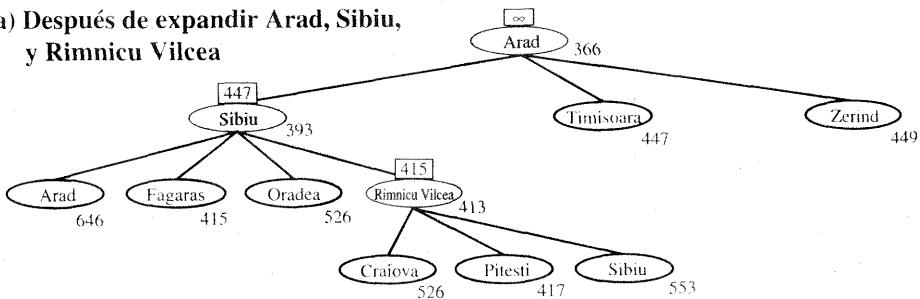
La BRPM es algo más eficiente que A*PI, pero todavía sufre de la regeneración excesiva de nodos. En el ejemplo de la Figura 4.6, la BRPM sigue primero el camino vía Rimnicu Vilcea, entonces «cambia su opinión» e intenta Fagaras, y luego cambia su opinión hacia atrás otra vez. Estos cambios de opinión ocurren porque cada vez que el mejor camino actual se extiende, hay una buena posibilidad que aumente su *f*-valor (*h* es por lo general menos optimista para nodos más cercanos al objetivo). Cuando esto pasa, en particular en espacios de búsqueda grandes, el segundo mejor camino podría convertirse en el mejor camino, entonces la búsqueda tiene que retroceder para seguirlo. Cada cambio de opinión corresponde a una iteración de A*PI, y podría requerir muchas nuevas expansiones de nodos olvidados para volver a crear el mejor camino y ampliarlo en un nodo más.

Como A*, BRPM es un algoritmo óptimo si la función heurística *h(n)* es admisible. Su complejidad en espacio es $O(bd)$, pero su complejidad en tiempo es bastante difícil de caracterizar: depende de la exactitud de la función heurística y de cómo cambia a menudo el mejor camino mientras se expanden los nodos. Tanto A*PI como BRPM están sujetos al aumento potencialmente exponencial de la complejidad asociada con la búsqueda en grafos (véase la Sección 3.5), porque no pueden comprobar para saber si hay estados repetidos con excepción de los que están en el camino actual. Así, pueden explorar el mismo estado muchas veces.

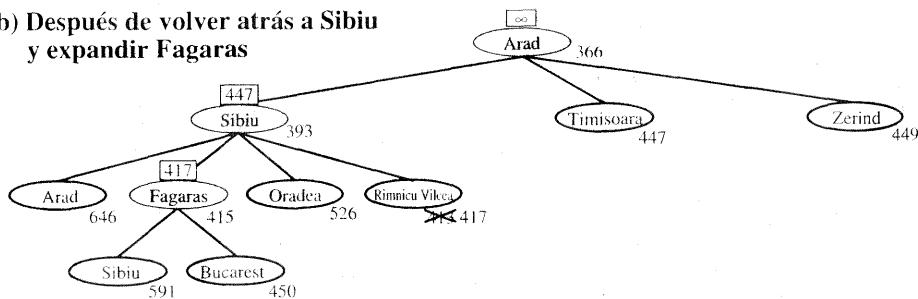
A*PI y BRPM sufren de utilizar *muy poca* memoria. Entre iteraciones, A*PI conserva sólo un número: el límite *f*-coste actual. BRPM conserva más información en la memoria, pero usa sólo $O(bd)$ de memoria: incluso si hubiera más memoria disponible, BRPM no tiene ningún modo de aprovecharse de ello.

Parece sensible, por lo tanto, usar toda la memoria disponible. Dos algoritmos que hacen esto son A*M (A* con memoria acotada) y A*MS (A*M simplificada). Describiremos A*MS, que es más sencillo. A*MS avanza como A*, expandiendo la mejor hoja

(a) Despues de expandir Arad, Sibiu, y Rimnicu Vilcea



(b) Despues de volver atrás a Sibiu y expandir Fagaras



(c) Despues de cambiar a Rimnicu Vilcea y expandir Pitesti

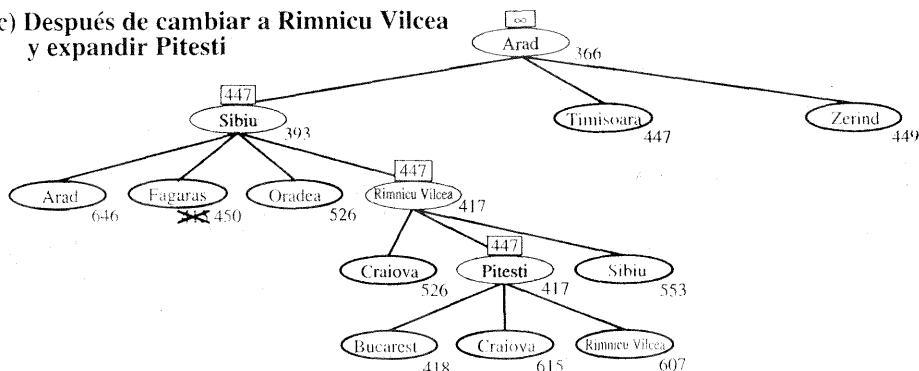


Figura 4.6 Etapas en una búsqueda BRPM para la ruta más corta a Bucarest. Se muestra el valor del f -límite para cada llamada recursiva sobre cada nodo actual. (a) Se sigue el camino vía Rimnicu Vilcea hasta que la mejor hoja actual (Pitesti) tenga un valor que es peor que el mejor camino alternativo (Fagaras). (b) La recursividad se aplica y el mejor valor de las hojas del subárbol olvidado (417) se le devuelve hacia atrás a Rimnicu Vilcea; entonces se expande Fagaras, revela un mejor valor de hoja de 450. (c) La recursividad se aplica y el mejor valor de las hojas del subárbol olvidado (450) se le devuelve hacia atrás a Fagaras; entonces se expande Rimnicu Vilcea. Esta vez, debido a que el mejor camino alternativo (por Timisoara) cuesta por lo menos 447, la expansión sigue por Bucarest.

hasta que la memoria esté llena. En este punto, no se puede añadir un nuevo nodo al árbol de búsqueda sin retirar uno viejo. A*MS siempre retira el peor nodo hoja (el de f -valor más alto). Como en la BRPM, A*MS entonces devuelve hacia atrás, a su padre, el valor del nodo olvidado. De este modo, el antepasado de un subárbol olvidado sabe la ca-

lidad del mejor camino en el subárbol. Con esta información, A*MS vuelve a generar el subárbol sólo cuando *todos los otros caminos* parecen peores que el camino olvidado. Otro modo de decir esto consiste en que, si todos los descendientes de un nodo n son olvidados, entonces no sabremos por qué camino ir desde n , pero todavía tendremos una idea de cuánto vale la pena ir desde n a cualquier nodo.

El algoritmo completo es demasiado complicado para reproducirse aquí⁵, pero hay un matiz digno de mencionar. Dijimos que A*MS expande la mejor hoja y suprime la peor hoja. ¿Y si *todos* los nodos hoja tienen el mismo f -valor? Entonces el algoritmo podría seleccionar el mismo nodo para eliminar y expandir. A*MS soluciona este problema expandiendo la mejor hoja *más nueva* y suprimiendo la peor hoja *más vieja*. Estos pueden ser el mismo nodo sólo si hay una sola hoja; en ese caso, el árbol actual de búsqueda debe ser un camino sólo desde la raíz a la hoja llenando toda la memoria. Si la hoja no es un nodo objetivo, entonces, *incluso si está sobre un camino solución óptimo*, esa solución no es accesible con la memoria disponible. Por lo tanto, el nodo puede descartarse como si no tuviera ningún sucesor.

A*MS es completo si hay alguna solución alcanzable, es decir, si d , la profundidad del nodo objetivo más superficial, es menor que el tamaño de memoria (expresada en nodos). Es óptimo si cualquier solución óptima es alcanzable; de otra manera devuelve la mejor solución alcanzable. En términos prácticos, A*MS bien podría ser el mejor algoritmo de uso general para encontrar soluciones óptimas, en particular cuando el espacio de estados es un grafo, los costos no son uniformes, y la generación de un nodo es costosa comparada con el gasto adicional de mantener las listas abiertas y cerradas.

Sobre problemas muy difíciles, sin embargo, a menudo al A*MS se le fuerza a cambiar hacia delante y hacia atrás continuamente entre un conjunto de caminos solución candidatos, y sólo un pequeño subconjunto de ellos puede caber en memoria (esto se parece al problema de *thrashing* en sistemas de paginación de disco). Entonces el tiempo extra requerido para la regeneración repetida de los mismos nodos significa que los problemas que serían prácticamente resolubles por A*, considerando la memoria ilimitada, se harían intratables para A*MS. Es decir, *las limitaciones de memoria pueden hacer a un problema intratable desde el punto de vista de tiempo de cálculo*. Aunque no haya ninguna teoría para explicar la compensación entre el tiempo y la memoria, parece que esto es un problema ineludible. La única salida es suprimir la exigencia de optimización.

THRASHING



ESPACIO DE ESTADOS METANIVEL

ESPACIO DE ESTADOS A NIVEL DE OBJETO

Aprender a buscar mejor

Hemos presentado varias estrategias fijas (primero en anchura, primero el mejor avara, etcétera) diseñadas por informáticos. ¿Podría un agente aprender a buscar mejor? La respuesta es sí, y el método se apoya sobre un concepto importante llamado el **espacio de estados metanivel**. Cada estado en un espacio de estados metanivel captura el estado interno (computacional) de un programa que busca en un **espacio de estados a nivel de objeto** como Rumanía. Por ejemplo, el estado interno del algoritmo A* consiste en el

⁵ Un esbozo apareció en la primera edición de este libro.

árbol actual de búsqueda. Cada acción en el espacio de estados metanivel es un paso de cómputo que cambia el estado interno; por ejemplo, cada paso de cómputo en A* expande un nodo hoja y añade sus sucesores al árbol. Así, la Figura 4.3, la cual muestra una secuencia de árboles de búsqueda más y más grandes, puede verse como la representación de un camino en el espacio de estados metanivel donde cada estado sobre el camino es un árbol de búsqueda a nivel de objeto.

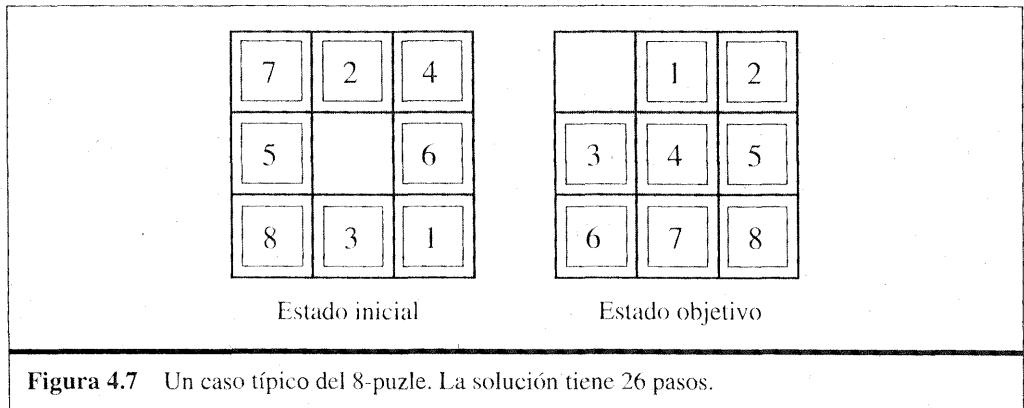
Ahora, el camino en la Figura 4.3 tiene cinco pasos, incluyendo un paso, la expansión de Fagaras, que no es especialmente provechoso. Para problemas más difíciles, habrá muchos de estos errores, y un algoritmo de **aprendizaje metanivel** puede aprender de estas experiencias para evitar explorar subárboles no prometedores. Las técnicas usadas para esta clase de aprendizaje están descritas en el Capítulo 21. El objetivo del aprendizaje es reducir al mínimo el **coste total** de resolver el problema, compensar el costo computacional y el coste del camino.

APRENDIZAJE
METANIVEL

4.2 Funciones heurísticas

En esta sección, veremos heurísticas para el 8-puzzle, para que nos den información sobre la naturaleza de las heurísticas en general.

El 8-puzzle fue uno de los primeros problemas de búsqueda heurística. Como se mencionó en la Sección 3.2, el objeto del puzzle es deslizar las fichas horizontalmente o verticalmente al espacio vacío hasta que la configuración empareje con la configuración objetivo (Figura 4.7).



El coste medio de la solución para los casos generados al azar del 8-puzzle son aproximadamente 22 pasos. El factor de ramificación es aproximadamente tres. (Cuando la ficha vacía está en el medio, hay cuatro movimientos posibles; cuando está en una esquina hay dos; y cuando está a lo largo de un borde hay tres.) Esto significa que una búsqueda exhaustiva a profundidad 22 miraría sobre $3^{22} \approx 3,1 \times 10^{10}$ estados. Manteniendo la pista de los estados repetidos, podríamos reducirlo a un factor de aproximadamente 170.000, porque hay sólo $9!/2 = 181.440$ estados distintos que son alcanzables.

(Véase el Ejercicio 3.4.) Esto es un número manejable, pero el número correspondiente para el puzzle-15 es aproximadamente 10^{13} , entonces lo siguiente es encontrar una buena función heurística. Si queremos encontrar las soluciones más cortas utilizando A*, necesitamos una función heurística que nunca sobreestima el número de pasos al objetivo. Hay una larga historia de tales heurísticas para el 15-puzzle; aquí están dos candidatas comúnmente usadas:

- h_1 = número de piezas mal colocadas. Para la Figura 4.7, las 8 piezas están fuera de su posición, así que el estado inicial tiene $h_1 = 8$. h_1 es una heurística admisible, porque está claro que cualquier pieza que está fuera de su lugar debe moverse por lo menos una vez.
- h_2 = suma de las distancias de las piezas a sus posiciones en el objetivo. Como las piezas no pueden moverse en diagonal, la distancia que contaremos será la suma de las distancias horizontales y verticales. Esto se llama a veces la **distancia en la ciudad** o **distancia de Manhattan**. h_2 es también admisible, porque cualquier movimiento que se puede hacer es mover una pieza un paso más cerca del objetivo. Las piezas 1 a 8 en el estado inicial nos dan una distancia de Manhattan de

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

Como era de esperar, ninguna sobreestima el coste solución verdadero, que es 26.

DISTANCIA DE
MANHATTAN

FACTOR DE
RAMIFICACIÓN EFICAZ

El efecto de la precisión heurística en el rendimiento

Una manera de caracterizar la calidad de una heurística es el b^* **factor de ramificación eficaz**. Si el número total de nodos generados por A* para un problema particular es N , y la profundidad de la solución es d , entonces b^* es el factor de ramificación que un árbol uniforme de profundidad d debería tener para contener $N + 1$ nodos. Así,

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

Por ejemplo, si A* encuentra una solución a profundidad cinco utilizando 52 nodos, entonces el factor de ramificación eficaz es 1,92. El factor de ramificación eficaz puede variar según los ejemplos del problema, pero por lo general es constante para problemas suficientemente difíciles. Por lo tanto, las medidas experimentales de b^* sobre un pequeño conjunto de problemas pueden proporcionar una buena guía para la utilidad total de la heurística. Una heurística bien diseñada tendría un valor de b^* cerca de 1, permitiría resolver problemas bastante grandes.

Para probar las funciones heurísticas h_1 y h_2 , generamos 1.200 problemas aleatorios con soluciones de longitudes de 2 a 24 (100 para cada número par) y los resolvemos con la búsqueda de profundidad iterativa y con la búsqueda en árbol A* usando tanto h_1 como h_2 . La Figura 4.8 nos da el número medio de nodos expandidos por cada estrategia y el factor de ramificación eficaz. Los resultados sugieren que h_2 es mejor que h_1 , y es mucho mejor que la utilización de la búsqueda de profundidad iterativa. Sobre nuestras soluciones con longitud 14, A* con h_2 es 30.000 veces más eficiente que la búsqueda no informada de profundidad iterativa.

Uno podría preguntarse si h_2 es siempre mejor que h_1 . La respuesta es sí. Es fácil ver de las definiciones de las dos heurísticas que, para cualquier nodo n , $h_2(n) \geq h_1(n)$.

DOMINACIÓN

Así decimos que h_2 **domina** a h_1 . La dominación se traslada directamente a la eficiencia: A* usando h_2 nunca expandirá más nodos que A* usando h_1 (excepto posiblemente para algunos nodos con $f(n) = C^*$). El argumento es simple. Recuerde la observación de la página 113 de que cada nodo con $f(n) < C^*$ será seguramente expandido. Esto es lo mismo que decir que cada nodo con $h(n) < C^* - g(n)$ será seguramente expandido. Pero debido a que h_2 es al menos tan grande como h_1 para todos los nodos, cada nodo que seguramente será expandido por la búsqueda A* con h_2 será seguramente también expandido con h_1 , y h_1 podría también hacer que otros nodos fueran expandidos. De ahí que es siempre mejor usar una función heurística con valores más altos, a condición de que no sobreestime y que el tiempo computacional de la heurística no sea demasiado grande.

d	Costo de la búsqueda			Factor de ramificación eficaz		
	BPI	A*(h_1)	A*(h_2)	BPI	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	73	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 Comparación de los costos de la búsqueda y factores de ramificación eficaces para la BÚSQUEDA-PROFOUNDIDAD-ITERATIVA y los algoritmos A* con h_1 y h_2 . Los datos son la media de 100 ejemplos del puzzle-8, para soluciones de varias longitudes.

Inventar funciones heurísticas admisibles

Hemos visto que h_1 (piezas más colocadas) y h_2 (distancia de Manhattan) son heurísticas bastante buenas para el 8-puzzle y que h_2 es mejor. ¿Cómo ha podido surgir h_2 ? ¿Es posible para un computador inventar mecánicamente tal heurística?

h_1 , h_2 son estimaciones de la longitud del camino restante para el 8-puzzle, pero también son longitudes de caminos absolutamente exactos para versiones *simplificadas* del puzzle. Si se cambiaron las reglas del puzzle de modo que una ficha pudiera moverse a todas partes, en vez de solamente al cuadrado adyacente vacío, entonces h_1 daría el número exacto de pasos en la solución más corta. Del mismo modo, si una ficha pudiera moverse a un cuadrado en cualquier dirección, hasta en un cuadrado ocupado, entonces h_2 daría el número exacto de pasos en la solución más corta. A un problema con menos restricciones en las acciones se le llama **problema relajado**. El costo de una solución óptima en un problema relajado es una heurística admisible para el problema original.



La heurística es admisible porque la solución óptima en el problema original es, por definición, también una solución en el problema relajado y por lo tanto debe ser al menos tan cara como la solución óptima en el problema relajado. Como la heurística obtenida es un costo exacto para el problema relajado, debe cumplir la desigualdad triangular y es por lo tanto **consistente** (véase la página 113).

Si la definición de un problema está escrita en un lenguaje formal, es posible construir problemas relajados automáticamente⁶. Por ejemplo, si las acciones del 8-puzzle están descritas como

Una ficha puede moverse del cuadrado A al cuadrado B si

A es horizontalmente o verticalmente adyacente a B y B es la vacía

podemos generar tres problemas relajados quitando una o ambas condiciones:

- (a) Una ficha puede moverse del cuadrado A al cuadrado B si A es adyacente a B.
- (b) Una ficha puede moverse del cuadrado A al cuadrado B si B es el vacío.
- (c) Una ficha puede moverse del cuadrado A al cuadrado B.

De (a), podemos obtener h_2 (distancia de Manhattan). El razonamiento es que h_2 sería el resultado apropiado si moviéramos cada ficha en dirección a su destino. La heurística obtenida de (b) se discute en el Ejercicio 4.9. De (c), podemos obtener h_1 (fichas mal colocadas), porque sería el resultado apropiado si las fichas pudieran moverse a su destino en un paso. Notemos que es crucial que los problemas relajados generados por esta técnica puedan resolverse esencialmente sin búsqueda, porque las reglas relajadas permiten que el problema sea descompuesto en ocho subproblemas independientes. Si el problema relajado es difícil de resolver, entonces los valores de la correspondencia heurística serán costosos de obtener⁷.

Un programa llamado ABSOLVER puede generar heurísticas automáticamente a partir de las definiciones del problema, usando el método del «problema relajado» y otras técnicas (Prieditis, 1993). ABSOLVER generó una nueva heurística para el 8-puzzle mejor que cualquier heurística y encontró el primer heurístico útil para el famoso puzzle cubo de Rubik.

Un problema con la generación de nuevas funciones heurísticas es que a menudo se falla al conseguir una heurística «claramente mejor». Si tenemos disponible un conjunto de heurísticas admisibles h_1, \dots, h_m para un problema, y ninguna de ellas domina a las demás, ¿qué deberíamos elegir? No tenemos por qué hacer una opción. Podemos tener lo mejor de todas, definiendo

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

Esta heurística compuesta usando cualquier función es más exacta sobre el nodo en cuestión. Como las heurísticas componentes son admisibles, h es admisible; es tam-

⁶ En los capítulos 8 y 11, describiremos lenguajes formales convenientes para esta tarea; con descripciones formales que puedan manipularse, puede automatizarse la construcción de problemas relajados. Por el momento, usaremos el castellano.

⁷ Note que una heurística perfecta puede obtenerse simplemente permitiendo a h ejecutar una búsqueda primero en anchura «a escondidas». Así, hay una compensación entre exactitud y tiempo de cálculo para las funciones heurísticas.

bién fácil demostrar que h es consistente. Además, h domina a todas sus heurísticas componentes.

SUBPROBLEMA

También se pueden obtener heurísticas admisibles del coste de la solución de un **subproblema** de un problema dado. Por ejemplo, la Figura 4.9 muestra un subproblema del puzzle-8 de la Figura 4.7. El subproblema implica la colocación de las fichas 1, 2, 3, 4 en sus posiciones correctas. Claramente, el coste de la solución óptima de este subproblema es una cota inferior sobre el coste del problema completo. Parece ser considerablemente más exacta que la distancia de Manhattan, en algunos casos.

MODELO DE BASES DE DATOS

La idea que hay detrás del **modelo de bases de datos** es almacenar estos costos exactos de las soluciones para cada posible subproblema (en nuestro ejemplo, cada configuración posible de las cuatro fichas y el vacío; note que las posiciones de las otras cuatro fichas son irrelevantes para los objetivos de resolver el subproblema, pero los movimientos de esas fichas cuentan realmente hacia el coste). Entonces, calculamos una heurística admisible h_{BD} , para cada estado completo encontrado durante una búsqueda, simplemente mirando la configuración del subproblema correspondiente en la base de datos. La base de datos se construye buscando hacia atrás desde el estado objetivo y registrando el coste de cada nuevo modelo encontrado; el gasto de esta búsqueda se amortiza sobre los siguientes problemas.

La opción de 1-2-3-4 es bastante arbitraria; podríamos construir también bases de datos para 5-6-7-8, y para 2-4-6-8, etcétera. Cada base de datos produce una heurística admisible, y esta heurística puede combinarse, como se explicó antes, tomando el valor máximo. Una heurística combinada de esta clase es mucho más exacta que la distancia de Manhattan; el número de nodos generados, resolviendo 15-puzzles aleatorios, puede reducirse en un factor de 1.000.

Uno podría preguntarse si las heurísticas obtenidas de las bases de datos 1-2-3-4 y 5-6-7-8 podrían sumarse, ya que los dos subproblemas parecen no superponerse. ¿Esto daría aún una heurística admisible? La respuesta es no, porque las soluciones del subproblema 1-2-3-4 y del subproblema 5-6-7-8 para un estado compartirán casi seguramente algunos movimientos (es improbable que 1-2-3-4 pueda colocarse en su lugar sin tocar 5-6-7-8, y *viceversa*). ¿Pero y si no contamos estos movimientos? Es decir no registramos el costo total para resolver el problema 1-2-3-4, sino solamente el número de mo-

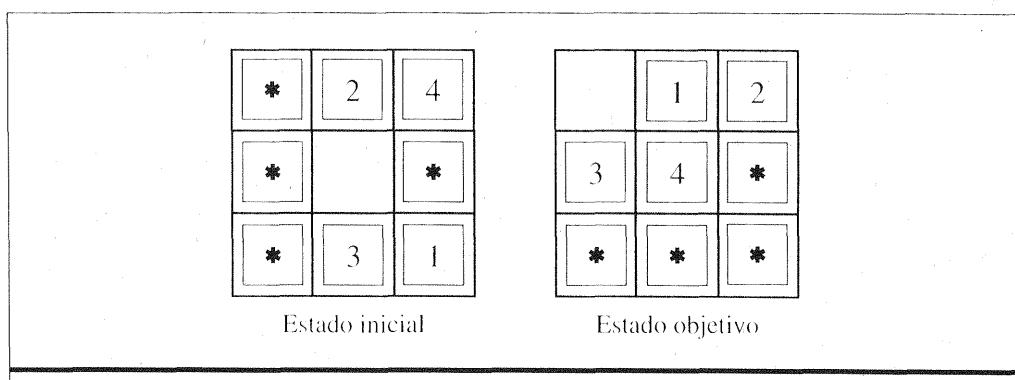


Figura 4.9 Un subproblema del puzzle-8 dado en la Figura 4.7. La tarea es conseguir las fichas 1, 2, 3 y 4 en sus posiciones correctas, sin preocuparse de lo que le pasan a las otras fichas.

vimientos que implican 1-2-3-4. Entonces es fácil ver que la suma de los dos costos todavía es una cota inferior del costo de resolver el problema entero. Esta es la idea que hay detrás del **modelo de bases de datos disjuntas**. Usando tales bases de datos, es posible resolver puzzles-15 aleatorios en milisegundos (el número de nodos generados se reduce en un factor de 10.000 comparado con la utilización de la distancia de Manhattan). Para puzzles-24, se puede obtener una aceleración de aproximadamente un millón.

El modelo de bases de datos disjuntas trabajan para puzzles de deslizamiento de fichas porque el problema puede dividirse de tal modo que cada movimiento afecta sólo a un subproblema, ya que sólo se mueve una ficha a la vez. Para un problema como el cubo de Rubik, esta clase de subdivisión no puede hacerse porque cada movimiento afecta a ocho o nueve de los 25 cubos. Actualmente, no está claro cómo definir bases de datos disjuntas para tales problemas.

Aprendizaje de heurísticas desde la experiencia

Una función heurística $h(n)$, como se supone, estima el costo de una solución que comienza desde el estado en el nodo n . ¿Cómo podría un agente construir tal función? Se dio una solución en la sección anterior (idear problemas relajados para los cuales puede encontrarse fácilmente una solución óptima). Otra solución es aprender de la experiencia. «La experiencia» aquí significa la solución de muchos 8-puzzles, por ejemplo. Cada solución óptima en un problema del 8-puzzle proporciona ejemplos para que pueda aprender la función $h(n)$. Cada ejemplo se compone de un estado del camino solución y el costo real de la solución desde ese punto. A partir de estos ejemplos, se puede utilizar un algoritmo de **aprendizaje inductivo** para construir una función $h(n)$ que pueda (con suerte) predecir los costos solución para otros estados que surjan durante la búsqueda. Las técnicas para hacer esto utilizando redes neuronales, árboles de decisión, y otros métodos, se muestran en el Capítulo 18 (los métodos de aprendizaje por refuerzo, también aplicables, serán descritos en el Capítulo 21).

Los métodos de aprendizaje inductivos trabajan mejor cuando se les suministran **características** de un estado que sean relevantes para su evaluación, más que sólo la descripción del estado. Por ejemplo, la característica «número de fichas mal colocadas» podría ser útil en la predicción de la distancia actual de un estado desde el objetivo. Llamemos a esta característica $x_1(n)$. Podríamos tomar 100 configuraciones del 8-puzzle generadas aleatoriamente y unir las estadísticas de sus costos de la solución actual. Podríamos encontrar que cuando $x_1(n)$ es cinco, el coste medio de la solución está alrededor de 14, etcétera. Considerando estos datos, el valor de x_1 puede usarse para predecir $h(n)$. Desde luego, podemos usar varias características. Una segunda característica $x_2(n)$ podría ser «el número de pares de fichas adyacentes que son también adyacentes en el estado objetivo». ¿Cómo deberían combinarse $x_1(n)$ y $x_2(n)$ para predecir $h(n)$? Una aproximación común es usar una combinación lineal:

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

Las constantes c_1 y c_2 se ajustan para dar el mejor ajuste a los datos reales sobre los costos de la solución. Presumiblemente, c_1 debe ser positivo y c_2 debe ser negativo.

4.3 Algoritmos de búsqueda local y problemas de optimización

Los algoritmos de búsqueda que hemos visto hasta ahora se diseñan para explorar sistemáticamente espacios de búsqueda. Esta forma sistemática se alcanza manteniendo uno o más caminos en memoria y registrando qué alternativas se han explorado en cada punto a lo largo del camino y cuáles no. Cuando se encuentra un objetivo, el *camino* a ese objetivo también constituye una *solución* al problema.

En muchos problemas, sin embargo, el camino al objetivo es irrelevante. Por ejemplo, en el problema de las 8-reinas (véase la página 74), lo que importa es la configuración final de las reinas, no el orden en las cuales se añaden. Esta clase de problemas incluyen muchas aplicaciones importantes como diseño de circuitos integrados, disposición del suelo de una fábrica, programación del trabajo en tiendas, programación automática, optimización de redes de telecomunicaciones, dirigir un vehículo, y la gestión de carteras.

Si no importa el camino al objetivo, podemos considerar una clase diferente de algoritmos que no se preocupen en absoluto de los caminos. Los algoritmos de **búsqueda local** funcionan con un solo **estado actual** (más que múltiples caminos) y generalmente se mueve sólo a los vecinos del estado. Típicamente, los caminos seguidos por la búsqueda no se retienen. Aunque los algoritmos de búsqueda local no son sistemáticos, tienen dos ventajas claves: (1) usan muy poca memoria (por lo general una cantidad constante); y (2) pueden encontrar a menudo soluciones razonables en espacios de estados grandes o infinitos (continuos) para los cuales son inadecuados los algoritmos sistemáticos.

BÚSQUEDA LOCAL
ESTADO ACTUAL

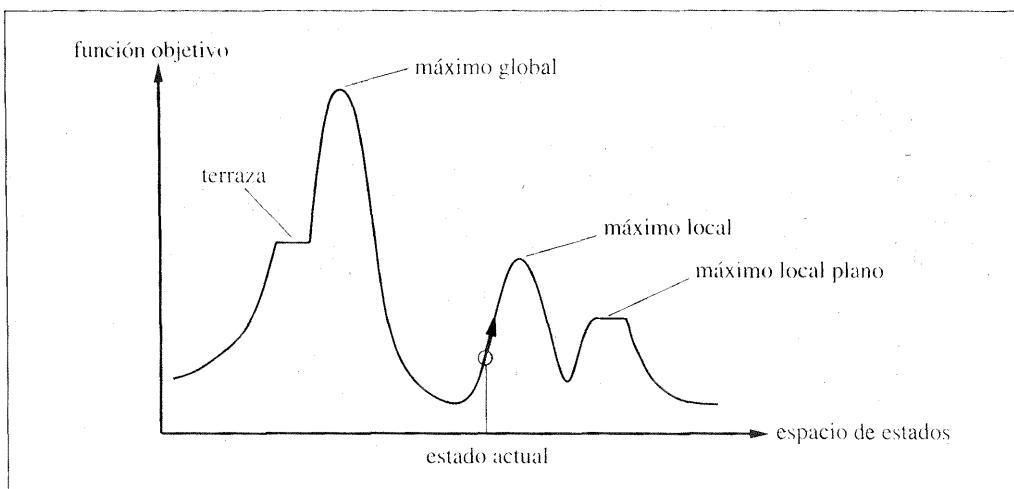


Figura 4.10 Un paisaje del espacio de estados unidimensional en el cual la elevación corresponde a la función objetivo. El objetivo es encontrar el máximo global. La búsqueda de ascensión de colinas modifica el estado actual para tratar de mejorarlo, como se muestra con la flecha. Se definen en el texto varios rasgos topográficos.

**PROBLEMAS DE
OPTIMIZACIÓN**

FUNCIÓN OBJETIVO

**PAISAJE DEL ESPACIO
DE ESTADOS**

MÍNIMO GLOBAL

MÁXIMO GLOBAL

**ASCENSIÓN DE
COLINAS**

Además de encontrar los objetivos, los algoritmos de búsqueda local son útiles para resolver **problemas de optimización** puros, en los cuales el objetivo es encontrar el mejor estado según una **función objetivo**. Muchos problemas de optimización no encajan en el modelo «estándar» de búsqueda introducido en el Capítulo 3. Por ejemplo, la naturaleza proporciona una función objetivo (idoneidad o salud reproductiva) que la evolución Darwiniana intenta optimizar, pero no hay ningún «test objetivo» y ningún «coste de camino» para este problema.

Para entender la búsqueda local, encontraremos muy útil considerar la forma o el **paisaje del espacio de estados** (como en la Figura 4.10). El paisaje tiene «posición» (definido por el estado) y «elevación» (definido por el valor de la función de coste heurística o función objetivo). Si la elevación corresponde al costo, entonces el objetivo es encontrar el valle más bajo (un **mínimo global**); si la elevación corresponde a una función objetivo, entonces el objetivo es encontrar el pico más alto (un **máximo global**). (Puedes convertir uno en el otro solamente al insertar un signo menos.) Los algoritmos de búsqueda local exploran este paisaje. Un algoritmo de búsqueda local completo siempre encuentra un objetivo si existe; un algoritmo óptimo siempre encuentran un mínimo/máximo global.

Búsqueda de ascensión de colinas

En la Figura 4.11 se muestra el algoritmo de búsqueda de **ascensión de colinas**. Es simplemente un bucle que continuamente se mueve en dirección del valor creciente, es decir, cuesta arriba. Termina cuando alcanza «un pico» en donde ningún vecino tiene un valor más alto. El algoritmo no mantiene un árbol de búsqueda, sino una estructura de datos del nodo actual que necesita sólo el registro del estado y su valor de función objetivo. La ascensión de colinas no mira delante más allá de los vecinos inmediatos del estado actual. Se parece a lo que hacemos cuando tratamos de encontrar la cumbre del Everest en una niebla mientras sufrimos amnesia.

Para ilustrar la ascensión de colinas, usaremos el **problema de las 8-reinas** introducido en la página 74. Los algoritmos de búsqueda local típicamente usan una **formula-**

función ASCENSION-COLINAS(*problema*) **devuelve** un estado que es un máximo local

entradas: *problema*, un problema

variables locales: *actual*, un nodo

vecino, un nodo

actual \leftarrow HACER-NODO(ESTADO-INICIAL[*problema*])

bucle hacer

vecino \leftarrow sucesor de valor más alto de *actual*

si VALOR[*vecino*] \leq VALOR[*actual*] **entonces devolver** ESTADO[*actual*]

actual \leftarrow *vecino*

Figura 4.11 El algoritmo de búsqueda ascensión de colinas (la versión de subida más rápida), que es la técnica de búsqueda local más básica. En cada paso el nodo actual se sustituye por el mejor vecino: en esta versión, el vecino con el VALOR más alto, pero si se utiliza una heurística h de estimación de costos, sería el vecino con h más bajo.

ción de estados completa, donde cada estado tiene a ocho reinas sobre el tablero, una por columna. La función sucesor devuelve todos los estados posibles generados moviendo una reina a otro cuadrado en la misma columna (entonces cada estado tiene $8 \times 7 = 56$ sucesores). La función de costo heurística h es el número de pares de reinas que se atan la una a la otra, directa o indirectamente. El mínimo global de esta función es cero, que ocurre sólo en soluciones perfectas. La Figura 4.12(a) muestra un estado con $h = 17$. La figura también muestra los valores de todos sus sucesores, con los mejores sucesores que tienen $h = 12$. Los algoritmos de ascensión de colinas eligen típicamente al azar entre el conjunto de los mejores sucesores, si hay más de uno.

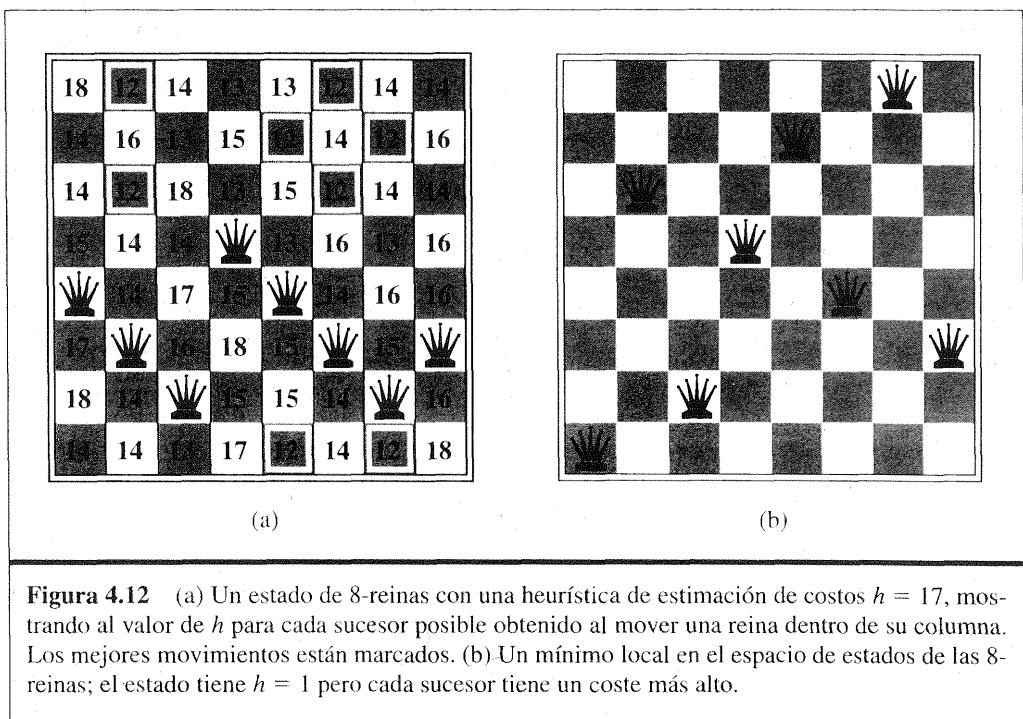


Figura 4.12 (a) Un estado de 8-reinas con una heurística de estimación de costos $h = 17$, mostrando al valor de h para cada sucesor posible obtenido al mover una reina dentro de su columna. Los mejores movimientos están marcados. (b) Un mínimo local en el espacio de estados de las 8-reinas; el estado tiene $h = 1$ pero cada sucesor tiene un coste más alto.

BÚSQUEDA LOCAL VORAZ

A veces a la ascensión de colinas se le llama **búsqueda local voraz** porque toma un estado vecino bueno sin pensar hacia dónde ir después. Aunque la avaricia sea considerada uno de los siete pecados mortales, resulta que los algoritmos avaros a menudo funcionan bastante bien. La ascensión de colinas a menudo hace el progreso muy rápido hacia una solución, porque es por lo general bastante fácil mejorar un estado malo. Por ejemplo, desde el estado de la Figura 4.12(a), se realizan solamente cinco pasos para alcanzar el estado de la Figura 4.12(b), que tiene $h = 1$ y es casi una solución. Lamentablemente, la ascensión de colinas a menudo se atasca por los motivos siguientes:

- **Máximo local:** un máximo local es un pico que es más alto que cada uno de sus estados vecinos, pero más abajo que el máximo global. Los algoritmos de ascensión de colinas que alcanzan la vecindad de un máximo local irán hacia el pico, pero entonces se atascarán y no podrán ir a ninguna otra parte. La Figura 4.10 ilustra el problema esquemáticamente. Más concretamente, el estado en la Figura 4.12(b) es

TERRAZA

MOVIMIENTO
LATERAL

de hecho un máximo local (es decir, un mínimo local para el coste h); cada movimiento de una sola reina hace la situación peor.

- **Crestas:** la Figura 4.13 muestra una cresta. Las crestas causan una secuencia de máximos locales que hace muy difícil la navegación para los algoritmos avaros.
- **Meseta:** una meseta es un área del paisaje del espacio de estados donde la función de evaluación es plana. Puede ser un máximo local plano, del que no existe ninguna salida ascendente, o una **terrazza**, por la que se pueda avanzar (véase la Figura 4.10). Una búsqueda de ascensión de colinas podría ser incapaz de encontrar su camino en la meseta.

En cada caso, el algoritmo alcanza un punto en el cual no se puede hacer ningún progreso. Comenzando desde un estado de las ocho reinas generado aleatoriamente, la ascensión de colinas por la zona más escarpada se estanca el 86 por ciento de las veces, y resuelve sólo el 14 por ciento de los problemas. Trabaja rápidamente, usando solamente cuatro pasos por regla general cuando tiene éxito y tres cuando se estanca (no está mal para un espacio de estados con $8^8 \approx 17$ millones de estados).

El algoritmo de la Figura 4.11 se para si alcanza una meseta donde el mejor sucesor tiene el mismo valor que el estado actual. ¿Podría ser una buena idea no continuar (y permitir un **movimiento lateral** con la esperanza de que la meseta realmente sea una terraza, como se muestra en la Figura 4.10)? La respuesta es por lo general sí, pero debemos tener cuidado. Si siempre permitimos movimientos laterales cuando no hay ningún movimiento ascendente, va a ocurrir un bucle infinito siempre que el algoritmo alcance un máximo local plano que no sea una terraza. Una solución común es poner un límite sobre el número de movimientos consecutivos laterales permitidos. Por ejemplo, podríamos permitir hasta, digamos, 100 movimientos laterales consecutivos en el problema de las ocho reinas. Esto eleva el porcentaje de casos de problemas resueltos por la ascensión de

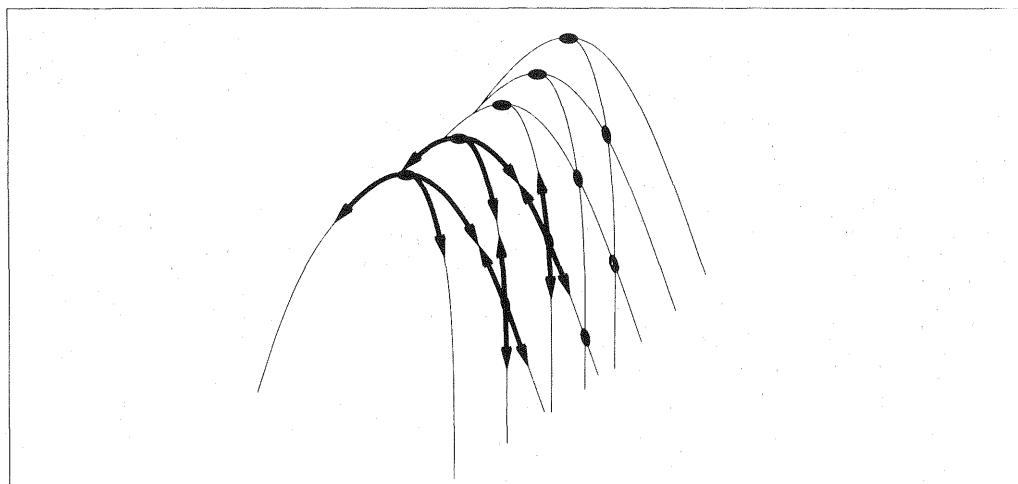


Figura 4.13 Ilustración de por qué las crestas causan dificultades para la ascensión de colinas. La rejilla de estados (círculos oscuros) se pone sobre una cresta que se eleva de izquierda a derecha y crea una secuencia de máximos locales que no están directamente relacionados el uno con el otro. De cada máximo local, todas las acciones disponibles se señalan cuesta abajo.

colinas del 14 al 94 por ciento. El éxito viene con un coste: el algoritmo hace en promedio aproximadamente 21 pasos para cada caso satisfactorio y 64 para cada fracaso.

Se han inventado muchas variantes de la ascensión de colinas. La **ascensión de colinas estocástica** escoge aleatoriamente de entre los movimientos ascendentes; la probabilidad de selección puede variar con la pendiente del movimiento ascendente. Éste por lo general converge más despacio que la subida más escarpada, pero en algunos paisajes de estados encuentra mejores soluciones. La **ascensión de colinas de primera opción** implementa una ascensión de colinas estocástica generando sucesores al azar hasta que se genera uno que es mejor que el estado actual. Esta es una buena estrategia cuando un estado tiene muchos (por ejemplo, miles) sucesores. El ejercicio 4.16 le pide investigar esto.

Los algoritmos de ascensión de colinas descritos hasta ahora son incompletos, a menudo dejan de encontrar un objetivo, cuando éste existe, debido a que pueden estancarse sobre máximos locales. La **ascensión de colinas de reinicio aleatorio** adopta el refrán conocido, «si al principio usted no tiene éxito, intente, intente otra vez». Esto conduce a una serie de búsquedas en ascensión de colinas desde los estados iniciales generados aleatoriamente⁸, parándose cuando se encuentra un objetivo. Es completa con probabilidad acercándose a 1, por la razón trivial de que generará finalmente un estado objetivo como el estado inicial. Si cada búsqueda por ascensión de colinas tiene una probabilidad p de éxito, entonces el número esperado de reinicios requerido es $1/p$. Para ejemplos de ocho reinas sin permitir movimientos laterales, $p \approx 0,14$, entonces necesitamos aproximadamente siete iteraciones para encontrar un objetivo (seis fracasos y un éxito). El número esperado de pasos es el coste de una iteración acertada más $(1 - p)/p$ veces el coste de fracaso, o aproximadamente 22 pasos. Cuando permitimos movimientos laterales, son necesarios $1/0,94 \approx 1,06$ iteraciones por regla general y $(1 \times 21) + (0,06/0,94) \times 64 \approx 25$ pasos. Para las ocho reinas, entonces, la ascensión de colinas de reinicio aleatorio es muy eficaz. Incluso para tres millones de reinas, la aproximación puede encontrar soluciones en menos de un minuto⁹.

El éxito de la ascensión de colinas depende muchísimo de la forma del paisaje del espacio de estados: si hay pocos máximos locales y mesetas, la ascensión de colinas con reinicio aleatorio encontrará una solución buena muy rápidamente. Por otro lado, muchos problemas reales tienen un paisaje que parece más bien una familia de puerco espines sobre un suelo llano, con puerco espines en miniatura que viven en la punta de cada aguja del puerco espín, y así *indefinidamente*. Los problemas NP-duros típicamente tienen un número exponencial de máximos locales. A pesar de esto, un máximo local, razonablemente bueno, a menudo se puede encontrar después de un número pequeño de reinicios.

Búsqueda de temple simulado

Un algoritmo de ascensión de colinas que nunca hace movimientos «cuesta abajo» hacia estados con un valor inferior (o coste más alto) garantiza ser incompleto, porque puede

⁸ La generación de un estado *aleatorio* de un espacio de estados especificado implícitamente puede ser un problema difícil en sí mismo.

⁹ Luby *et al.* (1993) demuestran que es mejor, a veces, reiniciar un algoritmo de búsqueda aleatoria después de cierta cantidad fija de tiempo y que puede ser mucho más eficiente que permitir que continúe la búsqueda indefinidamente. Rechazar o limitar el número de movimientos laterales es un ejemplo de esto.

TEMPLE SIMULADO

GRADIENTE DESCENDENTE

estancarse en un máximo local. En contraste, un camino puramente aleatorio, es decir moviéndose a un sucesor elegido uniformemente aleatorio de un conjunto de sucesores, es completo, pero sumamente ineficaz. Por lo tanto, parece razonable intentar combinar la ascensión de colinas con un camino aleatorio de algún modo que produzca tanto eficacia como completitud. El **temple simulado** es ese algoritmo. En metalurgia, el **temple** es el proceso utilizado para templar o endurecer metales y cristales calentándolos a una temperatura alta y luego gradualmente enfriarlos, así permite al material fundirse en un estado cristalino de energía baja. Para entender el temple simulado, cambiemos nuestro punto de vista de la ascensión de colinas al **gradiente descendente** (es decir, minimizando el coste) e imaginemos la tarea de colocar una pelota de ping-póng en la grieta más profunda en una superficie desigual. Si dejamos solamente rodar a la pelota, se parará en un mínimo local. Si sacudimos la superficie, podemos echar la pelota del mínimo local. El truco es sacudir con bastante fuerza para echar la pelota de mínimos locales, pero no lo bastante fuerte para desalojarlo del mínimo global. La solución del temple simulado debe comenzar sacudiendo con fuerza (es decir, a una temperatura alta) y luego gradualmente reducir la intensidad de la sacudida (es decir, a más baja temperatura).

El bucle interno del algoritmo del temple simulado (Figura 4.14) es bastante similar a la ascensión de colinas. En vez de escoger el *mejor* movimiento, sin embargo, escoge un movimiento *aleatorio*. Si el movimiento mejora la situación, es siempre aceptado. Por otra parte, el algoritmo acepta el movimiento con una probabilidad menor que uno. La probabilidad se disminuye exponencialmente con la «maldad» de movimiento (la cantidad ΔE por la que se empeora la evaluación). La probabilidad también disminuye cuando «la temperatura» T baja: los «malos» movimientos son más probables al comienzo cuando la temperatura es alta, y se hacen más improbables cuando T disminuye. Uno puede demostrar que si el *esquema* disminuye T bastante despacio, el algoritmo encontrará un óptimo global con probabilidad cerca de uno.

función TEMPLE-SIMULADO(*problema, esquema*) **devuelve** un estado solución

entradas: *problema*, un problema

esquema, una aplicación desde el tiempo a «temperatura»

variables locales: *actual*, un nodo

siguiente, un nodo

T, una «temperatura» controla la probabilidad de un paso hacia abajo

actual \leftarrow HACER-NODO(ESTADO-INICIAL{*problema*})

para *t* \leftarrow 1 a ∞ **hacer**

T \leftarrow *esquema*[*t*]

si *T* = 0 **entonces devolver** *actual*

siguiente \leftarrow un sucesor seleccionado aleatoriamente de *actual*

$\Delta E \leftarrow$ VALOR[*siguiente*] - VALOR[*actual*]

si $\Delta E > 0$ **entonces** *actual* \leftarrow *siguiente*

en caso contrario *actual* \leftarrow *siguiente* sólo con probabilidad $e^{\Delta E/T}$

Figura 4.14 Algoritmo de búsqueda de temple simulado, una versión de la ascensión de colinas estocástico donde se permite descender a algunos movimientos. Los movimientos de descenso se aceptan fácilmente al comienzo en el programa de templadura y luego menos, conforme pasa el tiempo. La entrada del *esquema* determina el valor de *T* como una función de tiempo.

A principios de los años 80, el temple simulado fue utilizado ampliamente para resolver problemas de distribución VLSI. Se ha aplicado ampliamente a programación de una fábrica y otras tareas de optimización a gran escala. En el Ejercicio 4.16, le pedimos que compare su funcionamiento con el de la ascensión de colinas con reinicio aleatorio sobre el puzzle de las n -reinas.

BÚSQUEDA POR HAZ LOCAL

Guardar solamente un nodo en la memoria podría parecer una reacción extrema para el problema de limitaciones de memoria. El algoritmo¹⁰ de **búsqueda por haz local** guarda la pista de k estados (no sólo uno). Comienza con estados generados aleatoriamente. En cada paso, se generan todos los sucesores de los k estados. Si alguno es un objetivo, paramos el algoritmo. Por otra parte, se seleccionan los k mejores sucesores de la lista completa y repetimos.

A primera vista, una búsqueda por haz local con k estados podría parecerse a ejecutar k reinicios aleatorios en paralelo en vez de en secuencia. De hecho, los dos algoritmos son bastante diferentes. En una búsqueda de reinicio aleatorio, cada proceso de búsqueda se ejecuta independientemente de los demás. *En una búsqueda por haz local, la información útil es pasada entre los k hilos paralelos de búsqueda.* Por ejemplo, si un estado genera varios sucesores buenos y los otros $k - 1$ estados generan sucesores malos, entonces el efecto es que el primer estado dice a los demás, «¡Venid aquí, la hierba es más verde!» El algoritmo rápidamente abandona las búsquedas infructuosas y mueve sus recursos a donde se hace la mayor parte del progreso.

En su forma más simple, la búsqueda de haz local puede sufrir una carencia de diversidad entre los k estados (se pueden concentrar rápidamente en una pequeña región del espacio de estados, haciendo de la búsqueda un poco más que una versión cara de la ascensión de colinas). Una variante llamada **búsqueda de haz estocástica**, análoga a la ascensión de colinas estocástica, ayuda a aliviar este problema. En vez de elegir los k mejores del conjunto de sucesores candidatos, la búsqueda de haz estocástica escoge a k sucesores aleatoriamente, con la probabilidad de elegir a un sucesor como una función creciente de su valor. La búsqueda de haz estocástica muestra algún parecido con el proceso de selección natural, por lo cual los «sucesores» (descendientes) de un «estado» (organismo) pueblan la siguiente generación según su «valor» (idoneidad o salud).

Algoritmos genéticos

Un **algoritmo genético** (o AG) es una variante de la búsqueda de haz estocástica en la que los estados sucesores se generan combinando *dos* estados padres, más que modificar un solo estado. La analogía a la selección natural es la misma que con la búsqueda de haz estocástica, excepto que ahora tratamos con reproducción sexual más que con la reproducción asexual.

¹⁰ Búsqueda por haz local es una adaptación de la **búsqueda de haz**, que es un algoritmo basado en camino.



BÚSQUEDA DE HAZ ESTOCÁSTICA

ALGORITMO GENÉTICO

POBLACIÓN
INDIVIDUO

FUNCIÓN IDONEIDAD
CRUCE

Como en la búsqueda de haz, los AGs comienzan con un conjunto de k estados generados aleatoriamente, llamados **población**. Cada estado, o **individuo**, está representado como una cadena sobre un alfabeto finito (el más común, una cadenas de 0s y 1s). Por ejemplo, un estado de las ocho reinas debe especificar las posiciones de las ocho reinas, cada una en una columna de ocho cuadrados, y se requieren $8 \times \log_2 8 = 24$ bits. O bien, el estado podría representarse como ocho dígitos, cada uno en el rango de uno a ocho (veremos más tarde que las dos codificaciones se comportan de forma diferente). La Figura 4.15(a) muestra una población de cuatro cadenas de ocho dígitos que representan estados de ocho reinas.

En la Figura 4.15(b)-(e) se muestra la producción de la siguiente generación de estados. En (b) cada estado se tasa con la función de evaluación o (en terminología AG) la **función idoneidad**. Una función de idoneidad debería devolver valores más altos para estados mejores, así que, para el problema de las 8-reinas utilizaremos el número de pares de reinas no atacadas, que tiene un valor de 28 para una solución. Los valores de los cuatro estados son 24, 23, 20 y 11. En esta variante particular del algoritmo genético, la probabilidad de ser elegido para la reproducción es directamente proporcional al resultado de idoneidad, y los porcentajes se muestran junto a los tanteos.

En (c), se seleccionan dos pares, de manera aleatoria, para la reproducción, de acuerdo con las probabilidades en (b). Notemos que un individuo se selecciona dos veces y uno ninguna¹¹. Para que cada par se aparee, se elige aleatoriamente un punto de **cruce** de las posiciones en la cadena. En la Figura 4.15 los puntos de cruce están después del tercer dígito en el primer par y después del quinto dígito en el segundo par¹².

En (d), los descendientes se crean cruzando las cadenas paternales en el punto de cruce. Por ejemplo, el primer hijo del primer par consigue los tres primeros dígitos del

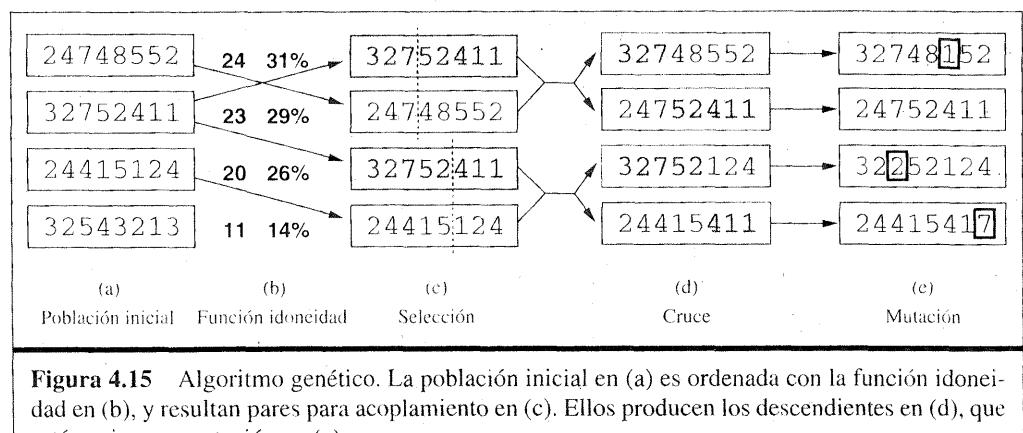
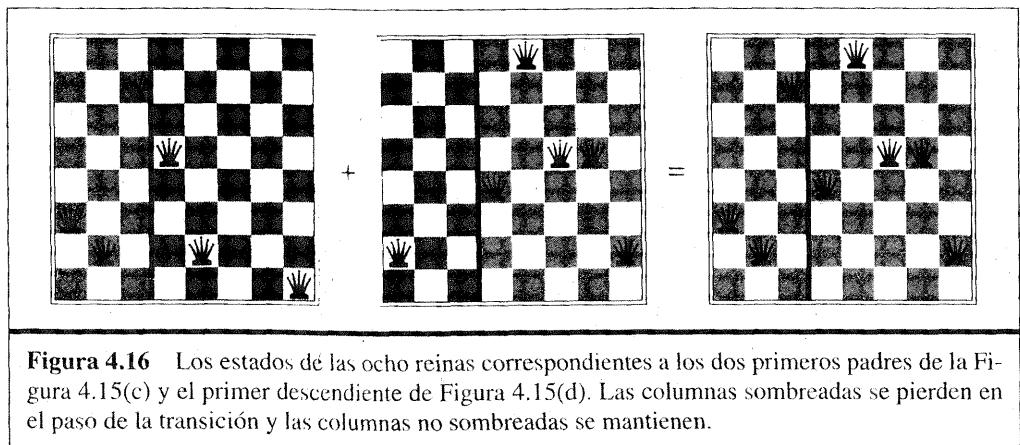


Figura 4.15 Algoritmo genético. La población inicial en (a) es ordenada con la función idoneidad en (b), y resultan pares para acoplamiento en (c). Ellos producen los descendientes en (d), que están sujetos a mutación en (e).

¹¹ Hay muchas variantes de esta regla de selección. Puede demostrarse que el método **selectivo**, en el que se desechan todos los individuos debajo de un umbral dado, converge más rápido que la versión aleatoria (Basum *et al.*, 1995).

¹² Son aquí los asuntos de codificación. Si se usa una codificación de 24 bit en vez de ocho dígitos, entonces el punto de cruce tiene 2/3 de posibilidad de estar en medio de un dígito, que resulta en una mutación esencialmente arbitraria de ese dígito.

primer padre y los dígitos restantes del segundo padre, mientras que el segundo hijo consigue los tres primeros dígitos del segundo padre y el resto del primer padre. En la Figura 4.16 se muestran los estados de las ocho reinas implicados en este paso de reproducción. El ejemplo ilustra el hecho de que, cuando dos estados padres son bastante diferentes, la operación de cruce puede producir un estado que está lejos de cualquiera de los estados padre. Esto es, a menudo, lo que ocurre al principio del proceso en el que la población es bastante diversa, así que el cruce (como en el temple simulado) con frecuencia realiza pasos grandes, al principio, en el espacio de estados en el proceso de búsqueda y pasos más pequeños, más tarde, cuando la mayor parte de individuos son bastante similares.



MUTACIÓN

Finalmente, en (e), cada posición está sujeta a la **mutación** aleatoria con una pequeña probabilidad independiente. Un dígito fue transformado en el primer, tercero, y cuarto descendiente. El problema de las 8-reinas corresponde a escoger una reina aleatoriamente y moverla a un cuadrado aleatorio en su columna. La figura 4.17 describe un algoritmo que implementa todos estos pasos.

Como en la búsqueda por haz estocástica, los algoritmos genéticos combinan una tendencia ascendente con exploración aleatoria y cambian la información entre los hijos paralelos de búsqueda. La ventaja primera, si hay alguna, del algoritmo genético viene de la operación de cruce. Aún puede demostrarse matemáticamente que, si las posiciones del código genético se permutan al principio en un orden aleatorio, el cruce no comunica ninguna ventaja. Intuitivamente, la ventaja viene de la capacidad del cruce para combinar bloques grandes de letras que han evolucionado independientemente para así realizar funciones útiles, de modo que se aumente el nivel de granularidad en el que funciona la búsqueda. Por ejemplo, podría ser que poner las tres primeras reinas en posiciones 2, 4 y 6 (donde ellas no se atacan las unas a las otras) constituya un bloque útil que pueda combinarse con otros bloques para construir una solución.

ESQUEMA

La teoría de los algoritmos genéticos explica cómo esta teoría trabaja utilizando la idea de un **esquema**, una subcadena en la cual algunas de las posiciones se pueden dejar inespecíficas. Por ejemplo, el esquema 246***** describe todos los estados de

función ALGORITMO-GENÉTICO(*población*, IDONEIDAD) **devuelve** un individuo

entradas: *población*, un conjunto de individuos

IDONEIDAD, una función que mide la capacidad de un individuo

repetir

nueva_población \leftarrow conjunto vacío

bucle para *i* **desde** 1 **hasta** TAMAÑO(*población*) **hacer**

x \leftarrow SELECCIÓN-ALEATORIA(*población*, IDONEIDAD)

y \leftarrow SELECCIÓN-ALEATORIA(*población*, IDONEIDAD)

hijo \leftarrow REPRODUCIR(*x,y*)

si (probabilidad aleatoria pequeña) **entonces** *hijo* \leftarrow MUTAR(*hijo*)

añadir *hijo* a *nueva_población*

población \leftarrow *nueva_población*

hasta que algún individuo es bastante adecuado, o ha pasado bastante tiempo

devolver el mejor individuo en la *población*, de acuerdo con la IDONEIDAD

función REPRODUCIR(*x,y*) **devuelve** un individuo

entradas: *x,y*, padres individuales

n \leftarrow LONGITUD(*x*)

c \leftarrow número aleatorio de 1 a *n*

devolver AÑADIR(SUBCADENA(*x*, 1, *c*),SUBCADENA(*y*, *c* + 1, *n*))

Figura 4.17 Algoritmo genético. El algoritmo es el mismo que el de la Figura 4.15, con una variación: es la versión más popular; cada cruce de dos padres produce sólo un descendiente, no dos.

ocho reinas en los cuales las tres primeras reinas están en posiciones 2, 4 y 6 respectivamente. A las cadenas que emparejan con el esquema (tal como 24613578) se les llaman **instancias** del esquema. Se puede demostrar que, si la idoneidad media de las instancias de un esquema está por encima de la media, entonces el número de instancias del esquema dentro de la población crecerá con el tiempo. Claramente, este efecto improbablemente será significativo si los bits adyacentes están totalmente no relacionados uno al otro, porque entonces habrá pocos bloques contiguos que proporcionen un beneficio consistente. Los algoritmos genéticos trabajan mejor cuando los esquemas corresponden a componentes significativos de una solución. Por ejemplo, si las cadenas son una representación de una antena, entonces los esquemas pueden representar los componentes de la antena, tal como reflectores y deflectores. Un componente bueno probablemente estará bien en una variedad de diseños diferentes. Esto sugiere que el uso acertado de algoritmos genéticos requiere la ingeniería cuidadosa de la representación.

En la práctica, los algoritmos genéticos han tenido un impacto extendido sobre problemas de optimización, como disposición de circuitos y el programado del trabajo en tiendas. Actualmente, no está claro si lo solicitado de los algoritmos genéticos proviene de su funcionamiento o de sus orígenes estéticamente agradables de la teoría de la evolución. Se han hecho muchos trabajos para identificar las condiciones bajo las cuales los algoritmos genéticos funcionan bien.

EVOLUCIÓN Y BÚSQUEDA

La teoría de la **evolución** fue desarrollada por Charles Darwin (1859) en *El Origen de Especies por medio de la Selección Natural*. La idea central es simple: las variaciones (conocidas como **mutaciones**) ocurren en la reproducción y serán conservadas en generaciones sucesivas aproximadamente en la proporción de su efecto sobre la idoneidad reproductiva.

La teoría de Darwin fue desarrollada sin el conocimiento de cómo los rasgos de los organismos se pueden heredar y modificar. Las leyes probabilísticas que gobiernan estos procesos fueron identificadas primero por Gregor Mendel (1866), un monje que experimentó con guisantes dulces usando lo que él llamó la fertilización artificial. Mucho más tarde, Watson y Crick (1953) identificaron la estructura de la molécula de ADN y su alfabeto, AGTC (adenina, guanina, timina, citocina). En el modelo estándar, la variación ocurre tanto por mutaciones en la secuencia de letras como por «el cruce» (en el que el ADN de un descendiente se genera combinando secciones largas del ADN de cada parente).

Ya se ha descrito la analogía con algoritmos de búsqueda local; la diferencia principal entre la búsqueda de haz estocástica y la evolución es el uso de la reproducción sexual, en donde los sucesores se generan a partir de *múltiples* organismos más que de solamente uno. Los mecanismos actuales de la evolución son, sin embargo, mucho más ricos de lo que permiten la mayoría de los algoritmos genéticos. Por ejemplo, las mutaciones pueden implicar inversiones, copias y movimientos de trozos grandes de ADN; algunos virus toman prestado el ADN de un organismo y lo insertan en otro; y hay genes reemplazables que no hacen nada pero se copian miles de veces dentro del genoma. Hay hasta genes que envenenan células de compañeros potenciales que no llevan el gen, bajando el aumento de sus posibilidades de réplica. Lo más importante es el hecho de que los *genes codifican los mecanismos* por los cuales se reproduce y traslada el genoma en un organismo. En algoritmos genéticos, esos mecanismos son un programa separado que no está representado dentro de las cadenas manipuladas.

La evolución Darwiniana podría parecer más bien un mecanismo ineficaz, y ha generado ciegamente aproximadamente 10^{45} organismos sin mejorar su búsqueda heurística un ápice. 50 años antes de Darwin, sin embargo, el gran naturalista francés Jean Lamarck (1809) propuso una teoría de evolución por la cual los rasgos *adquiridos por la adaptación durante la vida de un organismo* serían pasados a su descendiente. Tal proceso sería eficaz, pero no parece ocurrir en la naturaleza. Mucho más tarde, James Baldwin (1896) propuso una teoría superficialmente similar: aquel comportamiento aprendido durante la vida de un organismo podría acelerar la evolución. A diferencia de la de Lamarck, la teoría de Baldwin es completamente consecuente con la evolución Darwiniana, porque confía en presiones de selección que funcionan sobre individuos que han encontrado óptimos locales entre el conjunto de comportamientos posibles permitidos por su estructura genética. Las simulaciones por computadores modernos confirman que «el efecto de Baldwin» es real, a condición de que la evolución «ordinaria» pueda crear organismos cuya medida de rendimiento está, de alguna manera, correlacionada con la idoneidad actual.

4.4 Búsqueda local en espacios continuos

En el Capítulo 2, explicamos la diferencia entre entornos discretos y continuos, señalando que la mayor parte de los entornos del mundo real son continuos. Aún ninguno de los algoritmos descritos puede manejar espacios de estados continuos, ¡la función sucesor en la mayor parte de casos devuelve infinitamente muchos estados! Esta sección proporciona una *muy breve* introducción a técnicas de búsqueda local para encontrar soluciones óptimas en espacios continuos. La literatura sobre este tema es enorme; muchas de las técnicas básicas se originaron en el siglo xvii, después del desarrollo de cálculo Newton y Leibniz¹³. Encontraremos usos para estas técnicas en varios lugares del libro, incluso en los capítulos sobre aprendizaje, visión y robótica. En resumen, cualquier cosa que trata con el mundo real.

Comencemos con un ejemplo. Supongamos que queremos colocar tres nuevos aeropuertos en cualquier lugar de Rumanía, de forma tal que la suma de las distancias al cuadrado de cada ciudad sobre el mapa (Figura 3.2) a su aeropuerto más cercano sea mínima. Entonces el espacio de estados está definido por las coordenadas de los aeropuertos: (x_1, y_1) , (x_2, y_2) , y (x_3, y_3) . Es un espacio *seis-dimensional*; también decimos que los estados están definidos por seis **variables** (en general, los estados están definidos por un vector n -dimensional de variables, \mathbf{x}). Moverse sobre este espacio se corresponde a movimientos de uno o varios de los aeropuertos sobre el mapa. La función objetivo $f(x_1, y_1, x_2, y_2, x_3, y_3)$ es relativamente fácil calcularla para cualquier estado particular una vez que tenemos las ciudades más cercanas, pero bastante complicado anotar en general.

Un modo de evitar problemas continuos es simplemente discretizar la vecindad de cada estado. Por ejemplo, podemos movernos sólo sobre un aeropuerto a la vez, en la dirección x o y , en una cantidad fija $\pm \delta$. Con seis variables, nos da 12 sucesores para cada estado. Podemos aplicar entonces cualquiera de los algoritmos de búsqueda local descritos anteriormente. Uno puede aplicar también la ascensión de colinas estocástica y el temple simulado directamente, sin discretizar el espacio. Estos algoritmos eligen a los sucesores aleatoriamente, que pueden hacerse por la generación de vectores aleatorios de longitud δ .

GRADIENTE

Hay muchos métodos que intentan usar el **gradiente** del paisaje para encontrar un máximo. El gradiente de la función objetivo es un vector ∇f que nos da la magnitud y la dirección de la inclinación más escarpada. Para nuestro problema, tenemos

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

En algunos casos, podemos encontrar un máximo resolviendo la ecuación $\nabla f = 0$ (esto podría hacerse, por ejemplo, si estamos colocando solamente un aeropuerto; la solución es la media aritmética de todas las coordenadas de las ciudades). En muchos casos, sin embargo, esta ecuación no puede resolverse de forma directa. Por ejemplo, con tres aeropuertos, la expresión para el gradiente depende de qué ciudades son las más cercanas a cada aeropuerto en el estado actual. Esto significa que podemos calcular el gradiente

¹³ Un conocimiento básico de cálculo multivariante y aritmética vectorial es útil cuando uno lee esta sección.

localmente pero no *globalmente*. Incluso, podemos realizar todavía la ascensión de colinas por la subida más escarpada poniendo al día el estado actual con la fórmula

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

donde α es una constante pequeña. En otros casos, la función objetivo podría no estar disponible de una forma diferenciable, por ejemplo, el valor de un conjunto particular de posiciones de los aeropuertos puede determinarse ejecutando algún paquete de simulación económica a gran escala. En esos casos, el llamado **gradiente empírico** puede determinarse evaluando la respuesta a pequeños incrementos y decrecimientos en cada coordenada. La búsqueda de gradiente empírico es la misma que la ascensión de colinas con subida más escarpada en una versión discretizada del espacio de estados.

Bajo la frase « α es una constante pequeña» se encuentra una enorme variedad de métodos ajustando α . El problema básico es que, si α es demasiado pequeña, necesitamos demasiados pasos; si α es demasiado grande, la búsqueda podría pasarse del máximo. La técnica de **línea de búsqueda** trata de vencer este dilema ampliando la dirección del gradiente actual (por lo general duplicando repetidamente α) hasta que f comience a disminuir otra vez. El punto en el cual esto ocurre se convierte en el nuevo estado actual. Hay varias escuelas de pensamiento sobre cómo debe elegirse la nueva dirección en este punto.

Para muchos problemas, el algoritmo más eficaz es el venerable método de **Newton-Raphson** (Newton, 1671; Raphson, 1690). Es una técnica general para encontrar raíces de funciones, es decir la solución de ecuaciones de la forma $g(x) = 0$. Trabaja calculando una nueva estimación para la raíz x según la fórmula de Newton.

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

Para encontrar un máximo o mínimo de f , tenemos que encontrar x tal que el gradiente es cero (es decir, $\nabla f(\mathbf{x}) = \mathbf{0}$). Así $g(x)$, en la fórmula de Newton, se transforma en $\nabla f(\mathbf{x})$, y la ecuación de actualización puede escribirse en forma de vector-matriz como

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

donde $\mathbf{H}_f(\mathbf{x})$ es la matriz **Hesiana** de segundas derivadas, cuyo los elementos H_{ij} están descritos por $\partial^2 f / \partial x_i \partial x_j$. Ya que el Hesiano tiene n^2 entradas, Newton-Raphson se hace costoso en espacios dimensionalmente altos, y por tanto, se han desarrollado muchas aproximaciones.

Los métodos locales de búsqueda sufren de máximos locales, crestas, y mesetas tanto en espacios de estados continuos como en espacios discretos. Se pueden utilizar el reinicio aleatorio y el temple simulado y son a menudo provechosos. Los espacios continuos dimensionalmente altos son, sin embargo, lugares grandes en los que es fácil perderse.

Un tema final, que veremos de pasada, es la **optimización con restricciones**. Un problema de optimización está restringido si las soluciones debieran satisfacer algunas restricciones sobre los valores de cada variable. Por ejemplo, en nuestro problema de situar aeropuertos, podría restringir los lugares para estar dentro de Rumanía y sobre la tierra firme (más que en medio de lagos). La dificultad de los problemas de optimización con restricciones depende de la naturaleza de las restricciones y la función objetivo. La categoría más conocida es la de los problemas de **programación lineal**, en los cuales las restricciones deben ser desigualdades lineales formando una región *convexa* y la función

GRADIENTE EMPÍRICO

LÍNEA DE BÚSQUEDA

NEWTON-RAPHSON

HESIANA

OPTIMIZACIÓN CON
RESTRICCIONESPROGRAMACIÓN
LINEAL

objetiva es también lineal. Los problemas de programación lineal pueden resolverse en tiempo polinomial en el número de variables. También se han estudiado problemas con tipos diferentes de restricciones y funciones objetivo (programación cuadrática, programación cónica de segundo orden, etcétera).

4.5 Agentes de búsqueda *online* y ambientes desconocidos

BÚSQUEDA OFFLINE

Hasta ahora nos hemos centrado en agentes que usan algoritmos de **búsqueda offline**. Ellos calculan una solución completa antes de poner un pie en el mundo real (véase la Figura 3.1), y luego ejecutan la solución sin recurrir a su percepciones. En contraste, un agente de **búsqueda en línea (*online*)**¹⁴ funciona **intercalando** el cálculo y la acción: primero toma una acción, entonces observa el entorno y calcula la siguiente acción. La búsqueda *online* es una buena idea en dominios dinámicos o semidinámicos (dominios donde hay una penalización por holgazanear y por utilizar demasiado tiempo para calcular). La búsqueda *online* es una idea incluso mejor para dominios estocásticos. En general, una búsqueda *offline* debería presentar un plan de contingencia exponencialmente grande que considere todos los acontecimientos posibles, mientras que una búsqueda *online* necesita sólo considerar lo que realmente pasa. Por ejemplo, a un agente que juega al ajedrez se le aconseja que haga su primer movimiento mucho antes de que se haya resuelto el curso completo del juego.

PROBLEMA DE EXPLORACIÓN

La búsqueda online es una idea *necesaria* para un **problema de exploración**, donde los estados y las acciones son desconocidos por el agente; un agente en este estado de ignorancia debe usar sus acciones como experimentos para determinar qué hacer después, y a partir de ahí debe intercalar el cálculo y la acción.

El ejemplo básico de búsqueda *online* es un robot que se coloca en un edificio nuevo y lo debe explorar para construir un mapa, que puede utilizar para ir desde A a B. Los métodos para salir de laberintos (conocimiento requerido para aspirar a ser héroe de la Antigüedad) son también ejemplos de algoritmos de búsqueda *online*. Sin embargo, la exploración espacial no es la única forma de la exploración. Considere a un bebé recién nacido: tiene muchas acciones posibles, pero no sabe los resultados de ellas, y ha experimentado sólo algunos de los estados posibles que puede alcanzar. El descubrimiento gradual del bebé de cómo trabaja el mundo es, en parte, un proceso de búsqueda *online*.

Problemas de búsqueda en línea (*online*)

Un problema de búsqueda *online* puede resolverse solamente por un agente que execute acciones, más que por un proceso puramente computacional. Asumiremos que el agente sabe lo siguiente:

¹⁴ El término «en línea (*online*)» es comúnmente utilizado en informática para referirse a algoritmos que deben tratar con los datos de entrada cuando se reciben, más que esperar a que esté disponible el conjunto entero de datos de entrada.

- ACCIONES (s), que devuelve una lista de acciones permitidas en el estado s ;
- Funciones de coste individual $c(s, a, s')$ (notar que no puede usarse hasta que el agente sepa que s' es el resultado); y
- TEST-OBJETIVO(s).

Notemos en particular que el agente *no puede* tener acceso a los sucesores de un estado excepto si intenta realmente todas las acciones en ese estado. Por ejemplo, en el problema del laberinto de la Figura 4.18, el agente no sabe que *Subir* desde (1,1) conduce a (1,2); ni, habiendo hecho esto, sabe que *Bajar* lo devolverá a (1,1). Este grado de ignorancia puede reducirse en algunas aplicaciones (para ejemplo, un robot explorador podría saber cómo trabajan sus acciones de movimiento y ser ignorante sólo de las posiciones de los obstáculos).

Asumiremos que el agente puede reconocer siempre un estado que ha visitado anteriormente, y asumiremos que las acciones son deterministas (en el Capítulo 17, se relajará estos dos últimos axiomas). Finalmente, el agente podría tener acceso a una función heurística admisible $h(s)$ que estime la distancia del estado actual a un estado objetivo. Por ejemplo, en la Figura 4.18, el agente podría saber la posición del objetivo y ser capaz de usar la distancia heurística de Manhattan.

Típicamente, el objetivo del agente es alcanzar un estado objetivo minimizando el coste (otro objetivo posible es explorar simplemente el entorno entero). El costo es el costo total del camino por el que el agente viaja realmente. Es común comparar este costo con el costo del camino que el agente seguiría *si supiera el espacio de búsqueda de antemano*, es decir, el camino más corto actual (o la exploración completa más corta). En el lenguaje de algoritmos *online*, se llama **proporción competitiva**; nos gustaría que fuera tan pequeña como sea posible.

Aunque ésta suene como una petición razonable, es fácil ver que la mejor proporción alcanzable competitiva es infinita en algunos casos. Por ejemplo, si algunas acciones son irreversibles, la búsqueda *online* podría alcanzar, por casualidad, un estado sin salida del cual no es accesible ningún estado objetivo.

Quizás encuentre el término «por casualidad» poco convincente (después de todo, podría haber un algoritmo que no tome el camino sin salida mientras explora). Nuestra

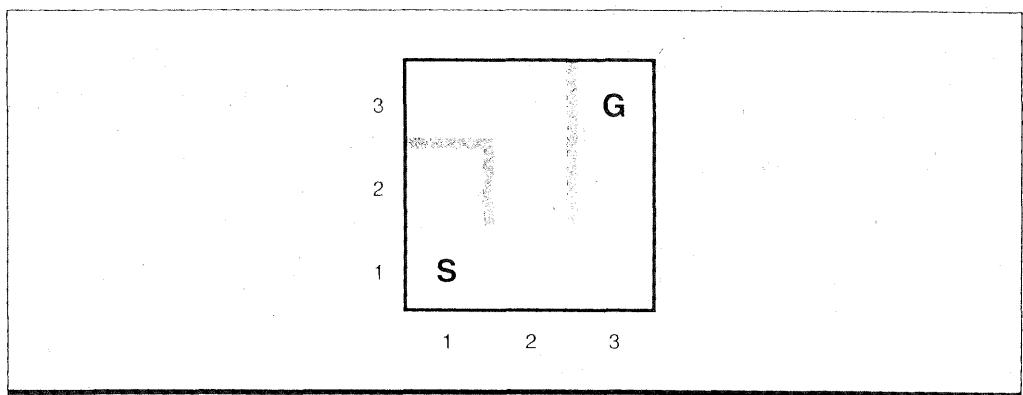


Figura 4.18 Un problema sencillo de un laberinto. El agente comienza en S y debe alcanzar G , pero no sabe nada del entorno.



ARGUMENTO DE
ADVERSARIO

SEGURAMENTE
EXPLORABLE

reclamación, para ser más precisos, consiste en que *ningún algoritmo puede evitar callejones sin salida en todos los espacios de estados*. Considere los dos espacios de estados sin salida de la Figura 4.19 (a). A un algoritmo de búsqueda *online* que haya visitado los estados S y A , los dos espacios de estados parecen *idénticos*, entonces debe tomar la misma decisión en ambos. Por lo tanto, fallará en uno de ellos. Es un ejemplo de un **argumento de adversario** (podemos imaginar un adversario que construye el espacio de estados, mientras el agente lo explora, y puede poner el objetivo y callejones sin salida donde le guste).

Los callejones sin salida son una verdadera dificultad para la exploración de un robot (escaleras, rampas, acantilados, y todas las clases de posibilidades presentes en terrenos naturales de acciones irreversibles). Para avanzar, asumiremos simplemente que el espacio de estados es **seguramente explorable**, es decir, algún estado objetivo es alcanzable desde cualquier estado alcanzable. Los espacios de estados con acciones reversibles, como laberintos y 8-puzzles, pueden verse como grafos no-dirigidos y son claramente explorables.

Incluso en entornos seguramente explorables no se puede garantizar ninguna proporción competitiva acotada si hay caminos de costo ilimitado. Esto es fácil de demostrar en entornos con acciones irreversibles, pero de hecho permanece cierto también para el caso reversible, como se muestra en la Figura 4.19(b). Por esta razón, es común describir el funcionamiento de los algoritmos de búsqueda *online* en términos del tamaño del espacio de estados entero más que, solamente, por la profundidad del objetivo más superficial.

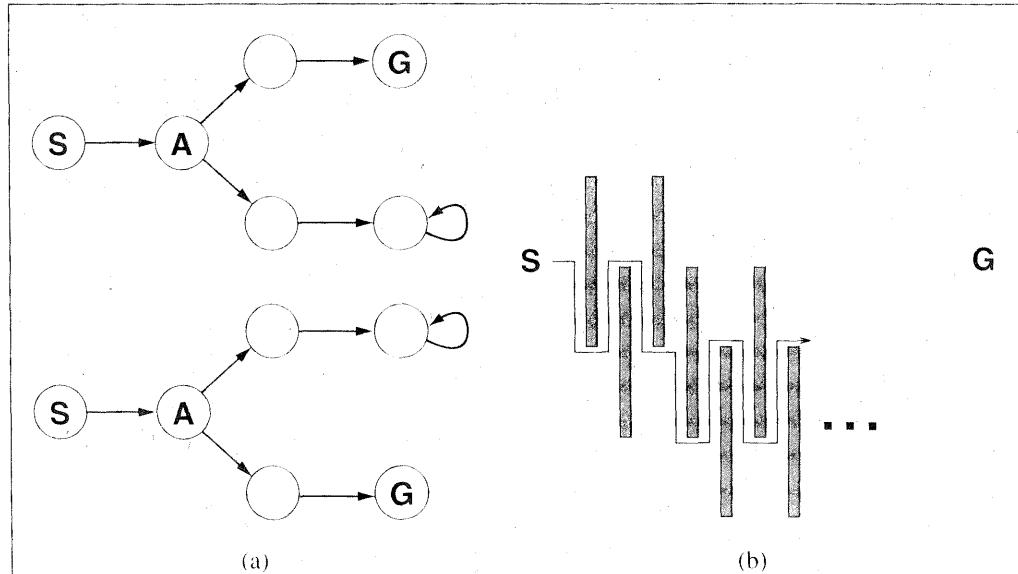


Figura 4.19 (a) Dos espacios de estados que podrían conducir a un agente de búsqueda *online* a un callejón sin salida. Cualquier agente fallará en al menos uno de estos espacios. (b) Un entorno de dos-dimensiones que puede hacer que un agente de búsqueda *online* siga una ruta arbitrariamente ineficaz al objetivo. Ante cualquier opción que tome el agente, el adversario bloquea esa ruta con otra pared larga y delgada, de modo que el camino seguido sea mucho más largo que el camino mejor posible.

Agentes de búsqueda en línea (*online*)

Después de cada acción, un agente *online* recibe una percepción al decirle que estado ha alcanzado; de esta información, puede aumentar su mapa del entorno. El mapa actual se usa para decidir dónde ir después. Esta intercalación de planificación y acción significa que los algoritmos de búsqueda *online* son bastante diferentes de los algoritmos de búsqueda *offline* vistos anteriormente. Por ejemplo, los algoritmos *offline* como A* tienen la capacidad de expandir un nodo en una parte del espacio y luego inmediatamente expandir un nodo en otra parte del espacio, porque la expansión de un nodo implica simulación más que verdaderas acciones. Un algoritmo *online*, por otra parte, puede expandir sólo el nodo que ocupa físicamente. Para evitar viajar a través de todo el árbol para expandir el siguiente nodo, parece mejor expandir los nodos en un orden *local*. La búsqueda primero en profundidad tiene exactamente esta propiedad, porque (menos cuando volvemos hacia atrás) el siguiente nodo a expandir es un hijo del nodo anteriormente expandido.

En la Figura 4.20 se muestra un agente de búsqueda primero en profundidad *online*. Este agente almacena su mapa en una tabla, *resultado*[*a,s*], que registra el estado que resulta de ejecutar la acción *a* en el estado *s*. Siempre que una acción del estado actual no haya sido explorada, el agente intenta esa acción. La dificultad viene cuando el agente ha intentado todas las acciones en un estado. En la búsqueda primero en profundidad *offline*, el estado es simplemente quitado de la cola; en una búsqueda *online*, el agente tiene que volver atrás físicamente. La búsqueda primero en profundidad, significa volver al estado el cual el agente incorporó el estado actual más recientemente. Esto se con-

```

función AGENTE-BPP-ONLINE(s') devuelve una acción
  entradas: s', una percepción que identifica el estado actual
  estático: resultado, una tabla, indexada por la acción y el estado, inicialmente vacía
           noexplorados, una tabla que enumera, para cada estado visitado, las acciones
           todavía no intentadas
           nohaciatrás, una tabla que enumera, para cada estado visitado, los nodos hacia
           atrás todavía no intentados
           s,a, el estado y acción previa, inicialmente nula

  si TEST-OBJETIVO(s') entonces devolver parar
  si s' es un nuevo estado entonces noexplorados[s'] ← ACCIONES(s')
  si s es no nulo entonces hacer
    resultado[a,s] ← s'
    añadir s al frente de nohaciatrás[s']
  si noexplorados[s'] esta vacío entonces
    si nohaciatrás[s'] esta vacío entonces devolver parar
    en caso contrario a ← una acción b tal que resultado[b, s'] = POP(nohaciatrás[s'])
  en caso contrario a ← POP(noexplorados[s']))
  s ← s'
  devolver a
```

Figura 4.20 Un agente de búsqueda *online* que utiliza la exploración primero en profundidad. El agente es aplicable, solamente, en espacios de búsqueda bidireccionales.

sigue guardando una tabla que pone en una lista, para cada estado, los estados predecesores a los cuales aún no ha vuelto. Si el agente se ha quedado sin estados a los que volver, entonces su búsqueda se ha completado.

Recomendamos al lector que compruebe el progreso del AGENTE-BPP-ONLINE cuando se aplica al laberinto de la Figura 4.18. Es bastante fácil ver que el agente, en el caso peor, terminará por cruzar cada enlace en el espacio de estados exactamente dos veces. Por exploración, esto es lo óptimo; para encontrar un objetivo, por otra parte, la proporción competitiva del agente podría ser arbitrariamente mala si se viaja sobre un camino largo cuando hay un objetivo directamente al lado del estado inicial. Una variante *online* de la profundidad iterativa resuelve este problema; para un entorno representado por un árbol uniforme, la proporción competitiva del agente es una constante pequeña.

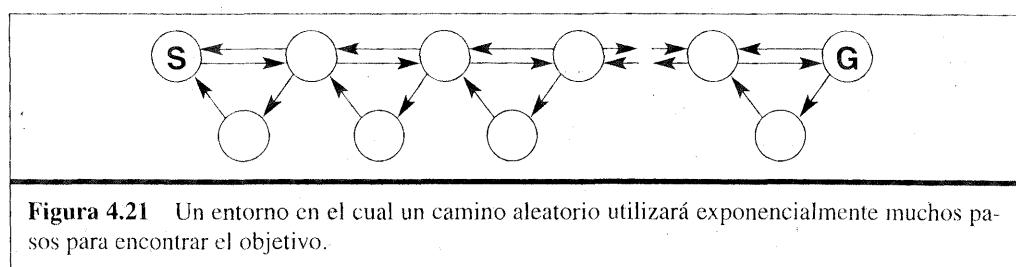
A causa de su método de vuelta atrás, el AGENTE-BPP-ONLINE trabaja sólo en espacios de estados donde las acciones son reversibles. Hay algoritmos ligeramente más complejos que trabajan en espacios de estados generales, pero ninguno de estos algoritmos tiene una proporción competitiva acotada.

Búsqueda local en línea (*online*)

Como la búsqueda primero en profundidad, la **búsqueda de ascensión de colinas** tiene la propiedad de localidad en sus expansiones de los nodos. ¡De hecho, porque mantiene un estado actual en memoria, la búsqueda de ascensión de colinas es ya un algoritmo de búsqueda *online*! Desafortunadamente, no es muy útil en su forma más simple porque deja al agente que se sitúe en máximos locales con ningún movimiento que hacer. Por otra parte, los reinicios aleatorios no pueden utilizarse, porque el agente no puede moverse a un nuevo estado.

CAMINO ALEATORIO

En vez de reinicios aleatorios, podemos considerar el uso de un **camino aleatorio** para explorar el entorno. Un camino aleatorio selecciona simplemente al azar una de las acciones disponibles del estado actual; se puede dar preferencia a las acciones que todavía no se han intentado. Es fácil probar que un camino aleatorio encontrará *al final* un objetivo o termina su exploración, a condición de que el espacio sea finito¹⁵. Por otra parte, el proceso puede ser muy lento. La Figura 4.21 muestra un entorno en el que un



¹⁵ El caso infinito es mucho más difícil. ¡Los caminos aleatorios son completos en rejillas unidimensionales y de dos dimensiones infinitas, pero no en rejillas tridimensionales! En el último caso, la probabilidad de que el camino vuelva siempre al punto de partida es alrededor de 0,3405 (véase a Hughes, 1995, para una introducción general).

camino aleatorio utilizará un número exponencial de pasos para encontrar el objetivo, porque, en cada paso, el progreso hacia atrás es dos veces más probable que el progreso hacia delante. El ejemplo es artificial, por supuesto, pero hay muchos espacios de estados del mundo real cuya topología causa estas clases de «trampas» para los caminos aleatorios.

Aumentar a la ascensión de colinas con *memoria* más que aleatoriedad, resulta ser una aproximación más eficaz. La idea básica es almacenar una «mejor estimación actual» $H(s)$ del coste para alcanzar el objetivo desde cada estado que se ha visitado. El comienzo de $H(s)$ es justo la estimación heurística $h(s)$ y se actualiza mientras que el agente gana experiencia en el espacio de estados. La Figura 4.22 muestra un ejemplo sencillo en un espacio de estados unidimensional. En (a), el agente parece estar estancado en un mínimo local plano en el estado sombreado. Más que permanecer donde está, el agente debe seguir por donde parece ser la mejor trayectoria al objetivo, basada en las estimaciones de los costes actuales para sus vecinos. El coste estimado para alcanzar el objetivo a través de un vecino s' es el coste para conseguir s' más el coste estimado para conseguir un objetivo desde ahí, es decir, $c(s, a, s') + H(s')$. En el ejemplo, hay dos acciones con costos estimados $1 + 9$ y $1 + 2$, así parece que lo mejor posible es moverse a la derecha. Ahora, está claro que la estimación del costo de dos para el estado sombreado fue demasiado optimista. Puesto que el mejor movimiento costó uno y condujo a un estado que está al menos a dos pasos de un objetivo, el estado sombreado debe estar por lo menos a tres pasos de un objetivo, así que su H debe actualizarse adecuadamente, como se muestra en la figura 4.22(b). Continuando este proceso, el agente se moverá hacia delante y hacia atrás dos veces más, actualizando H cada vez y «apartando» el mínimo local hasta que se escapa a la derecha.

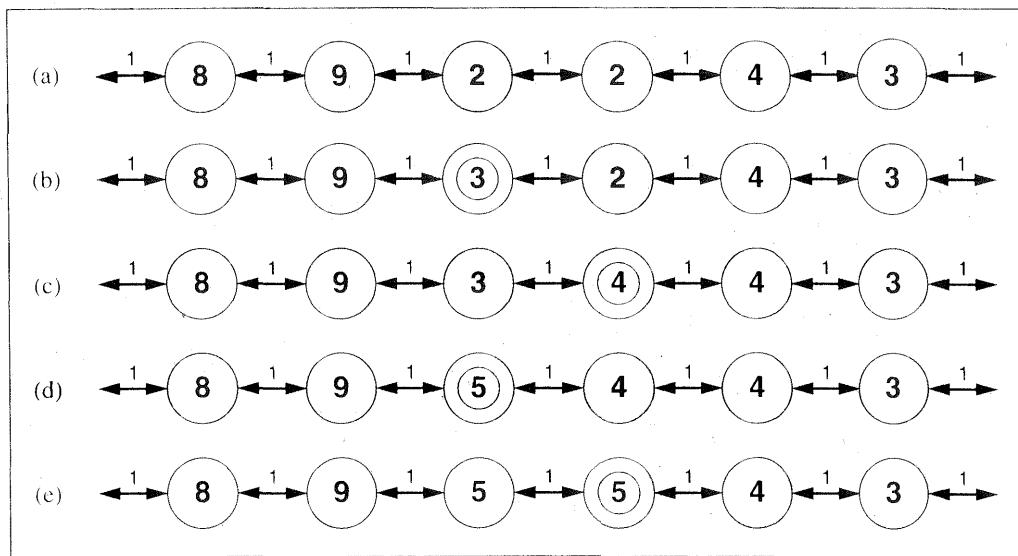


Figura 4.22 Cinco iteraciones de AA*TR en un espacio de estados unidimensional. Cada estado se marca con $H(s)$, la estimación del costo actual para alcanzar un objetivo, y cada arco se marca con su costo. El estado sombreado marca la ubicación del agente, y se rodean los valores actualizados en cada iteración.

AA*TR

OPTIMISMO BAJO
INCERTIDUMBRE

En la Figura 4.23 se muestra un agente que implementa este esquema, llamado aprendiendo A* en tiempo real (**AA*TR**). Como el AGENTE-BPP-ONLINE, éste construye un mapa del entorno usando la tabla *resultado*. Actualiza el costo estimado para el estado que acaba de dejar y entonces escoge el movimiento «aparentemente mejor» según sus costos estimados actuales. Un detalle importante es que las acciones que todavía no se han intentado en un estado s siempre se supone que dirigen inmediatamente al objetivo con el costo menor posible, $h(s)$. Este **optimismo bajo la incertidumbre** anima al agente a explorar nuevos y posiblemente caminos prometedores.

función AGENTE-AA*TR(s') **devuelve** una acción

entradas: s' , una percepción que identifica el estado actual

estático: $resultado$, una tabla, indexada por la acción y el estado, inicialmente vacía
 H , una tabla de costos estimados indexada por estado, inicialmente vacía
 s, a , el estado y acción previa, inicialmente nula

si TEST-OBJETIVO(s') **entonces devolver** *parar*

si s' es un nuevo estado (no en H) **entonces** $H[s'] \leftarrow h(s')$

a menos que s sea nulo

$resultado[a, s] \leftarrow s'$

$H[s] \leftarrow \min_{b \in ACCIONES(s)} COSTO-AA*TR(s, b, resultado[b, s], H)$

$a \leftarrow$ una acción b de $ACCIONES(s')$ que minimiza $COSTO-AA*TR(s', b, resultado[b, s'], H)$

$s \leftarrow s'$

devolver a

función COSTO-AA*TR(s, a, s', H) **devuelve** un costo estimado

si s' está indefinido **entonces devolver** $h(s)$

en otro caso devolver $c(s, a, s') + H[s']$

Figura 4.23 El AGENTE-AA*TR escoge una acción según los valores de los estados vecinos, que se actualizan conforme el agente se mueve sobre el espacio de estados.

Un agente AA*TR garantiza encontrar un objetivo en un entorno seguramente explorable y finito. A diferencia de A*, sin embargo, no es completo para espacios de estados infinitos (hay casos donde se puede dirigir infinitamente por mal camino). Puede explorar un entorno de n estados en $O(n^2)$ pasos, en el caso peor, pero a menudo lo hace mejor. El agente AA*TR es sólo uno de una gran familia de agentes *online* que pueden definirse especificando la regla de la selección de la acción y que actualiza la regla de maneras diferentes. Discutiremos esta familia, que fue desarrollada originalmente para entornos estocásticos, en el Capítulo 21.

Aprendizaje en la búsqueda en línea (*online*)

La ignorancia inicial de los agentes de búsqueda *online* proporciona varias oportunidades para aprender. Primero, los agentes aprenden un «mapa» del entorno (más precisamente, el resultado de cada acción en cada estado) simplemente registrando cada una de sus experiencias (notemos que la suposición de entornos deterministas quiere decir que

una experiencia es suficiente para cada acción). Segundo, los agentes de búsqueda locales adquieren estimaciones más exactas del valor de cada estado utilizando las reglas de actualización local, como en AA*TR. En el Capítulo 21 veremos que éstas actualizaciones convergen finalmente a valores *exactos* para cada estado, con tal de que el agente explore el espacio de estados de manera correcta. Una vez que se conocen los valores exactos, se pueden tomar las decisiones óptimas simplemente moviéndose al sucesor con el valor más alto (es decir, la ascensión de colinas pura es entonces una estrategia óptima).

Si usted siguió nuestra sugerencia de comprobar el comportamiento del AGENTE-BPP-ONLINE en el entorno de la Figura 4.18, habrá advertido que el agente no es muy brillante. Por ejemplo, después de ver que la acción *Arriba* va de (1,1) a (1,2), el agente no tiene la menor idea todavía que la acción *Abajo* vuelve a (1,1), o que la acción *Arriba* va también de (2,1) a (2,2), de (2,2) a (2,3), etcétera. En general, nos gustaría que el agente aprendiera que *Arriba* aumenta la coordenada y a menos que haya una pared en el camino, que hacia *Abajo* la reduce, etcétera. Para que esto suceda, necesitamos dos cosas: primero, necesitamos una representación formal y explícitamente manipulable para estas clases de reglas generales; hasta ahora, hemos escondido la información dentro de la caja negra llamada función sucesor. La Parte III está dedicada a este tema. Segundo, necesitamos algoritmos que puedan construir reglas generales adecuadas a partir de la observación específica hecha por el agente. Estos algoritmos se tratan en el Capítulo 18.

4.6 Resumen

Este capítulo ha examinado la aplicación de **heurísticas** para reducir los costos de la búsqueda. Hemos mirado varios algoritmos que utilizan heurísticas y encontramos que la optimalidad tiene un precio excesivo en términos del costo de búsqueda, aún con heurísticas buenas.

- **Búsqueda primero el mejor** es una BÚSQUEDA-GRAFO donde los nodos no expandidos de costo mínimo (según alguna medida) se escogen para la expansión. Los algoritmos primero el mejor utilizan típicamente una función heurística $h(n)$ que estima el costo de una solución desde n .
- **Búsqueda primero el mejor avara** expande nodos con $h(n)$ mínima. No es óptima, pero es a menudo eficiente.
- **Búsqueda A*** expande nodos con mínimo $f(n) = g(n) + h(n)$. A* es completa y óptima, con tal que garanticemos que $h(n)$ sea admisible (para BÚSQUEDA-ARBOL) o consistente (para BÚSQUEDA-GRAFO). La complejidad en espacio de A* es todavía prohibitiva.
- El rendimiento de los algoritmos de búsqueda heurística depende de la calidad de la función heurística. Las heurísticas buenas pueden construirse a veces relajando la definición del problema, por costos de solución precalculados para sub-problemas en un modelo de bases de datos, o aprendiendo de la experiencia con clases de problemas.

- **BRPM** y **A*MS** son algoritmos de búsqueda robustos y óptimos que utilizan cantidades limitadas de memoria; con suficiente tiempo, pueden resolver los problemas que A* no puede resolver porque se queda sin memoria.
- Los métodos de *búsqueda local*, como la **ascensión de colinas**, operan en formulaciones completas de estados, manteniendo sólo un número pequeño de nodos en memoria. Se han desarrollado varios algoritmos estocásticos, inclusive el **templo simulado**, que devuelven soluciones óptimas cuando se da un apropiado programa de enfriamiento. Muchos métodos de búsqueda local se pueden utilizar también para resolver problemas en espacios continuos.
- Un **algoritmo genético** es una búsqueda de ascensión de colinas estocástica en la que se mantiene una población grande de estados. Los estados nuevos se generan por **mutación** y por **cruce**, combinando pares de estados de la población.
- Los **problemas de exploración** surgen cuando el agente no tiene la menor idea acerca de los estados y acciones de su entorno. Para entornos seguramente explorables, los agentes de **búsqueda en línea** pueden construir un mapa y encontrar un objetivo si existe. Las estimaciones de las heurística, que se actualizan por la experiencia, proporcionan un método efectivo para escapar de mínimos locales.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

El uso de información heurística en la resolución de problemas aparece en un artículo de Simon y Newell (1958), pero la frase «búsqueda heurística» y el uso de las funciones heurísticas que estiman la distancia al objetivo llegaron algo más tarde (Newell y Ernst, 1965; Lin, 1965). Doran y Michie (1966) dirigieron muchos estudios experimentales de búsqueda heurística aplicados a varios problemas, especialmente al 8-puzzle y 15-puzzle. Aunque Doran y Michie llevaran a cabo un análisis teórico de la longitud del camino y «penetrancia» (proporción entre la longitud del camino y el número total de nodos examinados hasta el momento) en la búsqueda heurística, parecen haber ignorado la información proporcionada por la longitud actual del camino. El algoritmo A*, incorporando la longitud actual del camino en la búsqueda heurística, fue desarrollado por Hart, Nilsson y Raphael (1968), con algunas correcciones posteriores (Hart *et al.*, 1972). Dechter y Pearl (1985) demostraron la eficiencia óptima de A*.

El artículo original de A* introdujo la condición de consistencia en funciones heurísticas. La condición de monotonía fue introducida por Pohl (1977) como un sustituto más sencillo, pero Pearl (1984) demostró que las dos eran equivalentes. Varios algoritmos precedentes de A* utilizaron el equivalente de listas abiertas y cerradas; éstos incluyen la búsqueda primero en anchura, primero en profundidad, y costo uniforme (Bellman, 1957; Dijkstra, 1959). El trabajo de Bellman en particular mostró la importancia de añadir los costos de los caminos para simplificar los algoritmos de optimización.

Pohl (1970, 1977) fue el pionero en el estudio de la relación entre el error en las funciones heurísticas y la complejidad en tiempo de A*. La demostración de que A* se ejecuta en un tiempo lineal si el error de la función heurística está acotado por una constante puede encontrarse en Pohl (1977) y en Gaschnig (1979). Pearl (1984) reforzó este re-

sultado para permitir un crecimiento logarítmico en el error. El «factor de ramificación eficaz», medida de la eficiencia de la búsqueda heurística, fue propuesto por Nilsson (1971).

Hay muchas variaciones del algoritmo A*. Pohl (1973) propuso el uso del *ponderado dinámico*, el cual utiliza una suma ponderada $f_w(n) = w_g(n) + w_h(n)$ de la longitud del camino actual y de la función heurística como una función de evaluación, más que la suma sencilla $f(n) = g(n) + h(n)$ que utilizó A*. Los pesos w_g y w_h se ajustan dinámicamente con el progreso de la búsqueda. Se puede demostrar que el algoritmo de Pohl es ϵ -admisible (es decir, garantiza encontrar las soluciones dentro de un factor $1 + \epsilon$ de la solución óptima) donde ϵ es un parámetro suministrado al algoritmo. La misma propiedad es exhibida por el algoritmo A_ϵ^* (Pearl, 1984), el cual puede escoger cualquier nodo de la franja tal que su f -costo esté dentro de un factor $1 + \epsilon$ del nodo de la franja de f -costo más pequeño. La selección se puede hacer para minimizar el costo de la búsqueda.

A^* y otros algoritmos de búsqueda en espacio de estados están estrechamente relacionados con las técnicas de *ramificar-y-acotar* ampliamente utilizadas en investigación operativa (Lawler y Wood, 1966). Las relaciones entre la búsqueda en espacio de estados y ramificar-y-acotar se han investigado en profundidad (Kumar y Kanal, 1983; Nau *et al.*, 1984; Kumas *et al.*, 1988). Martelli y Montanari (1978) demostraron una conexión entre la programación dinámica (véase el Capítulo 17) y cierto tipo de búsqueda en espacio de estados. Kumar y Kanal (1988) intentan una «ambiciosa unificación» de la búsqueda heurística, programación dinámica, y técnicas de ramifica-y-acotar bajo el nombre de PDC (el «proceso de decisión compuesto»).

Como los computadores a finales de los años 1950 y principios de los años 1960 tenían como máximo unas miles de palabras de memoria principal, la búsqueda heurística con memoria-acotada fue un tema de investigación. El Grafo Atravesado (Doran y Michie, 1966), uno de los programas de búsqueda más antiguos, compromete a un operador después de realizar una búsqueda primero el mejor hasta el límite de memoria. A^*PI (Korf, 1985a, 1985b) fue el primero que usó un algoritmo de búsqueda heurística, óptima, con memoria-acotada y de la que se han desarrollado un número grande de variantes. Un análisis de la eficiencia de A^*PI y de sus dificultades con las heurística real-valoradas aparece en Patrick *et al.* (1992).

El BRPM (Korf, 1991, 1993) es realmente algo más complicado que el algoritmo mostrado en la Figura 4.5, el cual está más cercano a un algoritmo, desarrollado independientemente, llamado **extensión iterativa**, o EI (Russell, 1992). BRPM usa una cota inferior y una cota superior; los dos algoritmos se comportan idénticamente con heurísticas admisibles, pero BRPM expande nodos en orden primero el mejor hasta con una heurística inadmisible. La idea de guardar la pista del mejor camino alternativo apareció en la implementación elegante, en Prolog, de A^* realizada por Bratko (1986) y en el algoritmo DTA* (Russell y Wefald, 1991). El trabajo posterior también habló de espacios de estado meta nivel y aprendizaje meta nivel.

El algoritmo A^*M apareció en Chakrabarti *et al.* (1989). A^*MS , o A^*M simplificado, surgió de una tentativa de implementación de A^*M como un algoritmo de comparación para IE (Russell, 1992). Kaindl y Khorsand (1994) han aplicado A^*MS para producir un algoritmo de búsqueda bidireccional considerablemente más rápido que los

algoritmos anteriores. Korf y Zhang (2000) describen una aproximación divide-y-vencerás, y Zhou y Hansen (2002) introducen una búsqueda A* en un grafo de memoria-acotada. Korf (1995) revisa las técnicas de búsqueda de memoria-acotada.

La idea de que las heurísticas admisibles pueden obtenerse por relajación del problema aparece en el trabajo seminal de Held y Karp (1970), quien utilizó la heurística del mínimo-atravesando el árbol para resolver el PVC (véase el Ejercicio 4.8).

La automatización del proceso de relajación fue implementado con éxito por Prieditis (1993), construido sobre el trabajo previo de Mostow (Mostow y Prieditis, 1989). El uso del modelo de bases de datos para obtener heurísticas admisibles se debe a Gasser (1995) y Culberson y Schaeffer (1998); el modelo de bases de datos disjuntas está descrito por Korf y Felner (2002). La interpretación probabilística de las heurística fue investigada en profundidad por Pearl (1984) y Hansson y Mayer (1989).

La fuente bibliográfica más comprensiva sobre heurísticas y algoritmos de búsqueda heurístico está en el texto *Heuristics* de Pearl (1984). Este libro cubre de una manera especialmente buena la gran variedad de ramificaciones y variaciones de A*, incluyendo demostraciones rigurosas de sus propiedades formales. Kanal y Kumar (1988) presentan una antología de artículos importantes sobre la búsqueda heurística. Los nuevos resultados sobre algoritmos de búsqueda aparecen con regularidad en la revista *Artificial Intelligence*.

Las técnicas locales de búsqueda tienen una larga historia en matemáticas y en informática. En efecto, el método de Newton-Raphson (Newton, 1671; Raphson, 1690) puede verse como un método de búsqueda local muy eficiente para espacios continuos en los cuales está disponible la información del gradiente. Brent (1973) es una referencia clásica para algoritmos de optimización que no requieren tal información. La búsqueda de haz, que hemos presentado como un algoritmo de búsqueda local, se originó como una variante de anchura-acotada de la programación dinámica para el reconocimiento de la voz en el sistema HARPY (Lowerre, 1976). En Pearl (1984, el Capítulo 5) se analiza en profundidad un algoritmo relacionado.

El tema de la búsqueda local se ha fortalecido en los últimos años por los resultados sorprendentemente buenos en problemas de satisfacción de grandes restricciones como las *n*-reinas (Minton *et al.*, 1992) y de razonamiento lógico (Selman *et al.*, 1992) y por la incorporación de aleatoriedad, múltiples búsquedas simultáneas, y otras mejoras. Este renacimiento, de lo que Christos Papadimitriou ha llamado algoritmos de la «Nueva Era», ha provocado también el interés entre los informáticos teóricos (Koutsoupias y Papadimitriou, 1992; Aldous y Vazirani, 1994). En el campo de la investigación operativa, una variante de la ascensión de colinas, llamada **búsqueda tabú**, ha ganado popularidad (Glover, 1989; Glover y Laguna, 1997). Realizado sobre modelos de memoria limitada a corto plazo de los humanos, este algoritmo mantienen una lista tabú de *k* estados, previamente visitados, que no pueden visitarse de nuevo; así se mejora la eficiencia cuando se busca en grafos y además puede permitir que el algoritmo se escape de algunos mínimos locales. Otra mejora útil sobre la ascensión de colinas es el algoritmo de STAGE (Boyan y Moore, 1998). La idea es usar los máximos locales encontrados por la ascensión de colinas de reinicio aleatorio para conseguir una idea de la forma total del paisaje. El algoritmo adapta una superficie suave al conjunto de máximos locales y luego calcula analíticamente el máximo global de esa superficie. Éste se convierte en el nuevo punto de

reinicio. Se ha demostrado que este algoritmo trabaja, en la práctica, sobre problemas difíciles. (Gomes *et al.*, 1998) mostraron que las distribuciones, en tiempo de ejecución, de los algoritmos de vuelta atrás sistemáticos a menudo tienen una **distribución de cola pesada**, la cual significa que la probabilidad de un tiempo de ejecución muy largo es mayor que lo que sería predicho si los tiempos de ejecución fueran normalmente distribuidos. Esto proporciona una justificación teórica para los reinicios aleatorios.

El temple simulado fue inicialmente descrito por Kirkpatrick *et al.* (1983), el cual se basó directamente en el **algoritmo de Metrópolis** (usado para simular sistemas complejos en la física (Metrópolis *et al.*, 1953) y fue supuestamente inventado en la cena Los Alamos). El temple simulado es ahora un campo en sí mismo, con cien trabajos publicados cada año.

Encontrar soluciones óptimas en espacios continuos es la materia de varios campos, incluyendo la **teoría de optimización**, **teoría de control óptima**, y el **cálculo de variaciones**. Los convenientes (y prácticos) puntos de entrada son proporcionados por Press *et al.* (2002) y Bishop (1995). La **programación lineal** (PL) fue una de las primeras aplicaciones de computadores; el **algoritmo simplex** (Wood y Dantzig, 1949; Dantzig, 1949) todavía se utiliza a pesar de la complejidad exponencial, en el peor caso. Karmarkar (1984) desarrolló un algoritmo de tiempo polinomial práctico para PL.

El trabajo de Sewall Wright (1931), sobre el concepto de la **idoneidad de un paisaje**, fue un precursor importante para el desarrollo de los algoritmos genéticos. En los años 50, varios estadísticos, incluyendo Box (1957) y Friedman (1959), usaron técnicas evolutivas para problemas de optimización, pero no fue hasta que Rechenberg (1965, 1973) introdujera las **estrategias de evolución** para resolver problemas de optimización para planos aerodinámicos en la que esta aproximación ganó popularidad. En los años 60 y 70, John Holland (1975) defendió los algoritmos genéticos, como un instrumento útil y como un método para ampliar nuestra comprensión de la adaptación, biológica o de otra forma (Holland, 1995). El movimiento de **vida artificial** (Langton, 1995) lleva esta idea un poco más lejos, viendo los productos de los algoritmos genéticos como *organismos* más que como soluciones de problemas. El trabajo de Hinton y Nowlan (1987) y Ackley y Littman (1991) en este campo ha hecho mucho para clarificar las implicaciones del efecto de Baldwin. Para un tratamiento más a fondo y general sobre la evolución, recomendamos a Smith y Szathmáry (1999).

La mayor parte de comparaciones de los algoritmos genéticos con otras aproximaciones (especialmente ascensión de colinas estocástica) han encontrado que los algoritmos genéticos son más lentos en converger (O'Reilly y Oppacher, 1994; Mitchell *et al.*, 1996; Juels y Watternberg, 1996; Baluja, 1997). Tales conclusiones no son universalmente populares dentro de la comunidad de AG, pero tentativas recientes dentro de esa comunidad, para entender la búsqueda basada en la población como una forma aproximada de aprendizaje Bayesiano (véase el Capítulo 20), quizás ayude a cerrar el hueco entre el campo y sus críticas (Pelikan *et al.*, 1999). La teoría de **sistemas dinámicos cuadráticos** puede explicar también el funcionamiento de AGs (Rabani *et al.*, 1998). Véase Lohn *et al.* (2001) para un ejemplo de AGs aplicado al diseño de antenas, y Larrañaga *et al.* (1999) para una aplicación al problema de viajante de comercio.

El campo de la **programación genética** está estrechamente relacionado con los algoritmos genéticos. La diferencia principal es que las representaciones, que son muta-

das y combinadas, son programas más que cadenas de bits. Los programas se representan en forma de árboles de expresión; las expresiones pueden estar en un lenguaje estándar como Lisp o pueden estar especialmente diseñadas para representar circuitos, controladores del robot, etcétera. Los cruces implican unir los subárboles más que las subcadenas. De esta forma, la mutación garantiza que los descendientes son expresiones gramaticalmente correctas, que no lo serían si los programas fueran manipulados como cadenas.

El interés reciente en la programación genética fue estimulado por el trabajo de John Koza (Koza, 1992, 1994), pero va por detrás de los experimentos con código máquina de Friedberg (1958) y con autómatas de estado finito de Fogel *et al.* (1966). Como con los algoritmos genéticos, hay un debate sobre la eficacia de la técnica. Koza *et al.* (1999) describen una variedad de experimentos sobre el diseño automatizado de circuitos de dispositivos utilizando la programación genética.

Las revistas *Evolutionary Computation* y *IEEE Transactions on Evolutionary Computation* cubren los algoritmos genéticos y la programación genética; también se encuentran artículos en *Complex Systems*, *Adaptative Behavior*, y *Artificial Life*. Las conferencias principales son la *International Conference on Genetic Algorithms* y la *Conference on Genetic Programming*, recientemente unidas para formar la *Genetic and Evolutionary Computation Conference*. Los textos de Melanie Mitchell (1996) y David Fogel (2000) dan descripciones buenas del campo.

Los algoritmos para explorar espacios de estados desconocidos han sido de interés durante muchos siglos. La búsqueda primero en profundidad en un laberinto puede implementarse manteniendo la mano izquierda sobre la pared; los bucles pueden evitarse marcando cada unión. La búsqueda primero en profundidad falla con acciones irreversibles; el problema más general de exploración de **grafos Eulerianos** (es decir, grafos en los cuales cada nodo tiene un número igual de arcos entrantes y salientes) fue resuelto por un algoritmo debido a Hierholzer (1873). El primer estudio cuidadoso algorítmico del problema de exploración para grafos arbitrarios fue realizado por Deng y Papadimitriou (1990), quienes desarrollaron un algoritmo completamente general, pero demostraron que no era posible una proporción competitiva acotada para explorar un grafo general. Papadimitriou y Yannakakis (1991) examinaron la cuestión de encontrar caminos a un objetivo en entornos geométricos de planificación de caminos (donde todas las acciones son reversibles). Ellos demostraron que es alcanzable una pequeña proporción competitiva con obstáculos cuadrados, pero no se puede conseguir una proporción acotada con obstáculos generales rectangulares (véase la Figura 4.19).

El algoritmo LRTA* fue desarrollado por Korf (1990) como parte de una investigación en la **búsqueda en tiempo real** para entornos en los cuales el agente debe actuar después de buscar en sólo una cantidad fija del tiempo (una situación mucho más común en juegos de dos jugadores). El LRTA* es, de hecho, un caso especial de algoritmos de aprendizaje por refuerzo para entornos estocásticos (Barto *et al.*, 1995). Su política de optimismo bajo incertidumbre (siempre se dirige al estado no visitado más cercano) puede causar un modelo de exploración que es menos eficiente, en el caso sin información, que la búsqueda primero en profundidad simple (Koenig, 2000). Dasgupta *et al.* (1994) mostraron que la búsqueda en profundidad iterativa *online* es óptimamente eficiente para encontrar un objetivo en un árbol uniforme sin la información heurís-

GRAFOS EULERIANOS

BÚSQUEDA EN TIEMPO REAL

tica. Algunas variantes informadas sobre el tema LRTA* se han desarrollado con métodos diferentes para buscar y actualizar dentro de la parte conocida del grafo (Pemberton y Korf, 1992). Todavía no hay una buena comprensión de cómo encontrar los objetivos con eficiencia óptima cuando se usa información heurística.

BÚSQUEDA PARALELA

El tema de los algoritmos de **búsqueda paralela** no se ha tratado en el capítulo, en parte porque requiere una discusión larga de arquitecturas paralelas de computadores. La búsqueda paralela llega a ser un tema importante en IA y en informática teórica. Una introducción breve a la literatura de IA se puede encontrar en Mahanti y Daniels (1993).

EJERCICIOS



4.1 Trace cómo opera la búsqueda A* aplicada al problema de alcanzar Bucarest desde Lugoj utilizando la heurística distancia en línea recta. Es decir, muestre la secuencia de nodos que considerará el algoritmo y los valores f , g , y h para cada nodo.

4.2 El **algoritmo de camino heurístico** es una búsqueda primero el mejor en la cual la función objetivo es $f(n) = (2 - w)g(n) + wh(n)$. ¿Para qué valores del w está garantizado que el algoritmo sea óptimo? ¿Qué tipo de búsqueda realiza cuando $w = 0$?; ¿cuándo $w = 1$? y ¿cuándo $w = 2$?

4.3 Demuestre cada una de las declaraciones siguientes:

- La búsqueda primero en anchura es un caso especial de la búsqueda de coste uniforme.
- La búsqueda primero en anchura, búsqueda primero en profundidad, y la búsqueda de coste uniforme son casos especiales de la búsqueda primero el mejor.
- La búsqueda de coste uniforme es un caso especial de la búsqueda A*.



4.4 Idee un espacio de estados en el cual A*, utilizando la BÚSQUEDA-GRAFO, devuelva una solución sub-óptima con una función $h(n)$ admisible pero inconsistente.

4.5 Vimos en la página 109 que la heurística de distancia en línea recta dirige la búsqueda primero el mejor voraz por mal camino en el problema de ir de Iasi a Fagaras. Sin embargo, la heurística es perfecta en el problema opuesto: ir de Fagaras a Iasi. ¿Hay problemas para los cuales la heurística engaña en ambas direcciones?

4.6 Invente una función heurística para el 8-puzzle que a veces sobreestime, y muestre cómo puede conducir a una solución subóptima sobre un problema particular (puede utilizar un computador para ayudarse). Demuestre que, si h nunca sobreestima en más de c , A*, usando h , devuelve una solución cuyo coste excede de la solución óptima en no más de c .



4.7 Demuestre que si una heurística es consistente, debe ser admisible. Construya una heurística admisible que no sea consistente.

4.8 El problema del viajante de comercio (PVC) puede resolverse con la heurística del árbol mínimo (AM), utilizado para estimar el coste de completar un viaje, dado que ya se ha construido un viaje parcial. El coste de AM del conjunto de ciudades es

la suma más pequeña de los costos de los arcos de cualquier árbol que une todas las ciudades.

- a) Muestre cómo puede obtenerse esta heurística a partir de una versión relajada del PVC.
- b) Muestre que la heurística AM domina la distancia en línea recta.
- c) Escriba un generador de problemas para ejemplos del PVC donde las ciudades están representadas por puntos aleatorios en el cuadrado unidad.
- d) Encuentre un algoritmo eficiente en la literatura para construir el AM, y úselo con un algoritmo de búsqueda admisible para resolver los ejemplos del PVC.

4.9 En la página 122, definimos la relajación del 8-puzzle en el cual una ficha podía moverse del cuadrado A al cuadrado B si B era el blanco. La solución exacta de este problema define la **heurística de Gaschnig** (Gaschnig, 1979). Explique por qué la heurística de Gaschnig es al menos tan exacta como h_1 (fichas mal colocadas), y muestre casos donde es más exacta que h_1 y que h_2 (distancia de Manhattan). ¿Puede sugerir un modo de calcular la heurística de Gaschnig de manera eficiente?



4.10 Dimos dos heurísticas sencillas para el 8-puzzle: distancia de Manhattan y fichas mal colocadas. Varias heurísticas en la literatura pretendieron mejorarlas, véase, por ejemplo, Nilsson (1971), Mostow y Prieditis (1989), y Hansson *et al.* (1992). Pruebe estas mejoras, implementando las heurísticas y comparando el funcionamiento de los algoritmos que resultan.

4.11 Dé el nombre del algoritmo que resulta de cada uno de los casos siguientes:

- a) Búsqueda de haz local con $k = 1$.
- b) Búsqueda de haz local con $k = \infty$.
- c) Temple simulado con $T = 0$ en cualquier momento.
- d) Algoritmo genético con tamaño de la población $N = 1$.

4.12 A veces no hay una función de evaluación buena para un problema, pero hay un método de comparación bueno: un modo de decir si un nodo es mejor que el otro, sin adjudicar valores numéricos. Muestre que esto es suficiente para hacer una búsqueda primero el mejor. ¿Hay un análogo de A*?

4.13 Relacione la complejidad en tiempo de LRTA* con su complejidad en espacio.

4.14 Suponga que un agente está en un laberinto de 3x3 como el de la Figura 4.18. El agente sabe que su posición inicial es (1,1), que el objetivo está en (3,3), y que las cuatro acciones *Arriba*, *Abajo*, *Izquierda*, *Derecha* tienen sus efectos habituales a menos que estén bloqueadas por una pared. El agente *no* sabe dónde están las paredes internas. En cualquier estado, el agente percibe el conjunto de acciones legales; puede saber también si el estado ha sido visitado antes o si es un nuevo estado.

- a) Explique cómo este problema de búsqueda *online* puede verse como una búsqueda *offline* en el espacio de estados de creencia, donde el estado de creencia inicial incluye todas las posibles configuraciones del entorno. ¿Cómo de grande es el estado de creencia inicial? ¿Cómo de grande es el espacio de estados de creencia?
- b) ¿Cuántas percepciones distintas son posibles en el estado inicial?

- c) Describa las primeras ramas de un plan de contingencia para este problema. ¿Cómo de grande (aproximadamente) es el plan completo?

Nótese que este plan de contingencia es una solución para *todos los entornos posibles* que encajan con la descripción dada. Por lo tanto, intercalar la búsqueda y la ejecución no es estrictamente necesario hasta en entornos desconocidos.



- 4.15** En este ejercicio, exploraremos el uso de los métodos de búsqueda local para resolver los PVCs del tipo definido en el Ejercicio 4.8.

- Idee una aproximación de la ascensión de colinas para resolver los PVSs. Compare los resultados con soluciones óptimas obtenidas con el algoritmo A* con la heurística AM (Ejercicio 4.8).
- Idee una aproximación del algoritmo genético al problema del viajante de comercio. Compare los resultados a las otras aproximaciones. Puede consultar Larranaga *et al.* (1999) para algunas sugerencias sobre las representaciones.



- 4.16** Genere un número grande de ejemplos del 8-puzzle y de las 8-reinas y resuélvalos (donde sea posible) por la ascensión de colinas (variantes de subida más escarpada y de la primera opción), ascensión de colinas con reinicio aleatorio, y temple simulado. Mida el coste de búsqueda y el porcentaje de problemas resueltos y represente éstos gráficamente contra el costo óptimo de solución. Comente sus resultados.

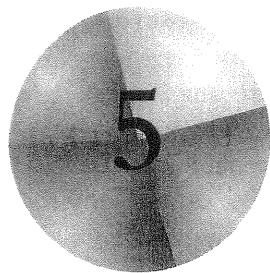


- 4.17** En este ejercicio, examinaremos la ascensión de colinas en el contexto de navegación de un robot, usando el entorno de la Figura 3.22 como un ejemplo.

- Repita el Ejercicio 3.16 utilizando la ascensión de colinas. ¿Cae alguna vez su agente en un mínimo local? ¿Es *possible* con obstáculos convexos?
- Construya un entorno no convexo poligonal en el cual el agente cae en mínimos locales.
- Modifique el algoritmo de ascensión de colinas de modo que, en vez de hacer una búsqueda a profundidad 1 para decidir dónde ir, haga una búsqueda a profundidad- k . Debería encontrar el mejor camino de k -pasos y hacer un paso sobre el camino, y luego repetir el proceso.
- ¿Hay algún k para el cual esté garantizado que el nuevo algoritmo se escape de mínimos locales?
- Explique cómo LRTA* permite al agente escaparse de mínimos locales en este caso.



- 4.18** Compare el funcionamiento de A* y BRPM sobre un conjunto de problemas generados aleatoriamente en dominios del 8-puzzle (con distancia de Manhattan) y del PVC (con AM, véase el Ejercicio 4.8). Discuta sus resultados. ¿Qué le pasa al funcionamiento de la BRPM cuando se le añade un pequeño número aleatorio a los valores heurísticos en el dominio del 8-puzzle?



Problemas de satisfacción de restricciones

En donde veremos cómo el tratar los estados como más que sólo pequeñas cajas negras conduce a la invención de una variedad de nuevos poderosos métodos de búsqueda y a un entendimiento más profundo de la estructura y complejidad del problema.

CAJA NEGRA

PROBLEMA DE SATISFACCIÓN DE RESTRICCIONES

REPRESENTACIÓN

Los Capítulos 3 y 4 exploraron la idea de que los problemas pueden resolverse buscando en un espacio de **estados**. Estos estados pueden evaluarse con heurísticas específicas del dominio y probados para ver si son estados objetivo. Desde el punto de vista del algoritmo de búsqueda, sin embargo, cada estado es una **caja negra** sin la estructura perceptible interna. Se representa por una estructura de datos arbitraria a la que se puede acceder sólo con las rutinas *específicas de problema* (la función sucesor, función heurística, y el test objetivo).

Este capítulo examina **problemas de satisfacción de restricciones**, cuyos estados y test objetivo forman una **representación** muy simple, estándar y estructurada (Sección 5.1). Los algoritmos de búsqueda se pueden definir aprovechándose de la estructura de los estados y utilizan las heurísticas de *propósito general* más que heurísticas *específicas de problema* para así permitir la solución de problemas grandes (Secciones 5.2-5.3). Quizá lo más importante sea que la representación estándar del test objetivo revela la estructura del problema (Sección 5.4). Esto conduce a métodos de descomposición de problemas y a una comprensión de la conexión entre la estructura de un problema y la dificultad para resolverlo.

5.1 Problemas de satisfacción de restricciones

VARIABLES

RESTRICIONES

Formalmente, un **problema de satisfacción de restricciones** (o PSR) está definido por un conjunto de **variables**, X_1, X_2, \dots, X_n , y un conjunto de **restricciones**, C_1, C_2, \dots, C_m . Cada

DOMINIO	variable X_i tiene un dominio no vacío D_i de valores posibles. Cada restricción C_i implica algún subconjunto de variables y especifica las combinaciones aceptables de valores para ese subconjunto. Un estado del problema está definido por una asignación de valores a unas o todas las variables, $\{X_i = v_i, X_j = v_j, \dots\}$. A una asignación que no viola ninguna restricción se llama asignación consistente o legal. Una asignación completa es una asignación en la que se menciona cada variable, y una solución de un PSR es una asignación completa que satisface todas las restricciones. Algunos problemas de satisfacción de restricciones (PSRs) también requieren una solución que maximiza una función objetivo .
VALORES	
ASIGNACIÓN	
CONSISTENTE	
FUNCIÓN OBJETIVO	

¿Qué significa todo esto? Suponga que, cansados de Rumanía, miramos un mapa de Australia que muestra cada uno de sus estados y territorios, como en la Figura 5.1(a), y que nos encargan la tarea de colorear cada región de rojo, verde o azul de modo que ninguna de las regiones vecinas tenga el mismo color. Para formularlo como un PSR, definimos las variables de las regiones: AO , TN , Q , NGS , V , AS y T . El dominio de cada variable es el conjunto $\{\text{rojo}, \text{verde}, \text{azul}\}$. Las restricciones requieren que las regiones vecinas tengan colores distintos; por ejemplo, las combinaciones aceptables para AO y TN son los pares

$$\{(rojo, verde), (rojo, azul), (verde, rojo), (verde, azul), (azul, rojo), (azul, verde)\}$$

(La restricción puede también representarse más sucintamente como la desigualdad $AO \neq TN$, a condición de que el algoritmo de satisfacción de restricciones tenga algún modo de evaluar tales expresiones.) Hay muchas soluciones posibles, como

$$\{AO = \text{rojo}, TN = \text{verde}, Q = \text{rojo}, NGS = \text{verde}, V = \text{rojo}, AS = \text{azul}, T = \text{rojo}\}$$

Es bueno visualizar un PSR como un **grafo de restricciones**, como el que se muestra en la Figura 5.1(b). Los nodos del grafo corresponden a variables del problema y los arcos corresponden a restricciones.

Tratar un problema como un PSR confiere varias ventajas importantes. Como la representación del estado en un PSR se ajusta a un modelo estándar (es decir, un conjunto de variables con valores asignados) la función sucesor y el test objetivo pueden escribirse de un modo genérico para que se aplique a todo PSR. Además, podemos desarrollar heurísticas eficaces y genéricas que no requieran ninguna información adicional ni experta del dominio específico. Finalmente, la estructura del grafo de las restricciones puede usarse para simplificar el proceso de solución, en algunos casos produciendo una reducción exponencial de la complejidad. La representación PSR es la primera, y más simple, de una serie de esquemas de representación que serán desarrollados a través de los capítulos del libro.

Es bastante fácil ver que a un PSR se le puede dar una **formulación incremental** como en un problema de búsqueda estándar:

- **Estado inicial:** la asignación vacía $\{\}$, en la que todas las variables no están asignadas.
- **Función de sucesor:** un valor se puede asignar a cualquier variable no asignada, a condición de que no suponga ningún conflicto con variables antes asignadas.
- **Test objetivo:** la asignación actual es completa.
- **Costo del camino:** un coste constante (por ejemplo, 1) para cada paso.

Cada solución debe ser una asignación completa y por lo tanto aparecen a profundidad n si hay n variables. Además, el árbol de búsqueda se extiende sólo a profundidad n . Por

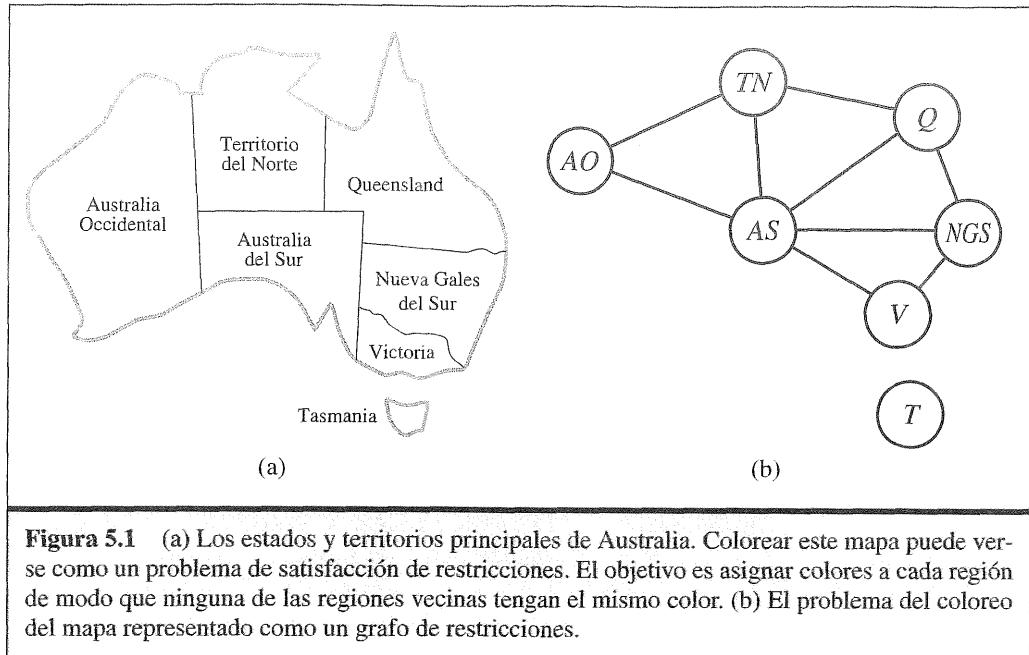


Figura 5.1 (a) Los estados y territorios principales de Australia. Colorear este mapa puede verse como un problema de satisfacción de restricciones. El objetivo es asignar colores a cada región de modo que ninguna de las regiones vecinas tengan el mismo color. (b) El problema del coloreo del mapa representado como un grafo de restricciones.

estos motivos, los algoritmos de búsqueda primero en profundidad son populares para PSRs. (Véase la Sección 5.2.) También *el camino que alcanza una solución es irrelevante*. De ahí, que podemos usar también una **formulación completa de estados**, en la cual cada estado es una asignación completa que podría o no satisfacer las restricciones. Los métodos de búsqueda local trabajan bien para esta formulación. (Véase la Sección 5.3.)

La clase más simple de PSR implica variables **discretas** y **dominios finitos**. Los problemas de coloreo del mapa son de esta clase. El problema de las 8-reinas descrito en el Capítulo 3 puede también verse como un PSR con dominio finito, donde las variables Q_1, Q_2, \dots, Q_8 son las posiciones de cada reina en las columnas 1..., 8 y cada variable tiene el dominio $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Si el tamaño máximo del dominio de cualquier variable, en un PSR, es d , entonces el número de posibles asignaciones completas es $O(d^n)$, es decir, exponencial en el número de variables. Los PSR con dominio finito incluyen a los **PSRs booleanos**, cuyas variables pueden ser *verdaderas* o *falsas*. Los PSRs booleanos incluyen como casos especiales algunos problemas NP-completos, como 3SAT. (Véase el Capítulo 7.) En el caso peor, por lo tanto, no podemos esperar resolver los PSRs con dominios finitos en menos de un tiempo exponencial. En la mayoría de las aplicaciones prácticas, sin embargo, los algoritmos para PSR de uso general pueden resolver problemas de *órdenes de magnitud* más grande que los resolvibles con los algoritmos de búsqueda de uso general que vimos en el Capítulo 3.

Las variables discretas pueden tener también **dominios infinitos** (por ejemplo, el conjunto de números enteros o de cadenas). Por ejemplo, cuando programamos trabajos de la construcción en un calendario, la fecha de comienzo de cada trabajo es una variable y los valores posibles son números enteros de días desde la fecha actual. Con dominios infinitos, no es posible describir restricciones enumerando todas las combinaciones permitidas de valores. En cambio, se debe utilizar un **lenguaje de restricción**. Por ejemplo,



DOMINIOS FINITOS

PSRs BOOLEANOS

DOMINIOS INFINITOS

LENGUAJE DE
RESTRICCIÓN

RESTRICCIONES
LINEALESRESTRICCIONES NO
LINEALESDOMINIOS
CONTINUOSPROGRAMACIÓN
LINEAL

RESTRICCIÓN UNARIA

RESTRICCIÓN BINARIA

CRIPTO-ARITMÉTICO

VARIABLES
AUXILIARESHIPER-GRAFO DE
RESTRICCIONES

si $Trabajo_1$, que utiliza cinco días, debe preceder a $Trabajo_3$, entonces necesitaríamos un lenguaje de restricción de desigualdades algebraicas como $ComienzoTrabajo_1 + 5 \leq ComienzoTrabajo_3$. Tampoco es posible resolver tales restricciones enumerando todas las asignaciones posibles, porque hay infinitas. Existen algoritmos solución especiales (de los que no hablaremos aquí) para **restricciones lineales** sobre variables enteras (es decir, restricciones, como la anterior, en la que cada variable aparece de forma lineal). Puede de demostrar que no existe un algoritmo para resolver **restricciones no lineales** generales sobre variables enteras. En algunos casos, podemos reducir los problemas de restricciones enteras a problemas de dominio finito simplemente acotando los valores de todas las variables. Por ejemplo, en un problema de programación, podemos poner una cota superior igual a la longitud total de todos los trabajos a programar.

Los problemas de satisfacción de restricciones con **dominios continuos** son muy comunes en el mundo real y son ampliamente estudiados en el campo de la investigación operativa. Por ejemplo, la programación de experimentos sobre el Telescopio Hubble requiere el cronometraje muy preciso de las observaciones; al comienzo y al final de cada observación y la maniobra son variables continuas que deben obedecer a una variedad de restricciones astronómicas, prioritarias y potentes. La categoría más conocida de PSRs en dominios continuos son los problemas de **programación lineal**, en donde las restricciones deben ser desigualdades lineales que forman una región *convexa*. Los problemas de programación lineal pueden resolverse en tiempo polinomial en el número de variables. Los problemas con tipos diferentes de restricciones y funciones objetivo también se han estudiado: programación cuadrática, programación cónica de segundo orden, etcétera.

Además del examen de los tipos de variables que pueden aparecer en los PSRs, es útil ver los tipos de restricciones. El tipo más simple es la **restricción unaria**, que restringe los valores de una sola variable. Por ejemplo, podría ser el caso en los que a los australianos del Sur les disgustara el color *verde*. Cada restricción unaria puede eliminarse simplemente con el preproceso del dominio de la variable quitando cualquier valor que viole la restricción. Una **restricción binaria** relaciona dos variables. Por ejemplo, $AS \neq NGS$ es una restricción binaria. Un PSR binario es un problema con restricciones sólo binarias; y puede representarse como un grafo de restricciones, como en la Figura 5.1(b).

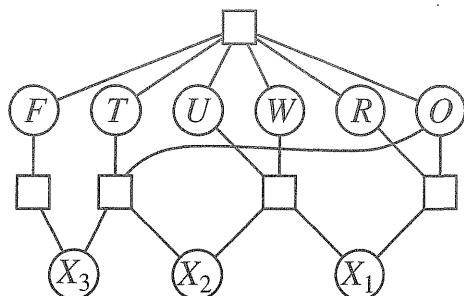
Las restricciones de orden alto implican tres o más variables. Un ejemplo familiar es el que proporcionan los puzzles **cripto-aritméticos**. (Véase la Figura 5.2(a).) Es habitual insistir que cada letra en un puzzle cripto-aritmético represente un dígito diferente. Para el caso de la Figura 5.2(a), éste sería representado como la restricción de seis variables $TodasDif(F, T, U, W, R, O)$. O bien, puede representarse por una colección de restricciones binarias como $F \neq T$. Las restricciones añadidas sobre las cuatro columnas del puzzle también implican varias variables y pueden escribirse como

$$\begin{aligned} O + O &= R + 10 \cdot X_1 \\ X_1 + W + W &= U + 10 \cdot X_2 \\ X_2 + T + T &= O + 10 \cdot X_3 \\ X_3 &= F \end{aligned}$$

donde X_1 , X_2 , y X_3 son **variables auxiliares** que representan el dígito (0 o 1) transferido a la siguiente columna. Las restricciones de orden alto pueden representarse en un **hiper-grafo de restricciones**, como el de la Figura 5.2(b). El lector habrá notado que

$$\begin{array}{r}
 T \ W \ O \\
 + T \ W \ O \\
 \hline
 F \ O \ U \ R
 \end{array}$$

(a)



(b)

Figura 5.2 (a) Un problema cripto-aritmético. Cada letra significa un dígito distinto; el objetivo es encontrar una sustitución de las letras por dígitos tal que la suma que resulta es aritméticamente correcta, con la restricción añadida de que no se permite la administración de ceros. (b) El hiper-grafo de restricciones para el problema cripto-aritmético muestra la restricción *TodasDif* así como las restricciones añadidas en las columnas. Cada restricción es una caja cuadrada relacionada con las variables que restringe.

la restricción *TodasDif* puede dividirse en restricciones binarias ($F \neq T, F \neq U$, etcétera). De hecho, como en el Ejercicio 5.11, se pide demostrar que la restricción de dominio finito y orden alto puede reducirse a un conjunto de restricciones binarias si se introducen suficientes variables auxiliares. A causa de esto, en este capítulo trataremos sólo restricciones binarias.

Las restricciones que hemos descrito hasta ahora han sido todas restricciones **absolutas**, la violación de las cuales excluye una solución potencial. Muchos PSRs del mundo real incluyen restricciones de **preferencia** que indican qué soluciones son preferidas. Por ejemplo, en un problema de horario de la universidad, el profesor X quizás prefiera enseñar por la mañana mientras que un profesor Y prefiere enseñar por la tarde. Un horario que tenga al profesor X enseñando a las dos de la tarde sería una solución (a menos que el profesor X resulte ser el director del departamento), pero no sería un óptimo. Las restricciones de preferencia pueden a menudo codificarse como costos sobre las asignaciones de variables individuales (por ejemplo, asignar un hueco por la tarde para el profesor X cuesta dos puntos contra la función objetivo total, mientras que un hueco de mañana cuesta uno). Con esta formulación, los PSRs con preferencias pueden resolverse utilizando métodos de búsqueda de optimización, basados en caminos o locales. No hablamos de tales PSRs fuera de este capítulo, pero sí proporcionamos algunas líneas de trabajo en la sección de notas bibliográficas.

PREFERENCIA

5.2 Búsqueda con vuelta atrás para PSR

La sección anterior dio una formulación de PSRs como problemas de búsqueda. Usando esta formulación, cualquiera de los algoritmos de búsqueda de los Capítulos 3 y 4

pueden resolver los PSRs. Suponga que aplicamos la búsqueda primero en anchura a la formulación del PSR genérico de la sección anterior. Rápidamente notamos algo terrible: el factor de ramificación en el nivel superior es de nd , porque cualquiera de los d valores se puede asignar a cualquiera de las n variables. En el siguiente nivel, el factor de ramificación es $(n-1)d$, etcétera para los n niveles. ¡Generamos un árbol con $n! \cdot d^n$ hojas, aunque haya sólo d^n asignaciones posibles completas!

COMMUTATIVIDAD



BÚSQUEDA CON VUELTA ATRÁS

Nuestra formulación del problema, aparentemente razonable pero ingenua, no ha hecho caso de una propiedad crucial común en todos los PSRs: la **commutatividad**. Un problema es comutativo si el orden de aplicación de cualquier conjunto de acciones no tiene ningún efecto sobre el resultado. Éste es el caso de los PSRs porque, asignando valores a variables, alcanzamos la misma asignación parcial sin tener en cuenta el orden. Por lo tanto, *todos los algoritmos de búsqueda para el PSR generan los sucesores considerando asignaciones posibles para sólo una variable en cada nodo del árbol de búsqueda*. Por ejemplo, en el nodo raíz de un árbol de búsqueda para colorear el mapa de Australia, podríamos tener una opción entre $AS = \text{rojo}$, $AS = \text{verde}$, y $AS = \text{azul}$, pero nunca elegiríamos entre $AS = \text{rojo}$ y $AO = \text{azul}$. Con esta restricción, el número de hojas es d^n , como era de esperar.

El término **búsqueda con vuelta atrás** se utiliza para la búsqueda primero en profundidad que elige valores para una variable a la vez y vuelve atrás cuando una variable no tiene ningún valor legal para asignarle. En la Figura 5.3 se muestra este algoritmo. Notemos que usa, en efecto, el método uno a la vez de la generación de sucesor incremental descrita en la página 86. También, extiende la asignación actual para generar un sucesor, más que volver a copiarlo. Como la representación de los PSRs está estandarizada, no hay ninguna necesidad de proporcionar a la BÚSQUEDA-CON-VUELTA-ATRÁS un estado inicial del dominio específico, una función sucesor, o un test del objetivo. En la Figura 5.4 se muestra parte del árbol de búsqueda para el problema de Australia, en donde hemos asignado variables en el orden AO, TN, Q...

función BÚSQUEDA-CON-VUELTA-ATRÁS(*psr*) **devuelve** una solución, o fallo
devolver VUELTA-ATRÁS-RECURSIVA({ }, *psr*)

función VUELTA-ATRÁS-RECURSIVA(*asignación*, *psr*) **devuelve** una solución, o fallo
si *asignación* es completa **entonces devolver** *asignación*
var \leftarrow SELECCIONA-VARIABLE-NOASIGNADA(VARIABLES[*psr*], *asignación*, *psr*)
para cada *valor* **en** ORDEN-VALORES-DOMINIO(*var*, *asignación*, *psr*) **hacer**
si *valor* es consistente con *asignación* de acuerdo a las RESTRICCIONES[*psr*] **entonces**
 añadir {*var* = *valor*} a *asignación*
 resultado \leftarrow VUELTA-ATRÁS-RECURSIVA(*asignación*, *psr*)
 si *resultado* \neq fallo **entonces devolver** *resultado*
 borrar {*var* = *valor*} de *asignación*
devolver fallo

Figura 5.3 Un algoritmo simple de vuelta atrás para problemas de satisfacción de restricciones. El algoritmo se modela sobre la búsqueda primero en profundidad recursivo del Capítulo 3. Las funciones SELECCIONA-VARIABLE-NOASIGNADA y ORDEN-VALORES-DOMINIO pueden utilizarse para implementar las heurísticas de propósito general discutidas en el texto.

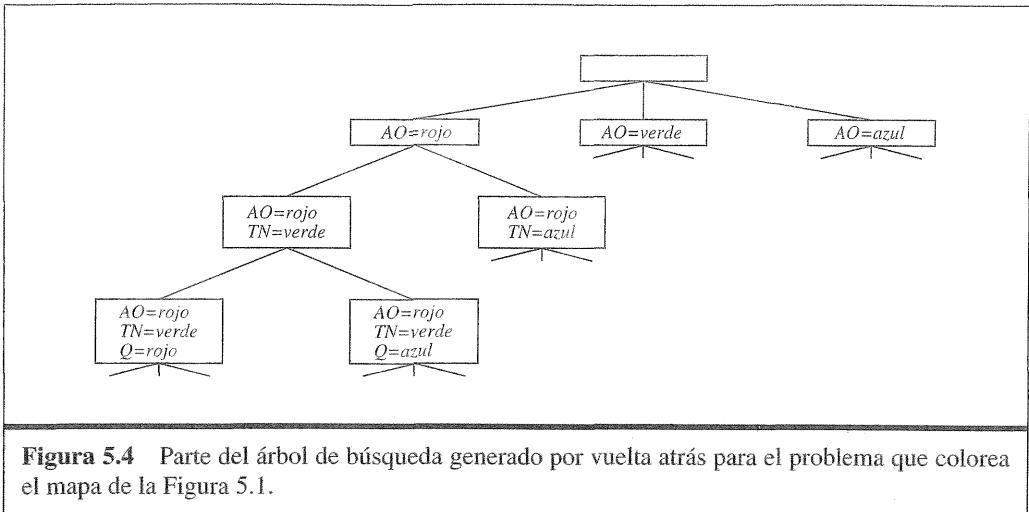


Figura 5.4 Parte del árbol de búsqueda generado por vuelta atrás para el problema que colorea el mapa de la Figura 5.1.

La vuelta atrás sencilla es un algoritmo sin información en la terminología del Capítulo 3, así que no esperamos que sea muy eficaz para problemas grandes. En la primera columna de la Figura 5.5 se muestran los resultados para algunos problemas y se confirman nuestras expectativas.

En el Capítulo 4 remediamos el funcionamiento pobre de los algoritmos de búsqueda sin información suministrándoles funciones heurísticas específicas del dominio obtenidas de nuestro conocimiento del problema. Resulta que podemos resolver los PSRs de

Problema	Vuelta atrás	VA + MVR	Comprobación hacia delante	CD + MVR	Mínimo conflicto
EE.UU. <i>n</i> -reinas	(> 1.000K) (> 40.000K)	(> 1.000K) 13.500K	2K (> 40.000K)	60 817K	64 4K
Zebra	3.859K	1K	35K	0,5K	2K
Aleatorio 1	415K	3K	26K	2K	
Aleatorio 2	942K	27K	77K	15K	

Figura 5.5 Comparación de varios algoritmos de PSR sobre varios problemas. Los algoritmos, de izquierda a derecha, son vuelta atrás simple, vuelta atrás con la heurística MVR, comprobación hacia delante, comprobación hacia delante con MVR y búsqueda local de conflictos mínimos. En cada celda está el número medio de comprobaciones consistentes (sobre cinco ejecuciones) requerido para resolver el problema; notemos que todas las entradas excepto las dos de la parte superior derecha están en miles (K). Los números en paréntesis significan que no se ha encontrado ninguna respuesta en el número asignado de comprobaciones. El primer problema es colorear, con cuatro colores, los 50 estados de los Estados Unidos de América. Los problemas restantes se han tomado de Bacchus y van Run (1995), Tabla 1. El segundo problema cuenta el número total de comprobaciones requeridas para resolver todos los problemas de *n*-reinas para *n* de dos a 50. El tercero es el «puzzle Zebra», descrito en el Ejercicio 5.13. Los dos últimos son problemas artificiales aleatorios. (Mínimos conflictos no se ejecutó sobre estos.) Los resultados sugieren que la comprobación hacia delante con la heurística MVR es mejor sobre todos estos problemas que los otros algoritmos con vuelta atrás, pero no siempre es mejor que los de búsqueda local de mínimo conflicto.

manera eficiente sin tal conocimiento específico del dominio. En cambio, encontramos métodos de propósito general que proporcionan las siguientes preguntas:

1. ¿Qué variable debe asignarse después, y en qué orden deberían intentarse sus valores?
2. ¿Cuáles son las implicaciones de las asignaciones de las variables actuales para las otras variables no asignadas?
3. Cuándo un camino falla, es decir, un estado alcanzado en el que una variable no tiene ningún valor legal, ¿puede la búsqueda evitar repetir este fracaso en caminos siguientes?

Las subsecciones siguientes contestan a cada una de estas preguntas.

Variable y ordenamiento de valor

El algoritmo con vuelta atrás contiene la línea

```
var ← SELECCIONA-VARIABLE-NOASIGNADA(VARIABLES[psr], asignación, psr)
```

Por defecto, SELECCIONA-VARIABLE-NOASIGNADA simplemente selecciona la siguiente variable no asignada en el orden dado por la lista VARIABLES[*psr*]. Esta variable estática, rara vez ordenada, da como resultado una búsqueda más eficiente. Por ejemplo, después de las asignaciones para $AO = \text{rojo}$ y $T = \text{verde}$, hay un sólo valor posible para AS , entonces tiene sentido asignar $AS = \text{azul}$ a continuación más que asignar un valor a Q . De hecho, después de asignar AS , las opciones para Q , NGS , y V están forzadas. Esta idea intuitiva (escoger la variable con menos valores «legales») se llama heurística de **mínimos valores restantes** (MVR). También llamada heurística «variable más restringida» o «primero en fallar», este último es porque escoge una variable que con mayor probabilidad causará pronto un fracaso, con lo cual podamos el árbol de búsqueda. Si hay una variable X con cero valores legales restantes, la heurística MVR seleccionará X y el fallo será descubierto inmediatamente (evitando búsquedas inútiles por otras variables que siempre fallarán cuando se seleccione X finalmente). La segunda columna de la Figura 5.5, etiquetada con VA + MVR, muestra el funcionamiento de esta heurística. El funcionamiento es de tres a 3.000 veces mejor que la vuelta atrás simple, según el problema. Notemos que nuestra medida de rendimiento no hace caso del coste suplementario de calcular los valores heurísticos; la siguiente subsección describe un método que maneja este coste.

La heurística MVR no ayuda en absoluto en la elección de la primera región a colorear en Australia, porque al principio cada región tiene tres colores legales. En este caso, el **grado heurístico** es más práctico. Intenta reducir el factor de ramificación sobre futuras opciones seleccionando la variable, entre las variables no asignadas, que esté implicada en el mayor número de restricciones. En la Figura 5.1, AS es la variable con el grado más alto, cinco; las otras variables tienen grado dos o tres, excepto T , que tiene cero. De hecho, una vez elegida AS , aplicando el grado heurístico que resuelve el problema sin pasos en falso puede elegir cualquier color consistente en cada punto seleccionado y todavía llegar a una solución sin vuelta atrás. La heurística del mínimo de valores restantes es por lo general una guía más poderosa, pero el grado heurístico puede ser útil como un desempate.

VALOR MENOS
RESTRINGIDO

Una vez que se selecciona una variable, el algoritmo debe decidir el orden para examinar sus valores. Para esto, la heurística del **valor menos restringido** puede ser eficaz en algunos casos. Se prefiere el valor que excluye las pocas opciones de las variables vecinas en el grafo de restricciones. Por ejemplo, supongamos que en la Figura 5.1 hemos generado la asignación parcial con $AO = \text{roja}$ y $TN = \text{verde}$, y que nuestra siguiente opción es para Q . Azul sería una opción mala, porque elimina el último valor legal del vecino de Q , SA . La heurística del valor menos restringido, por lo tanto, prefiere rojo al azul. En general, la heurística trata de dejar la flexibilidad máxima de las asignaciones de las variables siguientes. Desde luego, si tratamos de encontrar todas las soluciones a un problema, no sólo la primera, entonces el orden no importa porque tenemos que considerar cada valor de todos modos. Ocurre lo mismo si no hay soluciones al problema.

Propagación de la información a través de las restricciones

Hasta ahora nuestro algoritmo de búsqueda considera las restricciones sobre una variable sólo cuando la variable es elegida por **SELECCIONA-VARIABLE-NOASIGNADA**. Pero mirando algunas restricciones antes en la búsqueda, o incluso antes de que haya comenzado la búsqueda, podemos reducir drásticamente el espacio de ésta.

COMPROBACIÓN
HACIA DELANTE

Comprobación hacia delante

Otra manera para usar mejor las restricciones durante la búsqueda se llama **comprobación hacia delante**. Siempre que se asigne una variable X , el proceso de comprobación hacia delante mira cada variable no asignada Y que esté relacionada con X por una restricción y suprime del dominio de Y cualquier valor que sea inconsistente con el valor elegido para X . La Figura 5.6 muestra el progreso de una búsqueda, que colorea un mapa, con la comprobación hacia delante. Hay dos puntos importantes que debemos destacar sobre este ejemplo. Primero, notemos que después de asignar $AO = \text{rojo}$ y $Q = \text{verde}$, los dominios de TN y AS se reducen a un solo valor; hemos eliminado las ramificaciones de estas variables totalmente propagando la información de AO y Q . La

	AO	TN	Q	NGS	V	AS	T
Dominios iniciales	R V A	R V A	R V A	R V A	R V A	R V A	R V A
Después de $AO=\text{rojo}$	(R)	V A	R V A	R V A	R V A	V A	R V A
Después de $Q=\text{verde}$	(R)	A	(V)	R	A	R V A	A
Después de $V=\text{azul}$	(R)	A	(V)	R		(A)	R V A

Figura 5.6 Progreso de una búsqueda, que colorea un mapa, con comprobación hacia delante. $AO = \text{rojo}$ se asigna primero; entonces la comprobación hacia delante suprime *rojo* de los dominios de las variables vecinas TN y AS . Después $Q = \text{verde}$; *verde* se suprime de los dominios TN , AS , y NGS . Después $V = \text{azul}$; *azul* se suprime del dominio de NGS y AS , y se obtiene AS sin valores legales.

heurística MVR, la cual es una compañera obvia para la comprobación hacia delante, seleccionaría automáticamente AS y TN después. (En efecto, podemos ver la comprobación hacia delante como un modo eficiente de incrementar el cálculo de la información que necesita la heurística MVR para hacer su trabajo.) Un segundo punto a tener en cuenta es que, después de $V = \text{azul}$, el dominio de SA está vacío. Por eso, la comprobación hacia delante ha descubierto que la asignación parcial $\{AO = \text{rojo}, Q = \text{verde}, V = \text{azul}\}$ es inconsistente con las restricciones del problema, y el algoritmo volverá atrás inmediatamente.

Propagación de restricciones

PROPAGACIÓN DE
RESTRICCIONES

Aunque la comprobación hacia delante descubre muchas inconsistencias, no descubre todas. Por ejemplo, considere la tercera fila de la Figura 5.6. Muestra que cuando AO es *rojo* y Q es *verde*, tanto TN como AS están obligadas a ser azules. Pero son adyacentes y por tanto no pueden tener el mismo valor. La comprobación hacia delante no la descubre como una inconsistencia, porque no mira lo bastante lejos. La **propagación de restricciones** es el término general para la propagación de las implicaciones de una restricción sobre una variable en las otras variables; en este caso necesitamos propagar desde AO y Q a TN y AS , (como se hizo con la comprobación hacia delante) y luego a la restricción entre TN y AS para descubrir la inconsistencia. Y queremos hacer esto rápido: no es nada bueno reducir la cantidad de búsqueda si gastamos más tiempo propagando restricciones que el que hubiéramos gastado haciendo una búsqueda simple.

ARCO CONSISTENTE

La idea del **arco consistente** proporciona un método rápido de propagación de restricciones que es considerablemente más potente que la comprobación hacia delante. Aquí, «el arco» se refiere a un arco dirigido en el grafo de restricciones, como el arco de AS a NGS . Considerando los dominios actuales de AS y NGS , el arco es consistente si, para *todo* valor x de AS , hay *algún* valor y de NGS que es consistente con x . En la tercera fila de la Figura 5.6, los dominios actuales de AS y NGS son $\{\text{azul}\}$ y $\{\text{rojo}, \text{azul}\}$, respectivamente. Para $AS = \text{azul}$, hay una asignación consistente para NGS , a saber, $NGS = \text{roja}$; por lo tanto, el arco de AS a NGS es consistente. Por otra parte, el arco inverso desde NGS a AS no es consistente: para la asignación $NGS = \text{azul}$, no hay ninguna asignación consistente para AS . El arco puede hacerse consistente suprimiendo el valor *azul* del dominio de NGS .

Podemos aplicar también el arco consistente al arco de AS a TN en la misma etapa del proceso de búsqueda. La tercera fila de la tabla en la Figura 5.6 muestra que ambas variables tienen el dominio $\{\text{azul}\}$. El resultado es que *azul* debe suprimirse del dominio de AS , dejando el dominio vacío. Así, la aplicación del arco consistente ha causado la detección temprana de una inconsistencia que no es descubierta por la comprobación hacia delante pura.

La comprobación de la consistencia del arco puede aplicarse como un paso de preproceso antes de comenzar el proceso de búsqueda, o como un paso de propagación (como la comprobación hacia delante) después de cada asignación durante la búsqueda (a veces se llama a este último algoritmo MCA, *Mantenimiento de la Consistencia del Arco*). En uno u otro caso, el proceso debe aplicarse *repetidamente* hasta que no permanezcan

más inconsistencias. Esto es porque, siempre que un valor se suprime del dominio de alguna variable para quitar una inconsistencia de un arco, una nueva inconsistencia de arco podría surgir en arcos que señalan a aquella variable. El algoritmo completo para la consistencia de arco, AC-3, utiliza una cola para guardar los arcos que tienen que comprobarse (véase la Figura 5.7). Cada arco (X_i, X_j) se quita sucesivamente de la agenda y se comprueba; si cualquier valor requiere ser suprimido del dominio de X_i , entonces cada arco (X_k, X_i) señalando a X_i debe ser reinsertado sobre la cola para su comprobación. La complejidad de la comprobación de consistencia de arco puede analizarse como sigue: un PSR binario tiene a lo más $O(n^2)$ arcos; cada arco (X_k, X_i) puede insertarse en la agenda sólo d veces, porque X_i tiene a lo más d valores para suprimir; la comprobación de la consistencia de un arco puede hacerse en $O(d^2)$ veces; entonces el tiempo total, en el caso peor, es $O(n^2d^3)$. Aunque sea considerablemente más costoso que la comprobación hacia delante, el coste suplementario por lo general vale la pena¹.

Como los PSRs incluyen a 3SAT como un caso especial, no esperamos encontrar un algoritmo de tiempo polinomial que puede decidir si un PSR es consistente. De ahí, deducimos que la consistencia de arco no revela todas las inconsistencias posibles. Por ejemplo, en la Figura 5.1, la asignación parcial $\{AO = \text{rojo}, NGS = \text{rojo}\}$ es inconsistente, pero AC-3 no la encuentra. Las formas más potentes de la propagación pueden definirse utilizando la noción llamada ***k*-consistencia**. Un PSR es *k*-consistente si, para cualquier conjunto de $k - 1$ variables y para cualquier asignación consistente a esas variables, siempre se puede asignar un valor consistente a cualquier *k*-ésima variable.

K-CONSISTENCIA

función $AC-3(psr)$ **devuelve** el PSR, posiblemente con dominio reducido
entradas: psr , un PSR binario con variables $\{X_1, X_2, \dots, X_n\}$
variables locales: $cola$, una cola de arcos, inicialmente todos los arcos del psr

mientras $cola$ es no vacía **hacer**
 $(X_i, X_j) \leftarrow \text{BORRAR-PRIMERO}(cola)$
si $\text{BORRAR-VALORES-INCONSISTENTES}(X_i, X_j)$ **entonces**
para cada X_k en $\text{VECINOS}[X_i]$ **hacer**
 añadir (X_k, X_i) a la $cola$

función $\text{BORRAR-VALORES-INCONSISTENTES}(X_i, X_j)$ **devuelve** verdadero si y sólo si hemos borrado un valor
 $borrado \leftarrow \text{falso}$
para cada x en $\text{DOMINIO}[X_i]$ **hacer**
si no hay un y en $\text{DOMINIO}[X_j]$ que permita a (x,y) satisfacer la restricción entre X_i y X_j
entonces borrar x de $\text{DOMINIO}[X_i]$; $borrado \times \text{verdadero}$
devolver $borrado$

Figura 5.7 El algoritmo AC-3 de la consistencia del arco. Después de aplicar AC-3, cada arco es arco-consistente, o alguna variable tiene un dominio vacío, indicando que el PSR no puede hacerse arco-consistente (y así el PSR no puede resolverse). El nombre «AC-3» fue usado por el inventor del algoritmo (Mackworth, 1977) porque era la tercera versión desarrollada en el artículo.

¹ El algoritmo AC-4, debido a Mohr y Henderson (1986), se ejecuta en $O(n^2d^2)$. Véase el Ejercicio 5.10.

CONSISTENCIA DE NODO

CONSISTENCIA DE CAMINO

FUERTEMENTE
K-CONSISTENTE

Por ejemplo, la 1-consistencia significa que cada variable individual, por sí mismo, es consistente; también llamado **consistencia de nodo**. La 2-consistencia es lo mismo que la consistencia de arco. La 3-consistencia significa que cualquier par de variables adyacentes pueden siempre extenderse a una tercera variable vecina; también llamada **consistencia de camino**.

Un grafo es **fuertemente k -consistente** si es k -consistente y también $(k - 1)$ -consistente, $(k - 2)$ -consistente..., hasta 1-consistente. Ahora supongamos que tenemos un PSR con n nodos y lo hacemos fuertemente n -consistente (es decir, fuertemente k -consistente para $k = n$). Podemos resolver el problema sin vuelta atrás. Primero, elegimos un valor consistente para X_1 . Entonces tenemos garantizado el poder elegir un valor para X_2 porque el grafo es 2-consistente, para X_3 porque es 3-consistente, etcétera. Para cada variable X_i , tenemos sólo que averiguar los valores de d , en el dominio, para encontrar un valor consistente con X_1, \dots, X_{i-1} . Tenemos garantizado encontrar una solución en $O(nd)$. Desde luego, no tenemos ningún tiempo libre: cualquier algoritmo para establecer la n -consistencia debe llevar un tiempo exponencial en n , en el caso peor.

Hay una amplia diferencia entre consistencia de arco y n -consistencia: ejecutar controles de consistencia fuerte llevará más tiempo, pero tendrá un efecto mayor en la reducción del factor de ramificación y en descubrir asignaciones parciales inconsistentes. Es posible calcular el valor k más pequeño tal que al ejecutar la k -consistencia asegura que el problema puede resolverse sin volver atrás (véase la Sección 5.4), pero a menudo es poco práctico. En la práctica, la determinación del nivel apropiado de comprobación de la consistencia es sobre todo una ciencia empírica.

Manejo de restricciones especiales

Cierto tipo de restricciones aparecen con frecuencia en problemas reales y pueden manejararse utilizando algoritmos de propósito especial más eficientes que los métodos de propósito general descritos hasta ahora. Por ejemplo, la restricción *Todasdif* dice que todas las variables implicadas deben tener valores distintos (como en el problema criptográfico). Una forma simple de detección de inconsistencias para la restricción *Todasdif* trabaja como sigue: si hay m variables implicadas en la restricción, y si tienen n valores posibles distintos, y $m > n$, entonces la restricción no puede satisfacerse.

Esto nos lleva al algoritmo simple siguiente: primero, quite cualquier variable en la restricción que tenga un dominio con una sola posibilidad, y suprima el valor de esa variable de los dominios de las variables restantes. Repetir mientras existan variables con una sola posibilidad. Si en algún momento se produce un dominio vacío o hay más variables que valores en el dominio, entonces se ha descubierto una inconsistencia.

Podemos usar este método para detectar la inconsistencia en la asignación parcial $\{AO = \text{rojo}, NGS = \text{rojo}\}$ para la Figura 5.1. Notemos que las variables *SA*, *TN* y *Q* están efectivamente relacionadas por una restricción *Todasdif* porque cada par debe ser de un color diferente. Después de aplicar AC-3 con la asignación parcial, el dominio de cada variable se reduce a $\{\text{verde, azul}\}$. Es decir, tenemos tres variables y sólo dos colores, entonces se viola la restricción *Todasdif*. Así, un procedimiento simple de consistencia para una restricción de orden alto es a veces más eficaz que la aplicación de la consistencia de arco a un conjunto equivalente de restricciones binarias.

Quizá la restricción de orden alto más importante es la restricción de recursos, a veces llamada restricción *como-máximo*. Por ejemplo, PA_1, \dots, PA_4 denotan los números de personas asignadas a cada una de las cuatro tareas. La restricción que no asigna más de 10 personas en total, se escribe $\text{como-máximo}(10, PA_1, PA_2, PA_3, PA_4)$. Se puede descubrir una inconsistencia simplemente comprobando la suma de los valores mínimos de los dominios actuales; por ejemplo, si cada variable tiene el dominio $\{3, 4, 5, 6\}$, la restricción *como-máximo* no puede satisfacerse. También podemos hacer cumplir la consistencia suprimiendo el valor máximo de cualquier dominio si no es consistente con los valores mínimos de los otros dominios. Así, si cada variable, en nuestro ejemplo, tiene el dominio $\{2, 3, 4, 5, 6\}$, los valores 5 y 6 pueden suprimirse de cada dominio.

Para problemas grandes de recursos limitados con valores enteros (como son los problemas logísticos que implican el movimiento de miles de personas en cientos de vehículos) no es, por lo general, posible representar el dominio de cada variable como un conjunto grande de enteros y gradualmente reducir ese conjunto por los métodos de comprobación de la consistencia. En cambio, los dominios se representan por límites superiores e inferiores y son manejados por la propagación de límites. Por ejemplo, supongamos que hay dos vuelos, 271 y 272, para los cuales los aviones tienen capacidades de 156 y 385, respectivamente. Los dominios iniciales para el número de pasajeros sobre cada vuelo son

$$\text{Vuelo271} \in [0,165] \quad \text{y} \quad \text{Vuelo272} \in [0,385]$$

Supongamos ahora que tenemos la restricción adicional que los dos vuelos juntos deben llevar a las 420 personas: $\text{Vuelo271} + \text{Vuelo272} \in [420,420]$. Propagando los límites de las restricciones, reducimos los dominios a

$$\text{Vuelo271} \in [35,165] \quad \text{y} \quad \text{Vuelo272} \in [255,385]$$

Decimos que un PSR es consistente-acotado si para cada variable X , y tanto para los valores de las cotas inferior y superior de X , existe algún valor de Y que satisface la restricción entre X e Y , para cada variable Y . Esta clase de **propagación de límites** se utiliza, ampliamente, en problemas restringidos prácticos.

Vuelta atrás inteligente: mirando hacia atrás

El algoritmo de BÚSQUEDA-CON-VUELTA-ATRÁS de la Figura 5.3 tiene una política muy simple para saber qué hacer cuando falla una rama de búsqueda: hacia atrás hasta la variable anterior e intentar un valor diferente para ella. Se llama **vuelta atrás cronológica**, porque se visita de nuevo el punto de decisión *más reciente*. En esta subsección, veremos que hay muchos caminos mejores.

Veamos lo que ocurre cuando aplicamos la vuelta atrás simple de la Figura 5.1 con una variable fija que ordena Q, NGS, V, T, AS, AO, TN . Supongamos que hemos generado la asignación parcial $\{Q = \text{rojo}, NGS = \text{verde}, V = \text{azul}, T = \text{rojo}\}$. Cuando intentamos la siguiente variable, AS , vemos que cada valor viola una restricción. ¡Volvemos hacia atrás hasta T e intentamos un nuevo color para Tasmania! Obviamente esto es inútil (el nuevo color de Tasmania no puede resolver el problema con Australia del Sur).

CONJUNTO
CONFLICTO

SALTO-ATRÁS

SALTO-ATRÁS
DIRIGIDO POR
CONFLICTO

Una aproximación más inteligente a la vuelta atrás es ir hacia atrás hasta el conjunto de variables que *causaron el fracaso*. A este conjunto se le llama **conjunto conflicto**; aquí, el conjunto conflicto para AS es $\{Q, NGS, V\}$. En general, el conjunto conflicto para la variable X es el conjunto de variables previamente asignadas que están relacionadas con X por las restricciones. El método **salto-atrás** retrocede a la variable *más reciente* en el conjunto conflicto; en este caso, el salto-atrás debería saltar sobre Tasmania e intentar un nuevo valor para V . Esto se implementa fácilmente modificando la BÚSQUEDA-CON-VUELTA-ATRÁS de modo que acumule el conjunto conflicto mientras comprueba un valor legal para asignar. Si no se encuentra un valor legal, debería devolver el elemento más reciente del conjunto conflicto con el indicador de fracaso.

El lector habrá notado que la comprobación hacia delante puede suministrar el conjunto conflicto sin trabajo suplementario: siempre que la comprobación hacia delante, basada en una asignación a X , suprima un valor del dominio de Y , deberíamos añadir X al conjunto conflicto de Y . También, siempre que se suprima el último valor del dominio de Y , las variables en el conjunto conflicto de Y se añaden al conjunto conflicto de X . Entonces, cuando llegamos a Y , sabemos inmediatamente dónde volver atrás si es necesario.

El lector habrá notado algo raro: el salto-atrás ocurre cuando cada valor de un dominio está en conflicto con la asignación actual; ¡pero la comprobación hacia delante descubre este acontecimiento y previene la búsqueda de alcanzar alguna vez tal nodo! De hecho, puede demostrarse que *cada rama podada por el salto-atrás también se poda por la comprobación hacia delante*. De ahí, el salto-atrás simple es redundante con una búsqueda de comprobación hacia delante o, en efecto, en una búsqueda que usa la comprobación de consistencia fuerte, como MCA.

A pesar de las observaciones del párrafo anterior, la idea que hay detrás del salto-atrás sigue siendo buena: retroceder basándose en los motivos de fracaso. El salto-atrás se da cuenta del fracaso cuando el dominio de una variable se hace vacío, pero en muchos casos una rama es condenada mucho antes de que esto ocurra. Considere otra vez la asignación parcial $\{AO = \text{rojo}, NGS = \text{rojo}\}$ (que, de nuestra discusión anterior, es inconsistente). Supongamos que intentamos $T = \text{rojo}$ después y luego asignamos a TN, Q, V, AS . Sabemos que no se puede hacer ninguna asignación para estas cuatro últimas variables, ya que finalmente nos quedamos sin valores para intentar en TN . Ahora, la pregunta es, ¿a dónde regresar? El salto-atrás no puede trabajar, porque TN tiene realmente valores consistentes con las variables precedentes adjudicadas (TN no tiene un conjunto conflicto completo de variables precedentes que hicieron que fallara). Sabemos, sin embargo, que las cuatro variables TN, Q, V , y AS , juntas, dan fallo debido a un conjunto de variables precedentes, que directamente entran en conflicto con las cuatro. Esto conduce a una noción más profunda del conjunto conflicto para una variable como TN : es ese conjunto de variables precedentes el que causa que TN , junto con cualquier variable siguiente, no tenga ninguna solución consistente. En este caso, el conjunto es AO y NGS , así que el algoritmo debería regresar a NGS y saltarse Tasmania. Al algoritmo de salto-atrás que utiliza los conjuntos conflicto definidos de esta manera se le llama **salto-atrás dirigido-por-conflicto**.

Debemos explicar ahora cómo calculamos estos nuevos conjuntos conflictos. El método, de hecho, es muy simple. El fracaso «terminal» de una rama de búsqueda siempre

ocurre porque el dominio de una variable se hace vacío; esa variable tiene un conjunto conflicto estándar. En nuestro ejemplo, AS falla, y su conjunto conflicto es $\{AO, TN, Q\}$. Saltamos atrás a Q , y Q absorbe el conjunto conflicto de AS (menos Q , desde luego) en su propio conjunto conflicto directo, que es $\{TN, NGS\}$; el nuevo conjunto conflicto es $\{AO, TN, NGS\}$. Es decir no hay ninguna solución de Q hacia delante, dada la asignación precedente a $\{AO, TN, NGS\}$. Por lo tanto, volvemos atrás a NT , el más reciente de éstos. TN absorbe $\{AO, TN, NGS\} - \{TN\}$ en su propio conjunto conflicto directo $\{AO\}$, dando $\{AO, NGS\}$ (como en los párrafos anteriores). Ahora el algoritmo de salto atrás a NGS , como era de esperar. Resumiendo: sea X_j la variable actual, y sea $conf(X_j)$ su conjunto conflicto. Si para todo valor posible para X_j falla, saltamos atrás a la variable más reciente X_i , en $conf(X_i)$, y el conjunto

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$

APRENDER LA
RESTRICCIÓN

Salto-atrás dirigido-por-conflicto va hacia atrás al punto derecho en el árbol de búsqueda, pero no nos impide cometer los mismos errores en otra rama del árbol. **Aprender la restricción** realmente modifica el PSR añadiendo una nueva restricción inducida por estos conflictos.

5.3 Búsqueda local para problemas de satisfacción de restricciones

Los algoritmos de búsqueda local (véase la Sección 4.3) resultan ser muy eficaces en la resolución de muchos PSRs. Ellos utilizan una formulación de estados completa: el estado inicial asigna un valor a cada variable, y la función sucesor, por lo general, trabaja cambiando el valor de una variable a la vez. Por ejemplo, en el problema de las 8-reinas, el estado inicial podría ser una configuración arbitraria de ocho reinas en ocho columnas, y la función sucesor escoge a una reina y piensa en moverla a otra parte en su columna. Otra posibilidad sería comenzar con ocho reinas, una por columna, en una permutación de las ocho filas, y generaría a un sucesor tomando dos reinas que intercambian sus filas². Hemos visto ya, realmente, un ejemplo de búsqueda local para resolver un PSR: la aplicación de las ascensiones de colinas al problema de n-reinas (página 126). Otra es la aplicación de SAT-CAMINAR (página 251) para resolver problemas de satisfacción, un caso especial de PSRs.

MÍNIMO CONFLICTO

En la elección de un nuevo valor para una variable, la heurística más obvia debe seleccionar el valor que cause el número mínimo de conflictos con otras variables (heurística de **mínimos-conflictos**). La Figura 5.8 muestra el algoritmo y en la Figura 5.9 se muestra su aplicación a un problema de 8-reinas, cuantificada en la Figura 5.5.

Los mínimos-conflictos son sorprendentemente eficaces para muchos PSRs, en particular, cuando se ha dado un estado inicial razonable. En la última columna de la Figu-

² La búsqueda local puede extenderse fácilmente a PSRs con funciones objetivo. En este caso, todas las técnicas de ascensiones de colinas y temple simulado pueden aplicarse para optimizar la función objetivo.

función MÍNIMOS-CONFLICTOS(psr, max_pasos) devuelve una solución o fallo
variables de entrada: psr , un problema de satisfacción de restricciones
 max_pasos , número de pasos permitidos antes de abandonar

```

actual ← una asignación completa inicial para  $psr$ 
para i = 1 hasta  $max\_pasos$  hacer
    si  $actual$  es una solución para  $psr$  entonces devolver  $actual$ 
    var ← escoger aleatoriamente una variable conflictiva de  $VARIABLES[psr]$ 
    valor ← el valor  $v$  para  $var$  que minimiza  $CONFLICTOS(var, v, actual, psr)$ 
    conjunto  $var$  = valor en  $actual$ 
    devolver  $fallo$ 
```

Figura 5.8 El algoritmo de MÍNIMOS-CONFLICTOS para resolver PSRs por búsqueda local. El estado inicial puede elegirse al azar o por un proceso de asignación voraz que elige un valor de mínimo conflicto para cada variable a cambiar. La función CONFLICTOS cuenta el número de restricciones violadas por un valor particular, considerando el resto de la asignación actual.

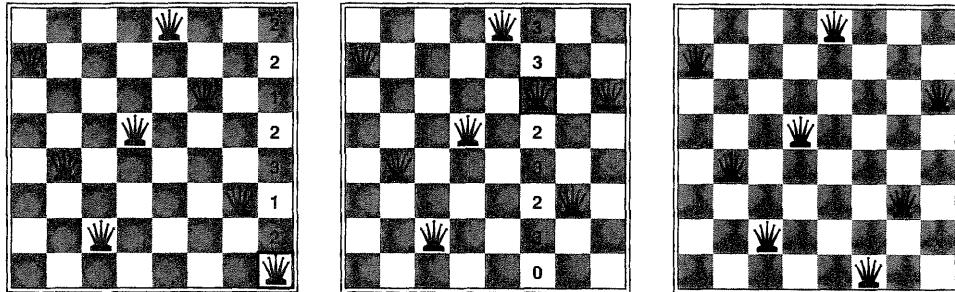


Figura 5.9 Una solución de dos pasos para un problema de 8-reinas usando mínimos-conflictos. En cada etapa, se elige una reina para la reasignación en su columna. El número de conflictos (en este caso, el número de reinas atacadas) se muestra en cada cuadrado. El algoritmo mueve a la reina al cuadrado de mínimo conflicto, deshaciendo los empates de manera aleatoria.

ra 5.5 se muestra su funcionamiento. Extraordinariamente, sobre el problema de las n -reinas, si no cuenta la colocación inicial de las reinas, el tiempo de ejecución de mínimos-conflictos es más o menos *independiente del tamaño del problema*. Resuelve hasta el problema de un millón de reinas en un promedio de 50 pasos (después de la asignación inicial). Esta observación notable fue el estímulo que condujo a gran parte de la investigación en los años 90 sobre la búsqueda local y la diferencia entre problemas fáciles y difíciles, los cuales veremos en el Capítulo 7. Hablando de forma aproximada, las n -reinas son fáciles para la búsqueda local porque las soluciones están densamente distribuidas en todas las partes del espacio de estados. Los mínimos-conflictos también trabajan bien para problemas difíciles. Por ejemplo, se han utilizado para programar las observaciones del Telescopio Hubble, reduciendo el tiempo utilizado para programar una semana de observaciones, de tres semanas (!) a alrededor de 10 minutos.

Otra ventaja de la búsqueda local consiste en que puede usarse en un ajuste *online* cuando el problema cambia. Esto es particularmente importante en la programación de problemas. El programa de vuelos de una semana puede implicar miles de vuelos y decenas de miles de asignaciones de personal, pero el mal tiempo en un aeropuerto puede convertir a la programación en no factible. Nos gustaría reparar la programación con un número mínimo de cambios. Esto puede hacerse fácilmente con un algoritmo de búsqueda local comenzando desde el programa actual. Una búsqueda con vuelta atrás con un nuevo conjunto de restricciones, por lo general, requiere mucho más tiempo y podría encontrar una solución, desde el programa actual, con muchos cambios.

5.4 La estructura de los problemas

En esta sección, examinamos las formas por las cuales la estructura del problema, representada por el grafo de restricciones, puede utilizarse para encontrar soluciones de forma rápida. La mayor parte de estas aproximaciones son muy generales y aplicables a otros problemas, por ejemplo, razonamiento probabilístico. Después de todo, la única forma que podemos esperar, posiblemente, que pueda tratar con el mundo real es la descomposición en muchos subproblemas. Viendo de nuevo la Figura 5.1(b), con miras a identificar la estructura del problema, se destaca un hecho: Tasmania no está relacionada con el continente³. Intuitivamente, es obvio que colorear Tasmania y colorear el continente son **subproblemas independientes** (cualquier solución para el continente combinado con cualquier solución para Tasmania produce una solución para el mapa entero). La independencia puede averiguarla simplemente buscando **componentes conectados** del grafo de restricciones. Cada componente corresponde a un subproblema PSR_i . Si la asignación S_i es una solución de PSR_i , entonces $\bigcup_i S_i$ es una solución de $\bigcup_i PSR_i$. ¿Por qué es tan importante? Consideremos lo siguiente: suponga que cada PSR_i tiene c variables del total de n variables, donde c es una constante. Entonces hay n/c subproblemas, cada uno de los cuales trae consigo como mucho d^c de trabajo para resolverlo. De ahí, que el trabajo total es $O(d^c n/c)$, que es lineal en n ; sin la descomposición, el trabajo total es $O(d^n)$, que es exponencial en n . Seamos más concretos: la división de un PSR booleano con $n = 80$ en cuatro subproblemas con $c = 20$ reduce el tiempo de resolución, en el caso peor, de la vida del universo hasta menos de un segundo.

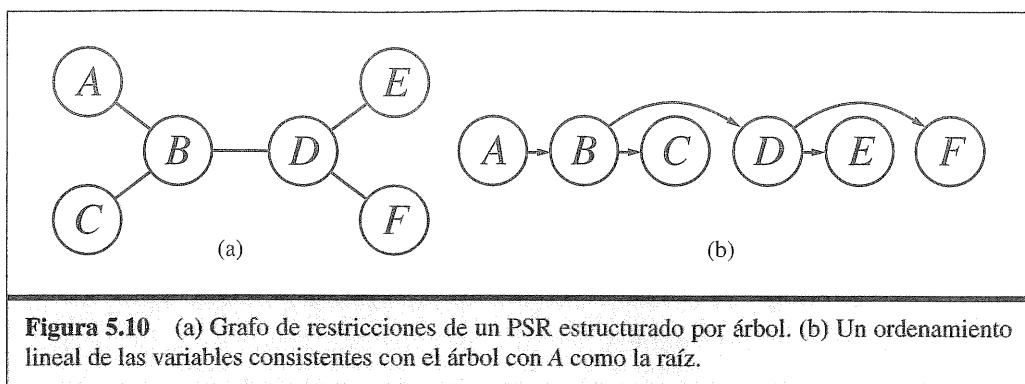
Los subproblemas completamente independientes son deliciosos, pero raros. En la mayoría de los casos, los subproblemas de un PSR están relacionados. El caso más simple es cuando el grafo de restricciones forma un **árbol**: cualquiera dos variables están relacionadas por, a lo sumo, un camino. La Figura 5.10(a) muestra un ejemplo esquemático⁴.

³ Un cartógrafo cuidadoso o un tasmanio patriótico podría objetar que Tasmania no debiera ser coloreada lo mismo que su vecino de continente más cercano, lo que evitaría la impresión de que pueda ser parte de aquel estado.

⁴ Tristemente, muy pocas regiones del mundo, con la excepción posible de Sulawesi, tienen mapas estructurados por árboles.

SUBPROBLEMAS
INDEPENDIENTES

COMPONENTES
CONECTADOS



Mostraremos que cualquier *PSR* estructurado por árbol puede resolverse en tiempo lineal en el número de variables. El algoritmo tiene los siguientes pasos:



1. Elija cualquier variable como la raíz del árbol, y ordene las variables desde la raíz a las hojas de tal modo que el padre de cada nodo en el árbol lo precede en el ordenamiento. (Véase la Figura 5.10(b).) Etiquetar las variables X_1, \dots, X_n en orden. Ahora, cada variable excepto la raíz tiene exactamente una variable padre.
 2. Para j desde n hasta 2, aplicar la consistencia de arco al arco (X_i, X_j) , donde X_i es el parente de X_j , quitando los valores del $\text{DOMINIO}[X_i]$ que sea necesario.
 3. Para j desde 1 a n , asigne cualquier valor para X_j consistente con el valor asignado para X_i , donde X_i es el parente de X_j .

Hay dos puntos claves a destacar. Primero, después del paso 2 el PSR es directamente arco-consistente, entonces la asignación de valores en el paso 3 no requiere ninguna vuelta atrás. (Véase la discusión de k -consistencia de la página 167.) Segundo, aplicando la comprobación de consistencia de arco en orden inverso en el paso 2, el algoritmo asegura que cualquier valor suprimido no puede poner en peligro la consistencia de arcos que ya han sido tratados. El algoritmo completo se ejecuta en tiempo $O(nd^2)$.

Ahora que tenemos un algoritmo eficiente para árboles, podemos considerar si los grafos restricción más generales pueden *reducirse* a árboles de alguna manera. Hay dos modos primarios de hacer esto, uno basado en quitar nodos y uno basado en nodos que sufren colisiones.

La primera aproximación implica valores de asignación a algunas variables de modo que las variables restantes formen un árbol. Consideremos el grafo restricción para Australia, mostrado otra vez en la Figura 5.11(a). Si pudieramos suprimir Australia del Sur, el grafo se haría un árbol, como en (b). Por suerte, podemos hacer esto (en el grafo, no en el continente) fijando un valor para *AS* y suprimiendo de los dominios de las otras variables cualquier valor que sea inconsistente con el valor elegido para *AS*.

Ahora, cualquier solución para el PSR después de que AS y sus restricciones se quiten, será consistente con el valor elegido para AS . (Para los PSRs binarios, la situación es más complicada con restricciones de orden alto.) Por lo tanto, podemos resolver el árbol restante con el algoritmo anterior y así resolver el problema entero. Desde luego, en el caso general (a diferencia de colorear el mapa) el valor elegido para AS podría ser

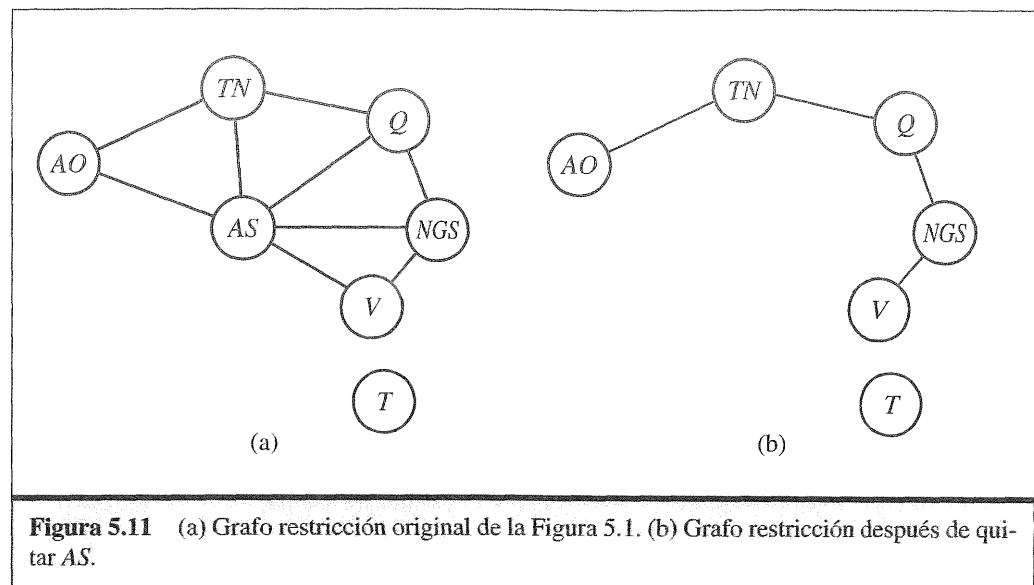


Figura 5.11 (a) Grafo restricción original de la Figura 5.1. (b) Grafo restricción después de quitar AS .

el incorrecto, entonces tendríamos que intentarlo con cada uno de ellos. El algoritmo general es como sigue:

1. Elegir un subconjunto S de $\text{VARIABLES}[psr]$ tal que el grafo de restricciones se convierta en un árbol después del quitar S . Llamamos a S un **ciclo de corte**.
2. Para cada asignación posible a las variables en S que satisface todas las restricciones sobre S ,
 - (a) quitar de los dominios de las variables restantes cualquier valor que sea inconsistente con la asignación para S , y
 - (b) si el PSR restante tiene una solución, devolverla junto con la asignación para S .

Si el ciclo de corte tiene tamaño c , entonces el tiempo de ejecución total es $O(d^c \cdot (n - c)d^2)$. Si el grafo es «casi un árbol» entonces c será pequeño y los ahorros sobre la vuelta atrás serán enormes. En el caso peor, sin embargo, c puede ser tan grande como $(n - 2)$. Encontrar el ciclo de corte *más pequeño* es NP-duro, pero se conocen varios algoritmos aproximados eficientes para esta tarea. A la aproximación algorítmica general se le llama **acondicionamiento del corte** que veremos de nuevo en el Capítulo 14, donde se usará para el razonamiento sobre probabilidades.

La segunda aproximación está basada en la construcción de una **descomposición en árbol** del grafo restricción en un conjunto de subproblemas relacionados. Cada subproblema se resuelve independientemente, y las soluciones que resultan son entonces combinadas. Como la mayoría de los algoritmos divide-y-vencerás, trabajan bien si ninguno de los subproblemas es demasiado grande. La Figura 5.12 muestra una descomposición de árbol del problema que colorea un mapa en cinco subproblemas. Una descomposición de árbol debe satisfacer las tres exigencias siguientes:

- Cada variable en el problema original aparece en al menos uno de los subproblemas.

- Si dos variables están relacionadas por una restricción en el problema original, deben aparecer juntas (junto con la restricción) en al menos uno de los subproblemas.
- Si una variable aparece en dos subproblemas en el árbol, debe aparecer en cada subproblema a lo largo del camino que une a esos subproblemas.

Las dos primeras condiciones aseguran que todas las variables y las restricciones están representadas en la descomposición. La tercera condición parece bastante técnica, pero simplemente refleja la restricción de que cualquier variable debe tener el mismo valor en cada subproblema en el cual aparece; los enlaces que unen subproblemas en el árbol hacen cumplir esta restricción. Por ejemplo, *AS* aparece en los cuatro subproblemas de la Figura 5.12. Puede verificar de la Figura 5.11 que esta descomposición tiene sentido.

Resolvemos cada subproblema independientemente; si alguno de ellos no tiene ninguna solución, sabemos que el problema entero no tiene ninguna solución. Si podemos resolver todos los subproblemas, entonces intentamos construir una solución global como sigue. Primero, vemos a cada subproblema como «una megavariable» cuyo dominio es el conjunto de todas las soluciones para el subproblema. Por ejemplo, los subproblemas más a la izquierda en la Figura 5.12 forman un problema que colorea el mapa con tres variables y de ahí que tienen seis soluciones (una es $\{AO = \text{rojo}, AS = \text{azul}, TN = \text{verde}\}$). Entonces, resolvemos las restricciones que unen los subproblemas utilizando el algoritmo eficiente para árboles. Las restricciones entre subproblemas simplemente insisten en que las soluciones del subproblema deben estar de acuerdo con sus variables compartidas. Por ejemplo, considerando la solución $\{AO = \text{rojo}, AS = \text{azul}, TN = \text{verde}\}$ para el primer subproblema, la única solución consistente para el siguiente subproblema es $\{AS = \text{azul}, TN = \text{verde}, Q = \text{rojo}\}$.

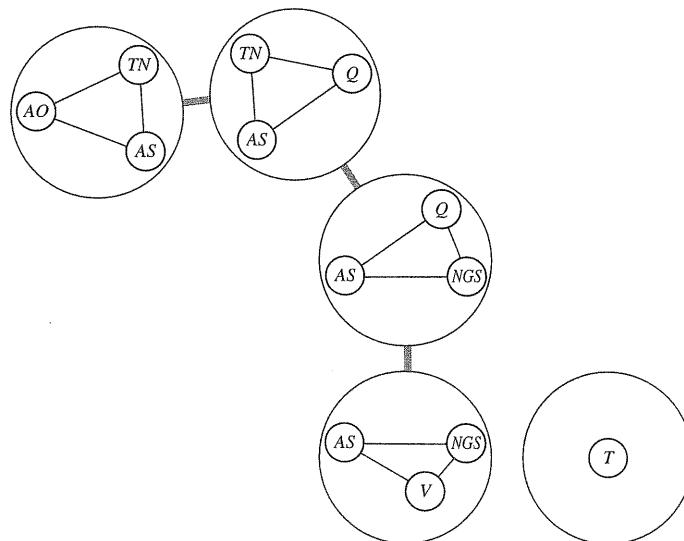


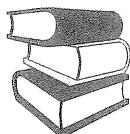
Figura 5.12 Una descomposición en árbol del grafo restricción de la Figura 5.11(a)



El grafo de restricciones admite muchas descomposiciones en árbol; el objetivo en la elección de una descomposición, es hacer los subproblemas tan pequeños como sea posible. La **anchura del árbol** de una descomposición en árbol de un grafo es menor que el tamaño del subproblema más grande; la anchura de árbol del grafo en sí mismo está definida por la anchura de árbol mínimo entre todas sus descomposiciones en árbol. Si un grafo tiene la anchura de árbol w , y nos dan la descomposición en árbol correspondiente, entonces el problema puede resolverse en $O(nd^{w+1})$ veces. De ahí que, los PSRs *con grafos de restricciones de anchura de árbol acotada son resolvibles en tiempo polinomial*. Lamentablemente, encontrar la descomposición con la anchura del árbol mínima es NP-duro, pero hay métodos heurísticos que trabajan bien en la práctica.

5.5 Resumen

- Los **problemas de satisfacción de restricciones** (o PSRs) consisten en variables con restricciones sobre ellas. Muchos problemas importantes del mundo real pueden describirse como PSRs. La estructura de un PSR puede representarse por su grafo de restricciones.
- La **búsqueda con vuelta atrás**, una forma de búsqueda primero en profundidad, es comúnmente utilizada para resolver PSRs.
- Las heurísticas de **mínimos valores restantes** y **mínimo grado restante** son métodos, independientes del dominio, para decidir qué variable elegir en una búsqueda con vuelta atrás. La heurística **valor menos restringido** ayuda en la ordenación de los valores de las variables.
- Propagando las consecuencias de las asignaciones parciales que se construyen, el algoritmo con vuelta atrás puede reducir enormemente el factor de ramificación del problema. La **comprobación hacia delante** es el método más simple para hacerlo. La imposición de la **consistencia del arco** es una técnica más poderosa, pero puede ser más costosa de ejecutar.
- La vuelta atrás ocurre cuando no se puede encontrar ninguna asignación legal para una variable. El **salto atrás dirigido por conflictos** vuelve atrás directamente a la fuente del problema.
- La búsqueda local usando la heurística de **mínimos conflictos** se ha aplicado a los problemas de satisfacción de restricciones con mucho éxito.
- La complejidad de resolver un PSR está fuertemente relacionada con la estructura de su grafo de restricciones. Los problemas estructurados por árbol pueden resolverse en tiempo lineal. El **acondicionamiento del corte** puede reducir un PSR general a uno estructurado por árbol y es muy eficiente si puede encontrarse un corte pequeño. Las técnicas de **descomposición en árbol** transforman el PSR en un árbol de subproblemas y son eficientes si la **anchura de árbol** del grafo de restricciones es pequeña.

EQUACIONES
DIOPHANTINECOLOREO DE
UN GRAFO

NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

El primer trabajo relacionado con la satisfacción de restricciones trató, en gran parte, restricciones numéricas. Las restricciones ecuacionales con dominios de enteros fueron estudiadas por el matemático indio Brahmagupta en el siglo VII; a menudo se les llaman **ecuaciones Diophantine**, después del matemático griego Diophantus (200-284), quien realmente consideró el dominio de racionales positivos. Los métodos sistemáticos para resolver ecuaciones lineales por eliminación de variables fueron estudiados por Gauss (1829); la solución de restricciones lineales en desigualdad se retoman con Fourier (1827).

Los problemas de satisfacción de restricciones con dominios finitos también tienen una larga historia. Por ejemplo, el **coloreo de un grafo** (el coloreo de un mapa es un caso especial) es un viejo problema en matemáticas. Según Biggs *et al.* (1986), la conjectura de cuatro colores (que cada grafo plano puede colorearse con cuatro o menos colores) la hizo primero, en 1852, Fancis Guthrie, un estudiante de Morgan. Esta solución resistió, a pesar de varias reclamaciones publicadas en contra, hasta que Appel y Haken (1977) realizaron una demostración, con la ayuda de un computador.

Clases específicas de problemas de satisfacción de restricciones aparecen a través de la historia de la informática. Uno de los primeros ejemplos más influyentes fue el sistema de SKETCHPAD (Sutherland, 1963), quien resolvió restricciones geométricas en diagramas y fue el precursor de los programas modernos de dibujo y las herramientas CAD. La identificación de PSRs como una clase *general* se debe a Ugo Montanari (1974). La reducción de PSRs de orden alto a PSRs puramente binarios con variables auxiliares (*véase* el Ejercicio 5.11) se debe originalmente al lógico Charles Sanders Peirce del siglo XIX. Fue introducido en la literatura de PSR por Dechter (1990b) y fue elaborado por Bacchus y van Beek (1998). Los PSRs con preferencias entre las soluciones son estudiados ampliamente en la literatura de optimización; *véase* Bistarelli *et al.* (1997) para una generalización del marco de trabajo de PSRs que tienen en cuenta las preferencias. El algoritmo de eliminación de cubo (Dechter, 1999) puede también aplicarse a problemas de optimización.

La búsqueda con vuelta atrás para la satisfacción de restricciones se debe a Bitner y Reingold (1975), aunque ellos remonten el algoritmo básico al siglo XIX. Bitner y Reingold también introdujeron la heurística MVR, que llamaron heurística de la *variable más restringida*. Brelaz (1979) usó el grado heurístico como un modo de deshacer el empate después de aplicar la heurística MVR. El algoritmo que resulta, a pesar de su simplicidad, es todavía el mejor método para el *k*-coloreo de grafos arbitrarios. Haralick y Elliot (1980) propusieron la *heurística del valor menos restringido*.

Los métodos de propagación de restricciones se popularizaron con el éxito de Waltz (1975) sobre problemas poliedríticos de etiquetado para la visión por computador. Waltz mostró que, en muchos problemas, la propagación completa elimina la necesidad del retroceso. Montanari (1974) introdujo la noción de redes de restricciones y la propagación por la consistencia del camino. Alan Mackworth (1977) propuso el algoritmo AC-3 para hacer cumplir la consistencia del arco así como la idea general de combinar la vuelta atrás con algún grado de imposición de la consistencia. AC-4, un algoritmo de consistencia

del arco más eficiente, fue desarrollado por Mohr y Henderson (1986). Inmediatamente después de que apareciera el trabajo de Mackworth, los investigadores comenzaron a experimentar sobre la compensación entre el coste al imponer la consistencia y las ventajas en términos de reducción de la búsqueda. Haralick y Elliot (1980) favorecieron el algoritmo de comprobación hacia delante mínima descrito por McGregor (1979), mientras que Gaschnig (1979) sugirió la comprobación completa de la consistencia del arco después de cada asignación de una variable (un algoritmo posterior llamado MAC de Sabin y Freuder (1994)). El trabajo último proporciona pruebas convincentes de que, sobre PSRs más difíciles, la comprobación completa de la consistencia de arco merece la pena. Freuder (1978, 1982) investigó la noción de k -consistencia y su relación con la complejidad de resolver PSRs. Apt (1999) describe un marco de trabajo algorítmico genérico dentro del cual se pueden analizar los algoritmos de propagación de consistencia.

Los métodos especiales para manejar restricciones de orden alto se han desarrollado principalmente dentro del contexto de la **programación lógica de restricciones**. Marriot y Stuckey (1998) proporcionan una excelente cobertura de la investigación en este área. La restricción *Todasdif* fue estudiada por Regin (1994). Las restricciones acotadas fueron incorporadas en la programación lógica de restricciones por Van Hentenryck *et al.* (1998).

Los métodos básicos de salto atrás se deben a John Gaschnig (1977, 1979). Kondrak y van Beek (1997) mostraron que este algoritmo es esencialmente subsumido por la comprobación hacia delante. El salto atrás dirigido por conflictos fue ideado por Prosser (1993). La forma más general y poderosa de la vuelta hacia atrás inteligente fue realmente desarrollada muy pronto por Stallman y Sussman (1997). Su técnica del **retroceso dirigido por dependencias** condujo al desarrollo de **sistemas de mantenimiento de la verdad** (Doyle, 1979), de los que hablaremos en la Sección 10.8. La conexión entre las dos áreas la analiza Kleer (1989).

El trabajo de Stallman y Sussman también introdujo la idea de la **grabación de restricciones**, en la cual los resultados parciales obtenidos por la búsqueda pueden salvarse y ser reutilizados más tarde. La idea fue introducida formalmente en la búsqueda con vuelta atrás de Dechter (1990a). **Marcar hacia atrás** (Gaschnig, 1979) es un método particularmente simple en el cual se salvan las asignaciones por parejas consistentes e inconsistentes y usadas para evitar comprobar de nuevo las restricciones. Marcar hacia atrás puede combinarse con salto atrás dirigido por conflictos; Kondrad y van Beek (1997) presentan un algoritmo híbrido que probablemente subsume cualquier método. El método del **retroceso dinámico** (Ginsberg, 1993) retiene asignaciones parciales satisfactorias de subconjuntos posteriores de variables cuando volvemos atrás sobre una opción anterior que no invalida el éxito posterior.

La búsqueda local en problemas de satisfacción de restricciones se popularizó con el trabajo de Kirkpatrick *et al.* (1983) sobre el **templo simulado** (véase el Capítulo 4), ampliamente utilizado para problemas de programación. El primero que propuso la heurística de mínimo conflicto fue Gu (1989) y desarrollada independientemente por Minton *et al.* (1992). Sosic y Gu (1994) mostraron cómo podría aplicarse para resolver el problema de 3.000.000 de reinas en menos de un minuto. El éxito asombroso de la búsqueda local, usando mínimos conflictos, sobre el problema de las n -reinas condujo a una nueva estimación de la naturaleza y predominio de problemas «fáciles» y «difíciles». Peter Cheeseman *et al.* (1991) exploraron la dificultad de los PSRs generados aleatoriamente

RETROCESO DIRIGIDO POR DEPENDENCIAS

GRABACIÓN DE RESTRICCIONES

MARCAR HACIA ATRÁS

RETROCESO DINÁMICO

y descubrieron que casi todos estos problemas son trivialmente fáciles o no tienen ninguna solución. Sólo si los parámetros del generador del problema se ponen en un cierto rango limitado, dentro del cual aproximadamente la mitad de los problemas son resolubles, encontramos casos de problemas «difíciles». Hablamos de este fenómeno en el Capítulo 7.

El trabajo que relaciona la estructura y la complejidad de los PSRs proviene de Freuder (1985), quien mostró que la búsqueda sobre árboles arco-consistentes trabaja sin ninguna vuelta atrás. Un resultado similar, con extensiones a hipergrafos acíclicos, fue desarrollado en la comunidad de base de datos (Beeri *et al.* 1983). Desde que esos trabajos fueron publicados, hubo mucho progreso en el desarrollo de más resultados generales que relacionan la complejidad de resolver un PSR con la estructura de su grafo de restricciones. La noción de anchura del árbol fue introducida por los teóricos de grafos Robertson y Seymour (1986). Dechter y Pearl (1987, 1989), construyendo sobre el trabajo de Freuder, aplicaron la misma noción (que ellos llamaron la **anchura inducida**) a problemas de satisfacción de restricciones y desarrollaron la aproximación de descomposición en árbol esbozado en la Sección 5.4. Usando este trabajo y sobre resultados de teoría de base de datos, Gottlob *et al.* (1999a, 1999b) desarrollaron una noción, **anchura del hiperárbol**, basada en la caracterización del PSR como un hipergrafo. Además mostraron que cualquier PSR con la anchura de hiperárbol w puede resolverse en tiempo $O(n^{w+1} \log n)$ y que la anchura de hiperárbol subsume todas las medidas de «anchura» (antes definidas) en el sentido de que hay casos donde la anchura de hiperárbol está acotada y las otras medidas no están acotadas.

Hay varias buenas revisiones de técnicas de PSRs, incluyendo la de Kumar (1992), Dechter y Frost (1999), y Bartak (2001); y la enciclopedia de artículos de Dechter (1992) y Mackworth (1992). Pearson y Jeavons (1997) contemplan clases manejables de PSRs, cubriendo tanto métodos de descomposición estructurales como métodos que confían en las propiedades de los dominios o restricciones. Kondrak y van Beek (1997) dan una revisión analítica de algoritmos de búsqueda con vuelta atrás, y Bacchus y van Run (1995) dan una revisión más empírica. Los textos de Tsang (1993) y de Marriott y Stuckey (1998) entran en más profundidad que la realizada en este capítulo. Varias aplicaciones interesantes se describen en la colección editada por Freuder y Mackworth (1994). Los trabajos sobre satisfacción de restricciones aparecen con regularidad en la revista *Artificial Intelligence* y en la revista especializada *Constraints*. La primera conferencia local es la *International Conference on Principles and Practice of Constraint Programming*, a menudo llamada CP.



EJERCICIOS

5.1 Defina con sus propias palabras los términos problema de satisfacción de restricciones, restricción, búsqueda con vuelta atrás, consistencia de arco, salto atrás y mínimos conflictos.

5.2 ¿Cuántas soluciones hay para el problema que colorea el mapa de la Figura 5.1?

5.3 Explique por qué es una buena heurística elegir la variable que está *más* restringida, en lugar del valor que está menos restringido en una búsqueda de PSR.

5.4 Considere el problema de construir (no resolver) crucigramas⁵: prueba de palabras en una rejilla rectangular. La rejilla, dada como parte del problema, especifica qué cuadrados son en blanco y cuáles sombreados. Asuma que se proporciona una lista de palabras (es decir, un diccionario) y que la tarea es llenar los cuadrados en blanco usando cualquier subconjunto de la lista. Formule este problema, con precisión, de dos modos:

- a) Como un problema general de búsqueda. Elija un algoritmo de búsqueda apropiado, y especifique una función heurística, si piensa que es necesaria. ¿Es mejor llenar con una letra o una palabra a la vez?
- b) Como un problema de satisfacción de restricciones ¿deberían las variables ser palabras o letras?

¿Qué formulación piensa que será mejor? ¿Por qué?

5.5 Dé formulaciones precisas para cada uno de los siguientes problemas de satisfacción de restricciones:

- a) **Planificación en el plano rectilíneo:** encuentre lugares no solapados en un rectángulo grande para varios rectángulos más pequeños.
- b) **Programación de clases:** Hay un número fijo de profesores y aulas, una lista de clases, y una lista de huecos posibles para las clases. Cada profesor tiene un conjunto de clases que él o ella pueden enseñar.

5.6 Resuelva a mano el problema criptoaritmético de la Figura 5.2, usando el retroceso, comprobación hacia delante, y las heurísticas MVR y la del valor menos restringido.

PLANIFICACIÓN EN
EL PLANO

PLANIFICACIÓN DE
CLASES



5.7 La Figura 5.5 prueba varios algoritmos sobre el problema de las n -reinas. Intente estos mismos algoritmos sobre problemas de coloreo de mapas generados aleatoriamente como sigue: disperse n puntos sobre el cuadrado unidad; seleccionando un punto X al azar, una X con una línea recta al punto más cercano Y tal que X no está ya relacionado con Y y la línea no cruce ninguna otra línea; repita el paso anterior hasta que no haya más uniones posibles. Construya la tabla de funcionamiento para el n más grande que pueda manejar, usando tanto colores $d = 4$ como $d = 3$. Comente sus resultados.

5.8 Utilice el algoritmo AC-3 para mostrar que la consistencia de arco es capaz de descubrir la inconsistencia de la asignación parcial $\{AO = \text{rojo}, V = \text{azul}\}$ para el problema de la Figura 5.1.

5.9 ¿Cuál es la complejidad, en el caso peor, al ejecutar AC-3 sobre un PSR estructurado por árbol?

5.10 AC-3 vuelve a poner sobre la cola cada arco (X_k, X_i) siempre que *algún* valor sea suprimido del dominio de X_i , aunque cada valor de X_k sea consistente con los valores restantes de X_i . Suponga que, para cada arco (X_k, X_i) , guardamos el número de valores restantes de X_i que son consistentes con cada valor de X_k . Explique cómo actualizar estos números de manera eficiente y muestre que la consistencia de arco puede hacerse en tiempo total $O(n^2d^2)$.

⁵ Ginsberg *et al.* (1990) hablan de varios métodos para construir crucigramas. Littman *et al.* (1999) abordan el problema más difícil: resolverlos.

5.11 Muestre cómo una restricción ternaria simple como « $A + B = C$ » puede transformarse en tres restricciones binarias usando una variable auxiliar. Puede suponer dominios finitos. (*Consejo*: considere una nueva variable que tome valores que son pares de otros valores, y considere que las restricciones como « X es el primer elemento del par Y ».) Después, muestre cómo las restricciones con más de tres variables pueden tratarse de modo similar. Finalmente, muestre cómo las restricciones con una sola variable pueden eliminarse cambiando los dominios de las variables. Esto completa la demostración de que cualquier PSR puede transformarse en un PSR con restricciones binarias.



5.12 Suponga que un grafo tiene un ciclo de corte de no más de k nodos. Describa un algoritmo simple para encontrar un ciclo de corte mínimo cuyo tiempo de ejecución no sea mucho más que $O(n^k)$ para un PSR con n variables. Busque en la literatura métodos para encontrar, aproximadamente, el ciclo de corte mínimo en tiempo polinomial en el tamaño del corte. ¿La existencia de tales algoritmos hacen práctico al método de ciclo de corte?

5.13 Considere el siguiente puzzle lógico: en cinco casas, cada una con un color diferente, viven cinco personas de nacionalidades diferentes, cada una de las cuales prefiere una marca diferente de cigarrillos, una bebida diferente, y un animal doméstico diferente. Considerando los hechos siguientes, la pregunta para contestar es «¿dónde vive Zebra, y en qué casa beben ellos el agua?»

El inglés vive en la casa roja.

El español posee el perro.

El noruego vive en la primera casa a la izquierda.

Los Kools son fumados en la casa amarilla.

El hombre que fuma Chesterfields vive en la casa al lado del hombre con el zorro.

El noruego vive al lado de la casa azul.

El fumador de Winston posee caracoles.

El fumador de Lucky Strike bebe zumo de naranja.

El ucraniano bebe el té.

El japonés fuma los Parlaments.

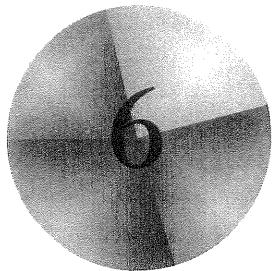
Los Kools son fumados en la casa al lado de la casa donde se guarda el caballo.

El café es bebido en la casa verde.

La casa verde está inmediatamente a la derecha (su derecha) de la casa de color marfil.

La leche es bebida en la casa del medio.

Discuta diferentes representaciones de este problema como un PSR. ¿Por qué preferiría una representación sobre otra?



Búsqueda entre adversarios

Donde examinaremos los problemas que surgen cuando tratamos de planear en un mundo donde otros agentes planean contra nosotros.

6.1 Juegos

En el Capítulo 2 se introdujeron los **entornos multiagente**, en los cuales cualquier agente tendrá que considerar las acciones de otros agentes y cómo afectan a su propio bienestar. La imprevisibilidad de estos otros agentes puede introducir muchas posibles **contingencias** en el proceso de resolución de problemas del agente, como se discutió en el Capítulo 3. En el Capítulo 2 también se introdujo la diferencia entre entornos multiagente **cooperativos** y **competitivos**. Los entornos competitivos, en los cuales los objetivos del agente están en conflicto, dan ocasión a problemas de **búsqueda entre adversarios**, a menudo conocidos como **juegos**.

JUEGOS

JUEGOS DE SUMA CERO

INFORMACIÓN PERFECTA

La **teoría matemática de juegos**, una rama de la Economía, ve a cualquier entorno multiagente como un juego a condición de que el impacto de cada agente sobre los demás sea «significativo», sin tener en cuenta si los agentes son cooperativos o competitivos¹. En IA, «los juegos» son, por lo general, una clase más especializada (que los teóricos de juegos llaman **juegos de suma cero**, de dos jugadores, por turnos, determinista, de **información perfecta**). En nuestra terminología, significan entornos deterministas, totalmente observables en los cuales hay dos agentes cuyas acciones deben alternar y en los que los valores utilidad, al final de juego, son siempre iguales y opuestos. Por ejemplo, si un jugador gana un juego de ajedrez (+1), el otro jugador necesita

¹ Los entornos con muchos agentes se ven mejor como entornos **económicos** más que como juegos.

riamente pierde (-1). Esta oposición entre las funciones de utilidad de los agentes hace la situación entre adversarios. Consideraremos brevemente en este capítulo juegos multi-jugador, juegos de suma no cero, y juegos estocásticos, pero aplazaremos hasta el Capítulo 17 la discusión de la teoría de juegos apropiada.

Los juegos han ocupado las facultades intelectuales de la gente (a veces a un grado alarmante) mientras ha existido la civilización. Para los investigadores de IA, la naturaleza abstracta de los juegos los hacen un tema atractivo a estudiar. El estado de un juego es fácil de representar, y los agentes están restringidos, por lo general, a un pequeño número de acciones cuyos resultados están definidos por reglas precisas. Los juegos físicos, como croquet y hockey sobre hielo, tienen descripciones mucho más complicadas, una variedad mucho más grande de acciones posibles, y reglas bastante imprecisas que definen la legalidad de las acciones. A excepción del fútbol de robots, estos juegos físicos no han tenido mucho interés en la comunidad de IA.

El jugar a juegos fue una de las primeras tareas emprendidas en IA. Hacia 1950, casi tan pronto como los computadores se hicieron programables, el ajedrez fue abordado por Konrad Zuse (el inventor del primer computador programable y del primer lenguaje de programación), por Claude Shannon (el inventor de la teoría de información), por Norbert Wiener (el creador de la teoría de control moderna), y por Alan Turing. Desde entonces, hubo un progreso continuo en el nivel de juego, hasta tal punto que las máquinas han superado a la gente en las damas y en Oteló, han derrotado a campeones humanos (aunque no siempre) en ajedrez y *backgammon*, y son competitivos en muchos otros juegos. La excepción principal es Go, en el que los computadores funcionan a nivel aficionado.

Los juegos, a diferencia de la mayor parte de los problemas de juguete estudiados en el Capítulo 3, son interesantes *porque* son demasiado difíciles para resolverlos. Por ejemplo, el ajedrez tiene un factor de ramificación promedio de aproximadamente 35, y los juegos a menudo van a 50 movimientos por cada jugador, entonces el árbol de búsqueda tiene aproximadamente 35^{100} o 10^{154} nodos (aunque el grafo de búsqueda tenga «sólo» aproximadamente 10^{40} nodos distintos). Por lo tanto, los juegos, como el mundo real, requieren la capacidad de tomar *alguna* decisión cuando es infactible calcular la decisión *óptima*. Los juegos también castigan la ineficiencia con severidad. Mientras que una implementación de la búsqueda A* que sea medio eficiente costará simplemente dos veces más para ejecutarse por completo, un programa de ajedrez que sea medio eficiente en la utilización de su tiempo disponible, probablemente tendrá que descartarse si no intervienen otros factores. La investigación en juegos ha generado, por lo tanto, varias ideas interesantes sobre cómo hacer uso, lo mejor posible, del tiempo.

Comenzamos con una definición del movimiento óptimo y un algoritmo para encontrarlo. Veremos técnicas para elegir un movimiento bueno cuando el tiempo es limitado. La **poda** nos permite ignorar partes del árbol de búsqueda que no marcan ninguna diferencia para obtener la opción final, y las **funciones de evaluación** heurísticas nos permiten aproximar la utilidad verdadera de un estado sin hacer una búsqueda completa. La Sección 6.5 habla de juegos como el *backgammon* que incluyen un elemento de posibilidad; también hablamos del *bridge*, que incluye elementos de **información imperfecta** al no ser visibles todas las cartas a cada jugador. Finalmente, veremos cómo se desenvuelven los programas de juegos contra la oposición humana y las direcciones para el desarrollo futuro.

6.2 Decisiones óptimas en juegos

Consideraremos juegos con dos jugadores, que llamaremos MAX y MIN por motivos que pronto se harán evidentes. MAX mueve primero, y luego mueven por turno hasta que el juego se termina. Al final de juego, se conceden puntos al jugador ganador y penalizaciones al perdedor. Un juego puede definirse formalmente como una clase de problemas de búsqueda con los componentes siguientes:

- **El estado inicial**, que incluye la posición del tablero e identifica al jugador que mueve.
- **Una función sucesor**, que devuelve una lista de pares (*movimiento, estado*), indicando un movimiento legal y el estado que resulta.
- **Un test terminal**, que determina cuándo se termina el juego. A los estados donde el juego se ha terminado se les llaman **estados terminales**.
- **Una función utilidad** (también llamada función objetivo o función de rentabilidad), que da un valor numérico a los estados terminales. En el ajedrez, el resultado es un triunfo, pérdida, o empate, con valores +1, -1 o 0. Algunos juegos tienen una variedad más amplia de resultados posibles: las rentabilidades en el *backgammon* se extienden desde +192 a -192. Este capítulo trata principalmente juegos de suma cero, aunque mencionemos brevemente juegos con «suma no cero».

TEST TERMINAL

ÁRBOL DE JUEGOS

El estado inicial y los movimientos legales para cada lado definen el **árbol de juegos**. La Figura 6.1 muestra la parte del árbol de juegos para el tic-tac-toe (tres en raya). Desde el estado inicial, MAX tiene nueve movimientos posibles. El juego alterna entre la colocación de una X para MAX y la colocación de un O para MIN, hasta que alcancemos nodos hoja correspondientes a estados terminales, de modo que un jugador tenga tres en raya o todos los cuadrados estén llenos. El número sobre cada nodo hoja indica el valor de utilidad del estado terminal desde el punto de vista de MAX; se supone que los valores altos son buenos para MAX y malos para MIN (por eso los nombres de los jugadores). Este trabajo de MAX al usar el árbol de búsqueda (en particular la utilidad de estados terminales) determina el mejor movimiento.

Estrategias óptimas

En un problema de búsqueda normal, la solución óptima sería una secuencia de movimientos que conducen a un estado objetivo (un estado terminal que es ganador). En un juego, por otra parte, MIN tiene algo que decir sobre ello. MAX por lo tanto debe encontrar una **estrategia contingente**, que especifica el movimiento de MAX en el estado inicial, después los movimientos de MAX en los estados que resultan de cada respuesta posible de MIN, después los movimientos de MAX en los estados que resultan de cada respuesta posible de MIN de los *anteriores* movimientos, etcétera. Hablando de forma aproximada, una estrategia óptima conduce a resultados al menos tan buenos como cualquier otra estrategia cuando uno juega con un oponente infalible. Comenzaremos mostrando cómo encontrar esta estrategia óptima, aunque será infactible, para MAX, al calcularla en juegos más complejos que tic-tac-toe.

ESTRATEGIA

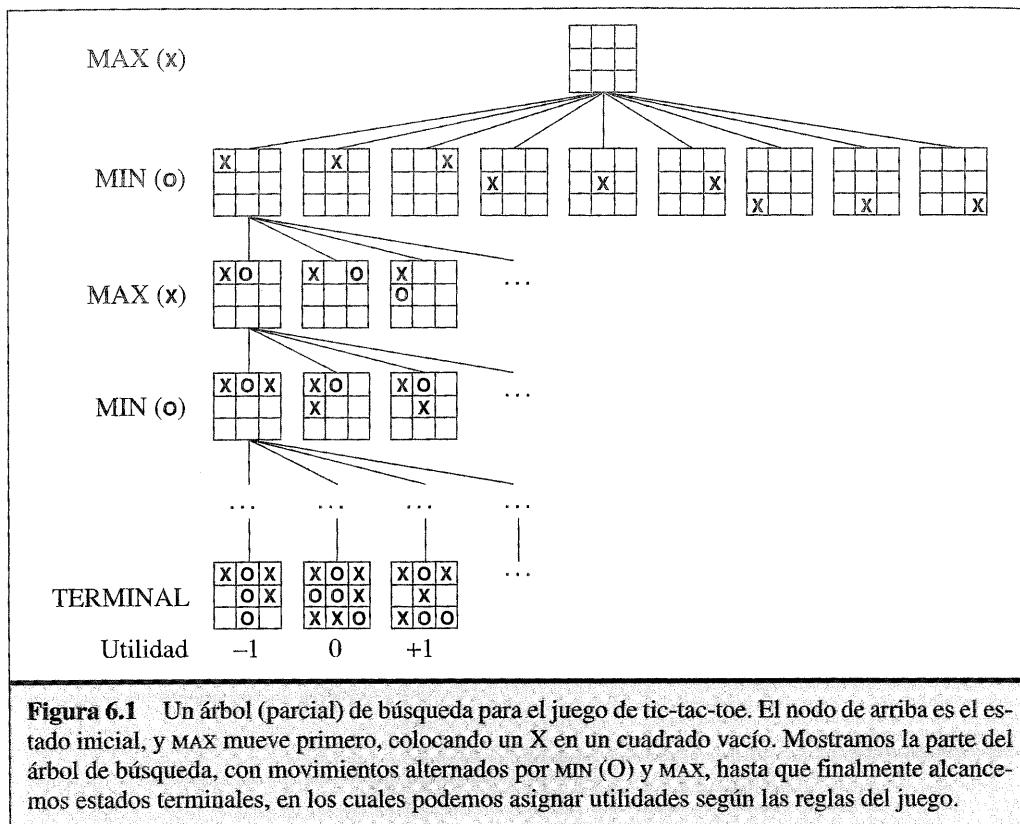
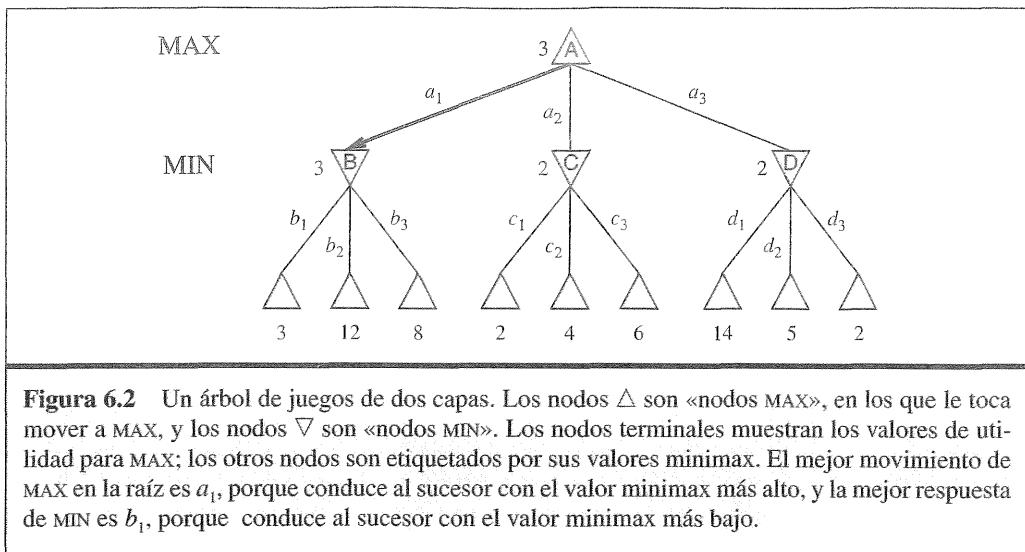


Figura 6.1 Un árbol (parcial) de búsqueda para el juego de tic-tac-toe. El nodo de arriba es el estado inicial, y MAX mueve primero, colocando un X en un cuadrado vacío. Mostramos la parte del árbol de búsqueda, con movimientos alternados por MIN (O) y MAX, hasta que finalmente alcanzamos estados terminales, en los cuales podemos asignar utilidades según las reglas del juego.

Incluso un juego simple como tic-tac-toe es demasiado complejo para dibujar el árbol de juegos entero, por tanto cambiemos al juego trivial de la Figura 6.2. Los movimientos posibles para MAX, en el nodo raíz, se etiquetan por a_1, a_2 , y a_3 . Las respuestas posibles a a_1 , para MIN, son b_1, b_2, b_3 , etc. Este juego particular finaliza después de un movimiento para MAX y MIN. (En el lenguaje de juegos, decimos que este árbol es un movimiento en profundidad, que consiste en dos medios movimientos, cada uno de los cuales se llama **capa**.) Las utilidades de los estados terminales en este juego varía desde dos a 14.

Considerando un árbol de juegos, la estrategia óptima puede determinarse examinando el **valor minimax** de cada nodo, que escribimos como el $\text{VALOR-MINIMAX}(n)$. El valor minimax de un nodo es la utilidad (para MAX) de estar en el estado correspondiente, *asumiendo que ambos jugadores juegan óptimamente* desde allí al final del juego. Obviamente, el valor minimax de un estado terminal es solamente su utilidad. Además, considerando una opción, MAX preferirá moverse a un estado de valor máximo, mientras que MIN prefiere un estado de valor mínimo. Entonces tenemos lo siguiente:

$$\text{VALOR-MINIMAX}(n) = \begin{cases} \text{UTILIDAD}(n) & \text{si } n \text{ es un estado terminal} \\ \max_{s \in \text{Sucesores}(n)} \text{VALOR-MINIMAX}(s) & \text{si } n \text{ es un estado MAX} \\ \min_{s \in \text{Sucesores}(n)} \text{VALOR-MINIMAX}(s) & \text{si } n \text{ es un estado MIN} \end{cases}$$



Aplicemos estas definiciones al árbol de juegos de la Figura 6.2. Los nodos terminales se etiquetan por sus valores de utilidad. El primer nodo de MIN, etiquetado B , tiene tres sucesores con valores 3, 12 y 8, entonces su valor minimax es 3. Del mismo modo, los otros dos nodos de MIN tienen un valor minimax de 2. El nodo raíz es un nodo MAX; sus sucesores tienen valores minimax de 3, 2 y 2; entonces tiene un valor minimax de 3. Podemos identificar también la **decisión minimax** en la raíz: la acción a_1 es la opción óptima para MAX porque conduce al sucesor con el valor minimax más alto.

Esta definición de juego óptimo para MAX supone que MIN también juega óptimamente (maximiza los resultados del *caso-peor* para MAX). ¿Y si MIN no juega óptimamente? Entonces es fácil demostrar (Ejercicio 6.2) que MAX lo hará aún mejor. Puede haber otras estrategias contra oponentes subóptimos que lo hagan mejor que la estrategia minimax; pero estas estrategias necesariamente lo hacen peor contra oponentes óptimos.

El algoritmo minimax

El **algoritmo minimax** (Figura 6.3) calcula la decisión minimax del estado actual. Usa un cálculo simple recurrente de los valores minimax de cada estado sucesor, directamente implementando las ecuaciones de la definición. La recursión avanza hacia las hojas del árbol, y entonces los valores minimax retroceden por el árbol cuando la recursión se va deshaciendo. Por ejemplo, en la Figura 6.2, el algoritmo primero va hacia abajo a los tres nodos izquierdos, y utiliza la función UTILIDAD para descubrir que sus valores son 3, 12 y 8 respectivamente. Entonces toma el mínimo de estos valores, 3, y lo devuelve como el valor del nodo B . Un proceso similar devuelve hacia arriba el valor de 2 para C y 2 para D . Finalmente, tomamos el máximo de 3, 2 y 2 para conseguir el valor de 3 para el nodo de raíz.

El algoritmo minimax realiza una exploración primero en profundidad completa del árbol de juegos. Si la profundidad máxima del árbol es m , y hay b movimientos legales

DECISIÓN MINIMAX

ALGORITMO MINIMAX

RETROCEDER

```

función DECISIÓN-MINIMAX(estado) devuelve una acción
variables de entrada: estado, estado actual del juego
    v  $\leftarrow$  MAX-VALOR(estado)
    devolver la acción de SUCESORES(estado) con valor v

función MAX-VALOR(estado) devuelve un valor utilidad
    si TEST-TERMINAL(estado) entonces devolver UTILIDAD(estado)
    v  $\leftarrow -\infty$ 
    para un s en SUCESORES(estado) hacer
        v  $\leftarrow$  MAX(v, MIN-VALOR(s))
    devolver v

función MIN-VALOR(estado) devuelve un valor utilidad
    si TEST-TERMINAL(estado) entonces devolver UTILIDAD(estado)
    v  $\leftarrow \infty$ 
    para un s en SUCESORES(estado) hacer
        v  $\leftarrow$  MIN(v, MAX-VALOR(s))
    devolver v

```

Figura 6.3 Un algoritmo para el cálculo de decisiones minimax. Devuelve la acción correspondiente al movimiento mejor posible, es decir, el movimiento que conduce al resultado con la mejor utilidad, conforme al axioma que el oponente juega para minimizar la utilidad. Las funciones VALOR-MAX y el VALOR-MIN pasan por el árbol de juegos entero, por todos los caminos hacia las hojas, para determinar el valor que le llega a un estado.

en cada punto, entonces la complejidad en tiempo del algoritmo minimax es $O(b^m)$. La complejidad en espacio es $O(bm)$ para un algoritmo que genere todos los sucesores a la vez, o $O(m)$ para un algoritmo que genere los sucesores uno por uno (véase la página 86). Para juegos reales, desde luego, los costos de tiempo son totalmente poco prácticos, pero este algoritmo sirve como base para el análisis matemático de juegos y para algoritmos más prácticos.

Decisiones óptimas en juegos multi-jugador

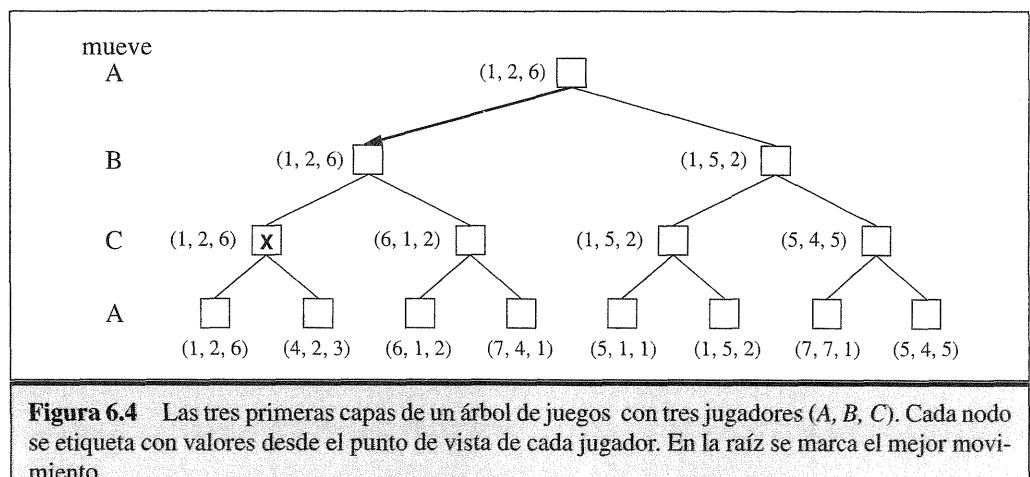
Muchos juegos populares permiten más de dos jugadores. Examinemos cómo obtener una extensión de la idea minimax a juegos multi-jugador. Esto es sencillo desde el punto de vista técnico, pero proporciona algunas nuevas cuestiones conceptuales interesantes.

Primero, tenemos que sustituir el valor para cada nodo con un *vector* de valores. Por ejemplo, en un juego de tres jugadores con jugadores *A*, *B* y *C*, un vector $\langle v_A, v_B, v_C \rangle$ asociado con cada nodo. Para los estados terminales, este vector dará la utilidad del estado desde el punto de vista de cada jugador. (En dos jugadores, en juegos de suma cero, el vector de dos elementos puede reducirse a un valor porque los valores son siempre opuestos.) El camino más simple de implementarlo es hacer que la función UTILIDAD devuelva un vector de utilidades.

Ahora tenemos que considerar los estados no terminales. Consideremos el nodo marcado con X en el árbol de juegos de la Figura 6.4. En ese estado, el jugador C elige qué hacer. Las dos opciones conducen a estados terminales con el vector de utilidad $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ y $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Como 6 es más grande que 3, C debería elegirlo como primer movimiento. Esto significa que si se alcanza el estado X , el movimiento siguiente conducirá a un estado terminal con utilidades $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. De ahí, que el valor que le llega a X es este vector. En general, el valor hacia atrás de un nodo n es el vector de utilidad de cualquier sucesor que tiene el valor más alto para el jugador que elige en n .

Alguien que juega juegos multi-jugador, como Diplomacy™, rápidamente se da cuenta de que hay muchas más posibilidades que en los juegos de dos jugadores. Los juegos multi-jugador, por lo general, implican **alianzas**, formales o informales, entre los jugadores. Se hacen y se rompen las alianzas conforme avanza el juego. ¿Cómo debemos entender tal comportamiento? ¿Las alianzas son una consecuencia natural de estrategias óptimas para cada jugador en un juego multi-jugador? Resulta que pueden ser. Por ejemplo suponga que A y B están en posiciones débiles y C está en una posición más fuerte. Entonces, a menudo, es óptimo tanto para A como para B atacar a C más que el uno al otro, por temor a que C los destruya. De esta manera, la colaboración surge del comportamiento puramente egoísta. Desde luego, tan pronto como C se debilita bajo el ataque conjunto, la alianza pierde su valor, y A o B podrían violar el acuerdo. En algunos casos, las alianzas explícitas solamente concretan lo que habría pasado de todos modos. En otro caso hay un estigma social a la rotura de una alianza, así que los jugadores deben equilibrar la ventaja inmediata de romper una alianza contra la desventaja a largo plazo de ser percibido como poco fiable. Véase la Sección 17.6 para estas complicaciones.

Si el juego no es de suma cero, la colaboración puede ocurrir también con sólo dos jugadores. Suponga, por ejemplo, que hay un estado terminal con utilidades $\langle v_A = 1.000, v_B = 1.000 \rangle$, y que 1.000 es la utilidad más alta posible para cada jugador. Entonces la estrategia óptima para ambos jugadores es hacer todo lo posible por alcanzar este estado (es decir los jugadores cooperarán automáticamente para conseguir un objetivo mutuamente deseable).



6.3 Poda alfa-beta

El problema de la búsqueda minimax es que el número de estados que tiene que examinar es exponencial en el número de movimientos. Lamentablemente no podemos eliminar el exponente, pero podemos dividirlo, con eficacia, en la mitad. La jugada es que es posible calcular la decisión minimax correcta sin mirar todos los nodos en el árbol de juegos. Es decir podemos tomar prestada la idea de **podar** del Capítulo 4 a fin de

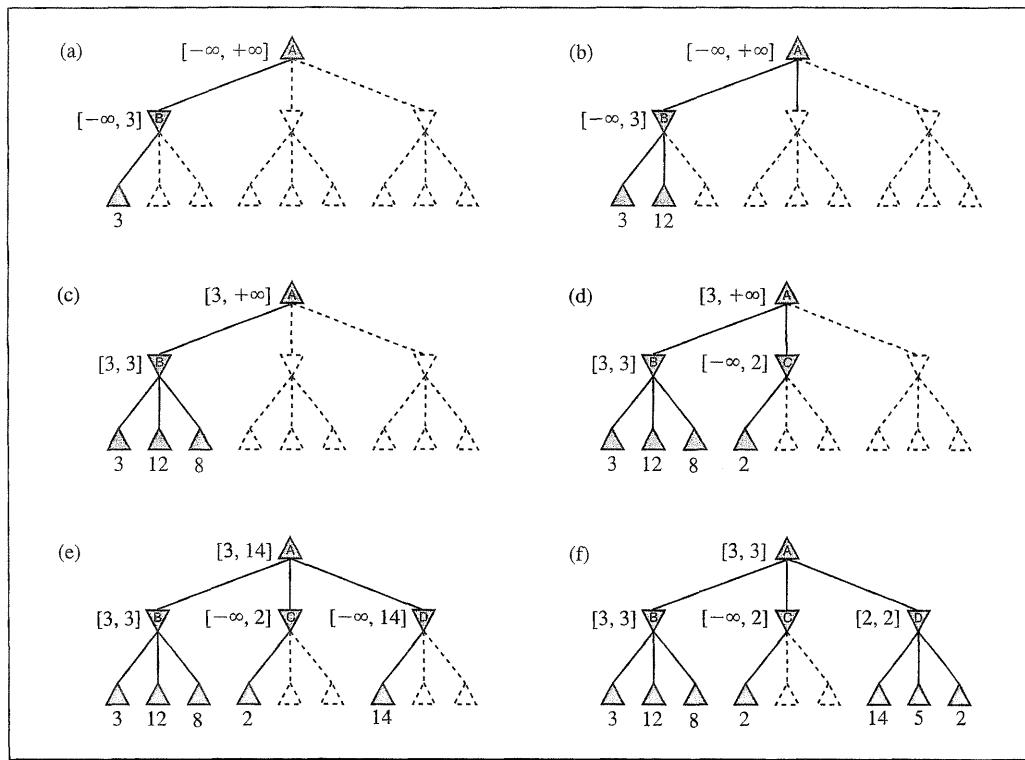


Figura 6.5 Etapas en el cálculo de la decisión óptima para el árbol de juegos de la Figura 6.2. En cada punto, mostramos el rango de valores posibles para cada nodo. (a) La primera hoja debajo de B tiene el valor 3. De ahí B , que es un nodo MIN, tiene un valor de *como máximo* 3. (b) La segunda hoja debajo de B tiene un valor de 12; MIN evitaría este movimiento, entonces el valor de B es todavía como máximo 3. (c) La tercera hoja debajo de B tiene un valor de 8; hemos visto a todos los sucesores de B , entonces el valor de B es exactamente 3. Ahora, podemos deducir que el valor de la raíz es *al menos* 3, porque MAX tiene una opción digna de 3 en la raíz. (d) La primera hoja debajo de C tiene el valor 2. De ahí, C , que es un nodo MIN, tiene un valor de *como máximo* 2. Pero sabemos que B vale 3, entonces MAX nunca elegiría C . Por lo tanto, no hay ninguna razón en mirar a los otros sucesores de C . Este es un ejemplo de la poda de alfa-beta. (e) La primera hoja debajo de D tiene el valor 14, entonces D vale *como máximo* 14. Este es todavía más alto que la mejor alternativa de MAX (es decir, 3), entonces tenemos que seguir explorando a los sucesores de D . Note también que ahora tenemos límites sobre todos los sucesores de la raíz, entonces el valor de la raíz es también como máximo 14. (f) El segundo sucesor de D vale 5, así que otra vez tenemos que seguir explorando. El tercer sucesor vale 2, así que ahora D vale exactamente 2. La decisión de MAX en la raíz es moverse a B , dando un valor de 3.

PODA ALFA-BETA

eliminar partes grandes del árbol. A la técnica que examinaremos se le llama **poda alfa-beta**. Cuando lo aplicamos a un árbol minimax estándar, devuelve el mismo movimiento que devolvería minimax, ya que podar las ramas no puede influir, posiblemente, en la decisión final.

Consideremos otra vez el árbol de juegos de dos capas de la Figura 6.2. Vamos a realizar el cálculo de la decisión óptima una vez más, esta vez prestando atención a lo que sabemos en cada punto del proceso. En la Figura 6.5 se explican los pasos. El resultado es que podemos identificar la decisión minimax sin evaluar dos de los nodos hoja.

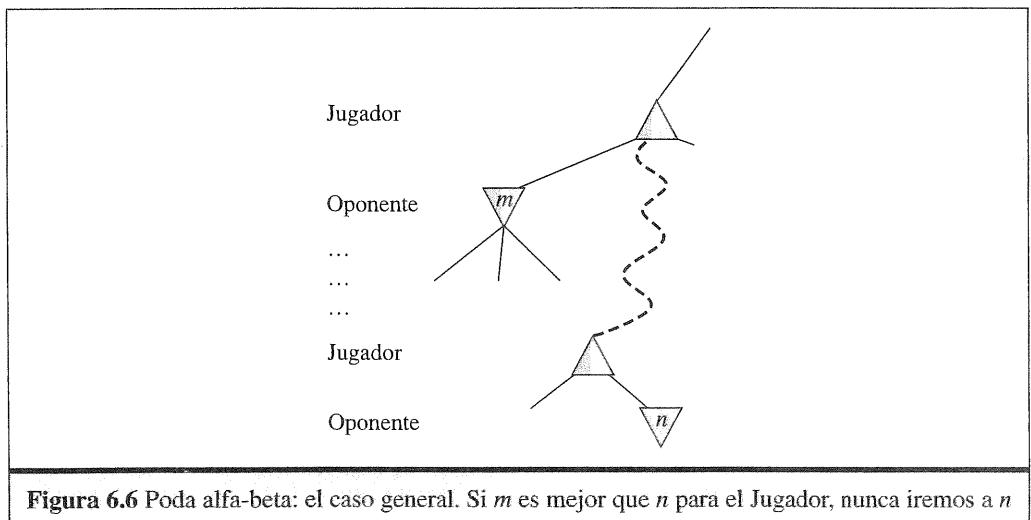
Otro modo de verlo es como una simplificación de la fórmula **VALOR-MINIMAX**. Los dos sucesores no evaluados del nodo *C* de la Figura 6.5 tienen valores *x* e *y*, y sea *z* el mínimo entre *x* e *y*. El valor del nodo raíz es

$$\begin{aligned}\text{MINIMAX-VALUE}(raíz) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \quad \text{donde } z \leq 2 \\ &= 3\end{aligned}$$

En otras palabras, el valor de la raíz y de ahí la decisión minimax son *independientes* de los valores de las hojas podadas *x* e *y*.

 La poda alfa-beta puede aplicarse a árboles de cualquier profundidad, y, a menudo, es posible podar subárboles enteros. El principio general es: considere un nodo *n* en el árbol (véase la Figura 6.6), tal que el Jugador tiene una opción de movimiento a ese nodo. Si el Jugador tiene una mejor selección *m* en el nodo padre de *n* o en cualquier punto más lejano, entonces *n* *nunca será alcanzado en el juego actual*. Una vez que hemos averiguado bastante sobre *n* (examinando a algunos de sus descendientes) para alcanzar esta conclusión, podemos podarlo.

Recuerde que la búsqueda minimax es primero en profundidad, así que en cualquier momento solamente tenemos que considerar los nodos a lo largo de un camino en el árbol.



La poda alfa-beta consigue su nombre de los dos parámetros que describen los límites sobre los valores hacia atrás que aparecen a lo largo del camino:

α = el valor de la mejor opción (es decir, valor más alto) que hemos encontrado hasta ahora en cualquier punto elegido a lo largo del camino para MAX.

β = el valor de la mejor opción (es decir, valor más bajo) que hemos encontrado hasta ahora en cualquier punto elegido a lo largo del camino para MIN.

La búsqueda alfa-beta actualiza el valor de α y β según se va recorriendo el árbol y poda las ramas restantes en un nodo (es decir, termina la llamada recurrente) tan pronto como el valor del nodo actual es peor que el actual valor α o β para MAX o MIN, respectivamente. La Figura 6.7 nos da el algoritmo completo. Animamos al lector a trazar su comportamiento cuando lo aplicamos al árbol de la Figura 6.5.

La eficacia de la poda alfa-beta es muy dependiente del orden en el que se examinan los sucesores. Por ejemplo, en la Figura 6.5(e) y (f), podríamos no podar ningún

función BÚSQUEDA-ALFA-BETA(*estado*) **devuelve** una acción

variables de entrada: *estado*, estado actual del juego

$v \leftarrow \text{MAX-VALOR}(\text{estado}, -\infty, +\infty)$

devolver la acción de SUCESORES(*estado*) con valor *v*

función MAX-VALOR(*estado*, α , β) **devuelve** un valor utilidad

variables de entrada: *estado*, estado actual del juego

α , valor de la mejor alternativa para MAX a lo largo del camino a *estado*

β , valor de la mejor alternativa para MIN a lo largo del camino a *estado*

si TEST-TERMINAL(*estado*) **entonces devolver** UTILIDAD(*estado*)

$v \leftarrow -\infty$

para *a*, *s* en SUCESORES(*estado*) **hacer**

$v \leftarrow \text{MAX}(V, \text{MIN-VALOR}(s, \alpha, \beta))$

si *v* $\geq \beta$ **entonces devolver** *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

devolver *v*

función MIN-VALOR(*estado*, α , β) **devuelve** un valor utilidad

variables de entrada: *estado*, estado actual del juego

α , valor de la mejor alternativa para MAX a lo largo del camino a *estado*

β , valor de la mejor alternativa para MIN a lo largo del camino a *estado*

si TEST-TERMINAL(*estado*) **entonces devolver** UTILIDAD(*estado*)

$v \leftarrow +\infty$

para *a*, *s* en SUCESORES(*estado*) **hacer**

$v \leftarrow \text{MIN}(v, \text{MAX-VALOR}(s, \alpha, \beta))$

si *v* $\leq \alpha$ **entonces devolver** *v*

$\beta \leftarrow \text{MIN}(\beta, V)$

devolver *v*

Figura 6.7 El algoritmo de búsqueda alfa-beta. Notemos que estas rutinas son lo mismo que las rutinas de MINIMAX de la Figura 6.3, excepto las dos líneas MIN-VALOR y MAX-VALOR que mantienen α y β (y la forma de hacer pasar estos parámetros).

sucesor de D si se hubieran generado primero los sucesores peores (desde el punto de vista de MIN). Si el tercer sucesor se hubiera generado primero, habríamos sido capaces de podar los otros dos. Esto sugiere que pudiera valer la pena tratar de examinar primero los sucesores que probablemente serán los mejores.

Si asumimos que esto puede hacerse², resulta que alfa-beta tiene que examinar sólo $O(b^{d/2})$ nodos para escoger el mejor movimiento, en vez de $O(b^d)$ para minimax. Esto significa que el factor de ramificación eficaz se hace \sqrt{b} en vez de b (para el ajedrez, seis en vez de 35). De otra manera, alfa-beta puede mirar hacia delante aproximadamente dos veces más que minimax en la misma cantidad del tiempo. Si los sucesores se examinan en orden aleatorio más que primero el mejor, el número total de nodos examinados será aproximadamente $O(b^{3d/4})$ para un moderado b . Para el ajedrez, una función de ordenación bastante sencilla (como intentar primero las capturas, luego las amenazas, luego mover hacia delante, y, por último, los movimientos hacia atrás) lo consiguen en aproximadamente un factor de dos del resultado $O(b^{d/2})$ del mejor caso. Añadir esquemas dinámicos que ordenan movimientos, como intentar primero los movimientos que fueron mejores la última vez, llegamos bastante cerca del límite teórico.

En el Capítulo 3, vimos que los estados repetidos en el árbol de búsqueda pueden causar un aumento exponencial del coste de búsqueda. En juegos, los estados repetidos ocurren con frecuencia debido a **transposiciones** (permutaciones diferentes de la secuencia de movimientos que terminan en la misma posición). Por ejemplo, si Blanco tiene un movimiento a_1 que puede ser contestado por Negro con b_1 y un movimiento no relacionado a_2 del otro lado del tablero puede ser contestado por b_2 , entonces las secuencias $[a_1, b_1, a_2, b_2]$ y $[a_1, b_2, a_2, b_1]$ terminan en la misma posición (como las permutaciones que comienzan con a_2). Vale la pena almacenar la evaluación de esta posición en una tabla *hash* la primera vez que se encuentre, de modo que no tuviéramos que volver a calcularlo las siguientes veces. Tradicionalmente a la tabla *hash* de posiciones se le llama **tabla de transposición**; es esencialmente idéntica a la lista *cerrada* en la BÚSQUEDA-GRAFOS (página 93). La utilización de una tabla de transposiciones puede tener un efecto espectacular a veces, tanto como doblar la profundidad accesible de búsqueda en el ajedrez. Por otra parte, si evaluamos un millón de nodos por segundo, no es práctico guardar *todos* ellos en la tabla de transposiciones. Se han utilizado varias estrategias para elegir los más valiosos.

TRANSPOSICIONES

TABLA DE
TRANSPOSICIONES

6.4 Decisiones en tiempo real imperfectas

El algoritmo minimax genera el espacio de búsqueda entero, mientras que el algoritmo alfa-beta permite que podemos partes grandes de él. Sin embargo, alfa-beta todavía tiene que buscar en todos los caminos, hasta los estados terminales, para una parte del espacio de búsqueda. Esta profundidad no es, por lo general, práctica porque los movimientos deben hacerse en una cantidad razonable de tiempo (típicamente, unos minutos como máximo). El trabajo de Shannon en 1950, *Programming a computer for playing chess*, pro-

² Obviamente, no puede hacerse; por otra parte ¡la función de ordenación podría usarse para jugar un juego perfecto!

puso un cambio: que los programas deberían cortar la búsqueda antes y entonces aplicar una **función de evaluación** heurística a los estados, convirtiendo, efectivamente, los nodos no terminales en hojas terminales. En otras palabras, la sugerencia deberá alterar minimax o alfa-beta de dos formas: se sustituye la función de utilidad por una función de evaluación heurística EVAL, que da una estimación de la utilidad de la posición, y se sustituye el test-terminal por un **test-límite** que decide cuándo aplicar EVAL.

TEST-LÍMITE

Funciones de evaluación

Una función de evaluación devuelve una estimación de la utilidad esperada de una posición dada, tal como hacen las funciones heurísticas del Capítulo 4 que devuelven una estimación de la distancia al objetivo. La idea de una estimación no era nueva cuando Shannon la propuso. Durante siglos, los jugadores de ajedrez (y aficionados de otros juegos) han desarrollado modos de juzgar el valor de una posición, debido a que la gente es aún más limitada, en cantidad de la búsqueda, que los programas de computador. Debería estar claro que el funcionamiento de un programa de juegos es dependiente de la calidad de su función de evaluación. Una función de evaluación inexacta dirigirá a un agente hacia posiciones que resultan estar perdidas. ¿Cómo diseñamos funciones de evaluación buenas?

Primero, la función de evaluación debería ordenar los estados *terminales* del mismo modo que la función de utilidad verdadera; por otra parte, un agente que la use podría seleccionar movimientos subóptimos aunque pueda ver delante el final del juego. Segundo, ¡el cálculo no debe utilizar demasiado tiempo! (La función de evaluación podría llamar a la DECISIÓN-MINIMAX como una subrutina y calcular el valor exacto de la posición, pero esto frustraría nuestro propósito: ahorrar tiempo.) Tercero, para estados no terminales, la función de evaluación debería estar fuertemente correlacionada con las posibilidades actuales de ganar.

Uno podría preguntarse sobre la frase «las posibilidades de ganar». Después de todo, el ajedrez no es un juego de azar: sabemos el estado actual con certeza. Pero si la búsqueda debe cortarse en estados no terminales, entonces necesariamente el algoritmo será *incierto* sobre los resultados finales de esos estados. Este tipo de incertidumbre está inducida por limitaciones computacionales, más que de información. Si consideramos la cantidad limitada de cálculo que se permiten a la función de evaluación cuando se aplica a un estado, lo mejor que se podría hacer es una conjectura sobre el resultado final.

CARACTERÍSTICAS

Hagamos esta idea más concreta. La mayoría de las funciones de evaluación trabajan calculando varias **características** del estado (por ejemplo, en el juego de ajedrez, el número de peones capturados por cada lado). Las características, juntas, definen varias *categorías* o *clases de equivalencia* de estados: los estados en cada categoría tienen los mismos valores para todas las características. Cualquier categoría dada, por lo general, tendrá algunos estados que conducen a triunfos, algunos que conducen a empates, y algunos que conducen a pérdidas. La función de evaluación no sabe cuál es cada estado, pero sí puede devolver un valor que refleje la *proporción* de estados con cada resultado. Por ejemplo, supongamos que nuestra experiencia sugiere que el 72 por ciento de los estados encontrados en la categoría conduce a un triunfo (utilidad +1);

VALOR ESPERADO

el 20 por ciento a una pérdida (-1), y el 8 por ciento a un empate (0). Entonces una evaluación razonable de estados en la categoría es el valor medio ponderado o **valor esperado**: $(0,72 \times +1) + (0,20 \times -1) + (0,08 \times 0) = 0,52$. En principio, el valor esperado se puede determinar para cada categoría, produciendo una función de evaluación que trabaja para cualquier estado. Mientras que con los estados terminales, la función de evaluación no tiene que devolver valores actuales esperados, la *ordenación* de los estados es el mismo.

VALOR MATERIAL

En la práctica, esta clase de análisis requiere demasiadas categorías y demasiada experiencia para estimar todas las probabilidades de ganar. En cambio, la mayoría de las funciones de evaluación calculan las contribuciones numéricas de cada característica y luego las *combinan* para encontrar el valor total. Por ejemplo, los libros de ajedrez dan, de forma aproximada, el **valor material** para cada pieza: cada peón vale 1, un caballo o el alfil valen 3, una torre 5, y la reina vale 9. Otras características como «la estructura buena del peón» y «seguridad del rey» podrían valer la mitad de un peón, por ejemplo. Estos valores de características, entonces, simplemente se suman para obtener la evaluación de la posición. Una ventaja segura equivalente a un peón da una probabilidad sustancial de ganar, y una ventaja segura equivalente a tres peones debería dar la casi victoria, como se ilustra en la Figura 6.8(a). Matemáticamente, a esta clase de función de evaluación se le llama **función ponderada lineal**, porque puede expresarse como

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s),$$

donde cada w_i es un peso y cada f_i es una característica de la posición. Para el ajedrez, los f_i podrían ser los números de cada clase de piezas sobre el tablero, y los w_i podrían ser los valores de las piezas (1 para el peón, 3 para el alfil, etc.).

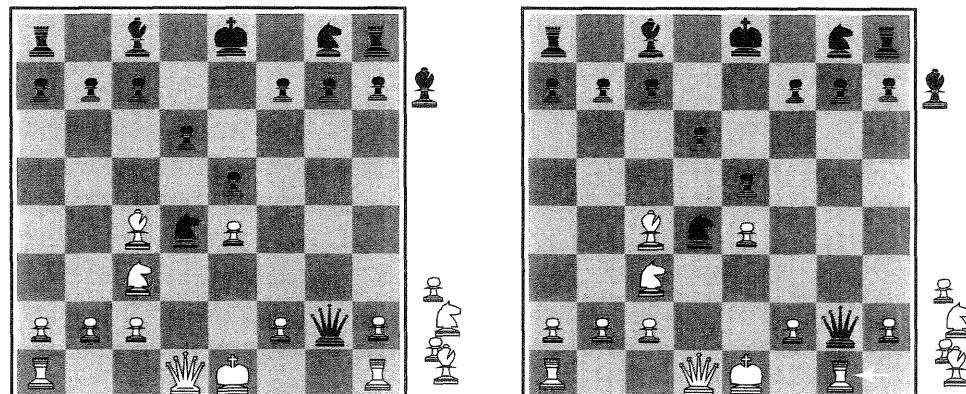


Figura 6.8 Dos posiciones ligeramente diferentes del ajedrez. En (a), negro tiene una ventaja de un caballo y dos peones y ganará el juego. En (b), negro perderá después de que blanca capture la reina.

La suma de los valores de las características parece razonable, pero de hecho implica un axioma muy fuerte: que la contribución de cada característica sea *independiente* de los valores de las otras características. Por ejemplo, la asignación del valor 3 a un alfil no utiliza el hecho de que el alfil es más poderoso en la fase final, cuando tienen mucho espacio para maniobrar. Por esta razón, los programas actuales para el ajedrez, y otros juegos, también utilizan combinaciones *no lineales* de características. Por ejemplo, un par de alfils podría merecer más la pena que dos veces el valor de un alfil, y un alfil merece más la pena en la fase final que al principio.

¡El lector habrá notado que las características y los pesos *no* son parte de las reglas del ajedrez! Provienen, desde hace siglos, de la experiencia humana al jugar al ajedrez. Considerando la forma lineal de evaluación, las características y los pesos dan como resultado la mejor aproximación a la ordenación exacta de los estados según su valor. En particular, la experiencia sugiere que una ventaja de más de un punto probablemente gane el juego, si no intervienen otros factores; una ventaja de tres puntos es suficiente para una victoria. En juegos donde no está disponible esta clase de experiencia, los pesos de la función de evaluación pueden estimarse con las técnicas de aprendizaje del Capítulo 18. La aplicación de estas técnicas al ajedrez han confirmado que un alfil es, en efecto, como aproximadamente tres peones.

Corte de la búsqueda

El siguiente paso es modificar la BUSQUEDA-ALFA-BETA de modo que llame a la función heurística EVAL cuando se corte la búsqueda. En términos de implementación, sustituimos las dos líneas de la Figura 6.7, que mencionan al TEST-TERMINAL, con la línea siguiente:

```
si TEST-CORTE(estado, profundidad) entonces devolver EVAL(estado)
```

También debemos llevar la contabilidad de la *profundidad* de modo que la profundidad actual se incremente sobre cada llamada recursiva. La aproximación más sencilla para controlar la cantidad de búsqueda es poner un límite de profundidad fijo, de modo que TEST-CORTE(*estado, profundidad*) devuelva verdadero para toda profundidad mayor que alguna profundidad fija *d*. (También debe devolver verdadero para todos los estados terminales, tal como hizo el TEST-TERMINAL.) La profundidad *d* se elige de modo que la cantidad de tiempo usado no exceda de lo que permiten las reglas del juego.

Una aproximación más robusta es aplicar profundidad iterativa, como se definió en el Capítulo 3. Cuando el tiempo se agota, el programa devuelve el movimiento seleccionado por la búsqueda completa más profunda. Sin embargo, estas aproximaciones pueden conducir a errores debido a la naturaleza aproximada de la función de evaluación. Considere otra vez la función de evaluación simple para el ajedrez basada en la ventaja del material. Suponga los programas de búsquedas al límite de profundidad, alcanzando la posición de la Figura 6.8(b), donde Negro aventaja por un caballo y dos peones. Esto estaría representado por el valor heurístico del estado, declarando que el estado conducirá probablemente a un triunfo de Negro. Pero en el siguiente movimiento de Blanco captura a la reina de Negro. De ahí, que la posición es realmente ganadora para Blanco, y que puede verse mirando una capa más.

ESTABLE

BÚSQUEDA DE ESTABILIDAD

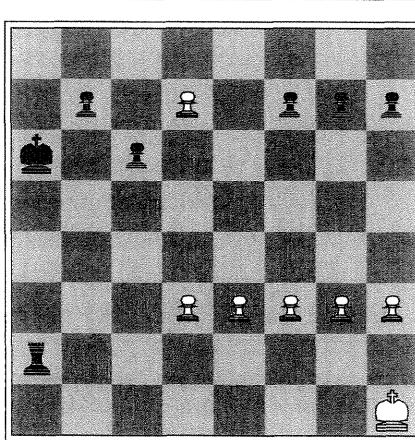
EFECTO HORIZONTE

EXTENSIONES EXCEPCIONALES

Obviamente, se necesita un test del límite más sofisticado. La función de evaluación debería aplicarse sólo a posiciones que son estables (es decir, improbablemente expuestas a grandes oscilaciones en su valor en un futuro próximo). En el ajedrez, por ejemplo, las posiciones en las cuales se pueden hacer capturas no son estables para una función de evaluación que solamente cuenta el material. Las posiciones no estables se pueden extender hasta que se alcancen posiciones estables. A esta búsqueda suplementaria se le llama **búsqueda de estabilidad o de reposo**; a veces se restringe a sólo ciertos tipos de movimientos, como movimientos de captura, que resolverán rápidamente la incertidumbre en la posición.

El **efecto horizonte** es más difícil de eliminar. Se produce cuando el programa afronta un movimiento, del oponente, que causa un daño serio e inevitable. Consideré el juego de ajedrez de la Figura 6.9. Negro aventaja en el material, pero si Blanco puede avanzar su peón de la séptima fila a la octava, el peón se convertirá en una reina y creará un triunfo fácil para Blanco. Negro puede prevenir este resultado en la capa 14 dando jaque a la reina blanca con la torre, pero al final, inevitablemente el peón se convertirá en una reina. El problema con la búsqueda de profundidad fija es que se cree que esquivar estos movimientos evitan el movimiento de convertirse en reina, decimos que los movimientos de esquivar empujan el movimiento de convertirse en reina (inevitable) «sobre el horizonte de búsqueda» a un lugar donde no puede detectarse.

Cuando las mejoras de *hardware* nos lleven a realizar búsquedas más profundas, se espera que el efecto horizonte ocurra con menos frecuencia (las secuencias que tardan mucho tiempo son bastante raras). El uso de **extensiones excepcionales** también ha sido bastante eficaz para evitar el efecto horizonte sin añadir demasiado coste a la búsqueda. Una extensión excepcional es un movimiento que es «claramente mejor» que todos los demás en una posición dada. Una búsqueda de extensión excepcional puede ir más allá del límite de profundidad normal sin incurrir mucho en el coste porque su factor de



Negra mueve

Figura 6.9 El efecto horizonte. Una serie de jaques de la torre negra fuerza al movimiento de convertirse en reina (inevitabile) de Blanco «sobre el horizonte» y hace que esta posición parezca un triunfo para Negro, cuando realmente es un triunfo para Blanco.

ramificación es 1. (Se puede pensar que la búsqueda de estabilidad es una variante de extensiones excepcionales.) En la Figura 6.9, una búsqueda de extensión excepcional encontrará el movimiento de convertirse en reina, a condición de que los movimientos de jaque de Negro y los movimientos del rey blanco puedan identificarse como «claramente mejores» que las alternativas.

Hasta ahora hemos hablado del corte de la búsqueda a un cierto nivel y que la poda alfa-beta, probablemente, no tiene ningún efecto sobre el resultado. Es también posible hacer la **poda hacia delante**, en la que podamos inmediatamente algunos movimientos de un nodo. Claramente, la mayoría de la gente que juega al ajedrez sólo considera unos pocos movimientos de cada posición (al menos conscientemente). Lamentablemente, esta aproximación es bastante peligrosa porque no hay ninguna garantía de que el mejor movimiento no sea podado. Esto puede ser desastroso aplicado cerca de la raíz, porque entonces a menudo el programa omitirá algunos movimientos «evidentes». La poda hacia delante puede usarse en situaciones especiales (por ejemplo, cuando dos movimientos son simétricos o equivalentes, sólo consideramos uno) o para nodos profundos en el árbol de búsqueda.

La combinación de todas las técnicas descritas proporciona un programa que puede jugar al ajedrez loablemente (o a otros juegos). Asumamos que hemos implementado una función de evaluación para el ajedrez, un test del límite razonable con una búsqueda de estabilidad, y una tabla de transposiciones grande. También asumamos que, después de meses de intentos tediosos, podemos generar y evaluar alrededor de un millón de nodos por segundo sobre los últimos PCs, permitiéndonos buscar aproximadamente 200 millones de nodos por movimiento bajo un control estándar del tiempo (tres minutos por movimiento). El factor de ramificación para el ajedrez es aproximadamente 35, como media, y 35^5 son aproximadamente 50 millones, así si usamos la búsqueda minimax podríamos mirar sólo cinco capas. Aunque no sea incompetente, tal programa puede ser engañado fácilmente por un jugador medio de ajedrez, el cual puede planear, de vez en cuando, seis u ocho capas. Con la búsqueda alfa-beta nos ponemos en aproximadamente 10 capas, y resulta un nivel experto del juego. La Sección 6.7 describe técnicas de poda adicionales que pueden ampliar la profundidad eficaz de búsqueda a aproximadamente 14 capas. Para alcanzar el nivel de gran maestro necesitaríamos una función de evaluación ajustada y una base de datos grande de movimientos de apertura y movimientos finales óptimos. No sería malo tener un supercomputador para controlar este programa.

6.5 Juegos que incluyen un elemento de posibilidad

En la vida real, hay muchos acontecimientos imprevisibles externos que nos ponen en situaciones inesperadas. Muchos juegos reflejan esta imprevisibilidad con la inclusión de un elemento aleatorio, como el lanzamiento de dados. De esta manera, ellos nos dan un paso más cercano a la realidad, y vale la pena ver cómo afecta al proceso de toma de decisiones.

Backgammon es un juego típico que combina la suerte y la habilidad. Se hacen rodar unos dados, al comienzo del turno de un jugador, para determinar los movimientos

legales. En la posición *backgammon* de la Figura 6.10, por ejemplo, Blanco ha hecho rodar un 6-5, y tiene cuatro movimientos posibles.

Aunque Blanco sabe cuáles son sus propios movimientos legales, no sabe lo que le va a salir a Negro con los dados y por eso no sabe cuáles serán sus movimientos legales. Esto significa que Blanco no puede construir un árbol de juegos estándar de la forma que vimos en el ajedrez y tic-tac-toe. Un árbol de juegos en el *backgammon* debe incluir nodos de posibilidad además de los nodos MAX y MIN. En la Figura 6.11 se rodean con círculos los nodos de posibilidad. Las ramas que salen desde cada nodo posibilidad denotan las posibles tiradas, y cada una se etiqueta con la tirada y la posibilidad de que ocurra. Hay 36 resultados al hacer rodar dos dados, cada uno igualmente probable; pero como un 6-5 es lo mismo que un 5-6, hay sólo 21 resultado distintos. Los seis dobles (1-1 a 6-6) tienen una posibilidad de 1/36, los otros 15 resultados distintos un 1/18 cada uno.

El siguiente paso es entender cómo se toman las decisiones correctas. Obviamente, todavía queremos escoger el movimiento que conduzca a la mejor posición. Sin embargo, las posiciones no tienen valores minimax definidos. En cambio, podemos calcular el **valor esperado**, donde la expectativa se toma sobre todos los posibles resultados que podrían ocurrir. Éste nos conduce a generalizar el **valor minimax** para juegos deterministas a un **valor minimaxesperado** para juegos con nodos de posibilidad. Los nodos terminales y los MAX y MIN (para los que se conocen sus resultados) trabajan exactamente del mismo modo que antes; los nodos de posibilidad se evalúan tomando

NODOS DE POSIBILIDAD

VALOR
MINIMAXESPERADO

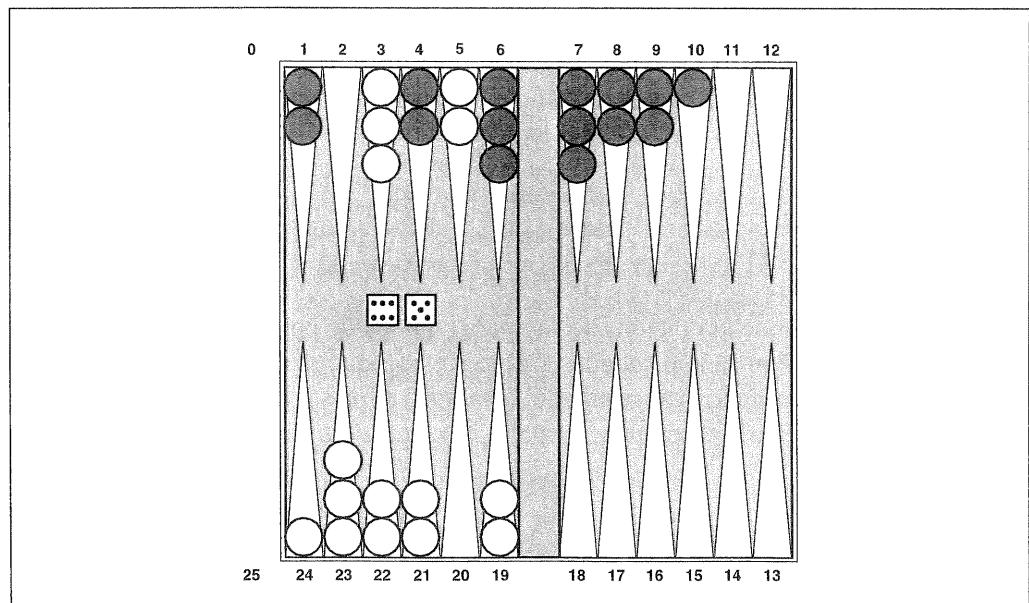


Figura 6.10 Una posición típica del *backgammon*. El objetivo del juego es mover todas las fichas del tablero. Blanco mueve a la derecha hacia 25, y los movimientos de Negro al contrario, hacia 0. Una ficha puede moverse a cualquier posición a menos que haya varias piezas del oponente; si hay un oponente, es capturado y debe comenzar. En la posición mostrada, Blanca ha sacado 6-5 y debe elegir entre cuatro movimientos legales: (5-10,5-11), (5-11,19-24), (5-10,10-16) y (5-11,11-16).

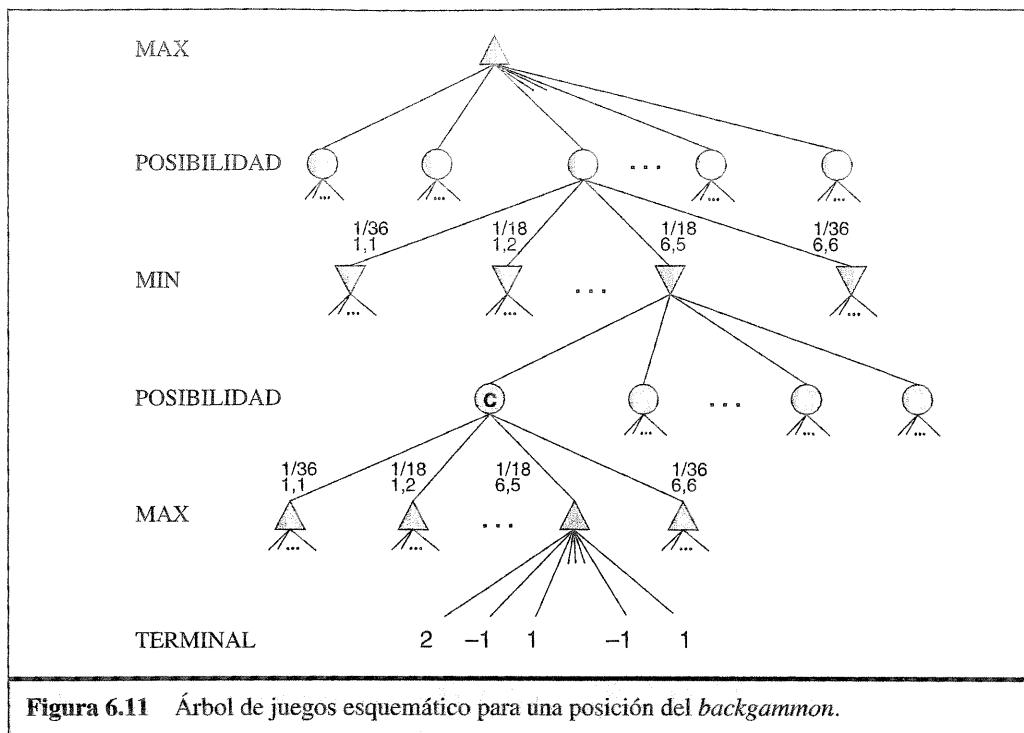


Figura 6.11 Árbol de juegos esquemático para una posición del *backgammon*.

el promedio ponderado de los valores que se obtienen de todos los resultados posibles, es decir,

$$\text{MINIMAXESPERADO}(n) =$$

$$\begin{cases} \text{UTILIDAD}(n) & \text{si } n \text{ es un nodo terminal} \\ \max_{s \in \text{Sucesores}(n)} \text{MINIMAXESPERADO}(s) & \text{si } n \text{ es un nodo MAX} \\ \min_{s \in \text{Sucesores}(n)} \text{MINIMAXESPERADO}(s) & \text{si } n \text{ es un nodo MIN} \\ \sum_{s \in \text{Sucesores}(n)} P(s) \cdot \text{MINIMAXESPERADO}(s) & \text{si } n \text{ es un nodo posibilidad} \end{cases}$$

donde la función sucesor para un nodo de posibilidad n simplemente aumenta el estado n con cada resultado posible para producir cada sucesor s , y $P(s)$ es la probabilidad de que ocurra ese resultado. Estas ecuaciones pueden aplicarse recursivamente hasta la raíz del árbol, como en minimax. Dejamos los detalles del algoritmo como un ejercicio.

Evaluación de la posición en juegos con nodos de posibilidad

Como con minimax, la aproximación obvia es cortar la búsqueda en algún punto y aplicar una función de evaluación a cada hoja. Uno podría pensar que las funciones de evaluación para juegos como *backgammon* deberían ser como funciones de evaluación para el ajedrez (solamente tienen que dar tanteos más altos a mejores posiciones). Pero de hecho, la presencia de nodos de posibilidades significa que uno tiene que tener más cui-

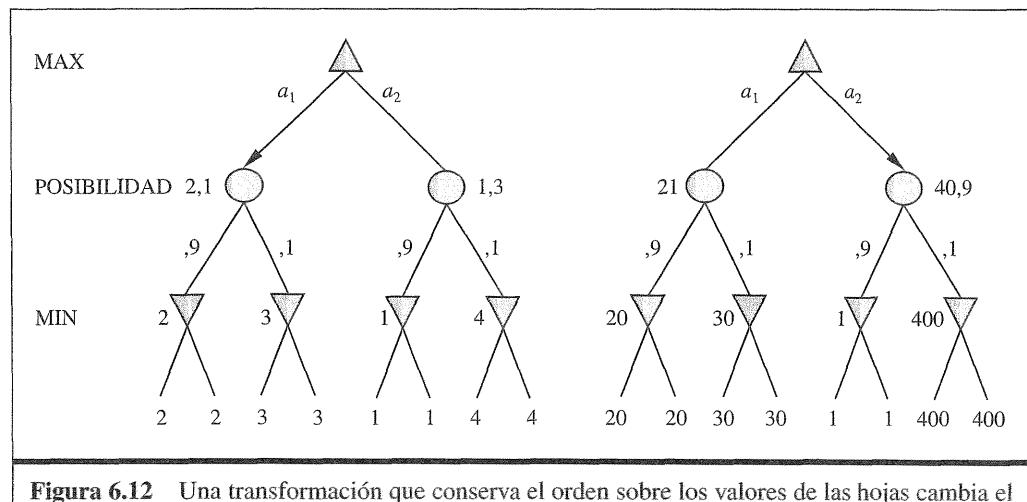
dado sobre la evaluación de valores medios. La Figura 6.12 muestra lo que sucede: con una función de evaluación que asigne valores [1,2,3,4] a las hojas, el movimiento A_1 es el mejor; con valores [1,20,30,400], el movimiento A_2 es el mejor. ¡De ahí, que los programas se comportan de forma totalmente diferente si hacemos un cambio de escala de algunos valores de la evaluación! Resulta que, para evitar esta sensibilidad, la función de evaluación debe ser una transformación *lineal positiva* de la probabilidad de ganancia desde una posición (o, más generalmente, de la utilidad esperada de la posición). Esto es una propiedad importante y general de situaciones en las que está implicada la incertidumbre, y hablaremos de ello en el Capítulo 16.

Complejidad del minimaxesperado

Si el programa supiera de antemano todos los resultados de las tiradas que ocurrirían para el resto del juego, resolver un juego con dados sería como resolver un juego sin dados, en el que minimax lo hace en $O(b^m)$ veces. Como minimaxesperado considera también todas las secuencias de las tiradas posibles, tendrá $O(b^m n^m)$, donde n es el número de resultados distintos.

Incluso si la profundidad de búsqueda se limita a una pequeña profundidad d , el coste adicional, comparado con el de minimax, lo hace poco realista para considerar anticiparse muy lejos en la mayoría de los juegos de azar. En *backgammon* n es 21 y b está por lo general alrededor de 20, pero en algunas situaciones podemos llegar a 4.000 para los resultados dobles. Tres capas serían, probablemente, todo lo que podríamos manejar.

Otro modo de ver el problema es este: la ventaja de alfa-beta consiste en ignorar los progresos futuros que apenas van a suceder, dado el mejor juego. Así, se concentra en acontecimientos probables. En juegos con dados, *no* hay secuencias probables de movimientos, porque para que ocurran esos movimientos, los dados tendrían que salir primero del modo correcto para hacerlos legales. Este es un problema general siempre que aparece la incertidumbre: las posibilidades se multiplican enormemente, y la formación



de proyectos detallados de acciones se hacen inútiles, porque el mundo probablemente no jugará así.

Sin duda esto le habrá ocurrido al lector que haya aplicado algo como la poda alfa-beta a árboles de juegos con nodos posibilidad. Resulta que esto podría ocurrir. El análisis para los nodos MIN y MAX no se altera, pero sí podemos podar también los nodos de posibilidades, usando un poco de ingenio. Consideremos el nodo de posibilidad C de la Figura 6.11 y lo que le pasa a su valor cuando examinamos y evaluamos a sus hijos. ¿Es posible encontrar un límite superior sobre el valor de C antes de que hayamos mirado a todos sus hijos? (Recuerde que esto es lo que alfa-beta necesita para podar un nodo y su subárbol.) A primera vista, podría parecer imposible, porque el valor de C es el *promedio* de los valores de sus hijos. Hasta que no hayamos visto todos los resultados de las tiradas, este promedio podría ser cualquier cosa, porque los hijos no examinados podrían tener cualquier valor. Pero si ponemos límites sobre los valores posibles de la función de utilidad, entonces podemos llegar a poner límites para el promedio. Por ejemplo, si decimos que todos los valores de utilidad están entre $+3$ y -3 , entonces el valor de los nodos hoja está acotado, y por su parte *podemos* colocar una cota superior sobre el valor de un nodo de posibilidad sin ver a todos sus hijos.

Juegos de cartas

Los juegos de cartas son interesantes por muchos motivos además de su conexión con los juegos de azar. Entre la enorme variedad de juegos, nos centraremos en aquellos en los cuales las cartas se reparten al azar al principio del juego, y cada jugador recibe un conjunto de cartas que no son visibles a los otros jugadores. Tales juegos incluyen bridge, *whist*, corazones y algunas formas del póker.

A primera vista, podría parecer que los juegos de cartas son como juegos de dados: ¡las cartas se reparten al azar y determinan los movimientos disponibles para cada jugador, como si todos los dados fueran lanzados al principio! Perseguiremos esta observación más tarde. Resultará ser bastante útil en la práctica. También se equivoca bastante, por motivos interesantes.

Imaginemos dos jugadores, MAX y MIN, jugando algunas manos de bridge con cuatro cartas visibles. Las manos son, donde MAX juega primero:

MAX: $\heartsuit 6 \diamond 6 \clubsuit 9 8$ MIN: $\heartsuit 4 \spadesuit 2 \clubsuit 10 5$

Suponga que MAX sale con $\clubsuit 9$, MIN debe seguir ahora el palo, jugando el $\clubsuit 10$ o el $\clubsuit 5$. MIN juega el $\clubsuit 10$ y gana la mano. MIN va después y sale con $\spadesuit 2$. MAX no tiene picas (y así no puede ganar la mano) y por lo tanto debe tirar alguna carta. La opción obvia es $\diamond 6$ porque las otras dos cartas restantes son ganadoras. Ahora, cualquier carta que MIN saque para la siguiente mano, MAX ganará (ganará ambas manos) y el juego será de empate con dos manos cada uno. Es fácil mostrar, usando una variante conveniente de minimax (Ejercicio 6.12), que la salida de MAX con el $\clubsuit 9$ es de hecho una opción óptima.

Ahora vamos a modificar la mano de MIN, sustituyendo los $\heartsuit 4$ con los $\diamond 4$:

MAX: $\heartsuit 6 \diamond 6 \clubsuit 9 8$ MIN: $\heartsuit 4 \spadesuit 2 \clubsuit 10 5$

Los dos casos son completamente simétricos: el juego será idéntico, salvo que en la segunda mano MAX tirará el $\heartsuit 6$. Otra vez, el juego será de empate a dos manos cada uno y la salida con $\clubsuit 9$ es una opción óptima.

Hasta ahora, bien. Ahora vamos a esconder una de las cartas de MIN: MAX sabe que MIN tiene lo de la primera mano (con el $\heartsuit 4$) o lo de la segunda (con el $\diamondsuit 4$), pero no sabe cuál tiene realmente. MAX razona como sigue:

El $\clubsuit 9$ es una opción óptima contra la primera mano de MIN y contra la segunda mano de MIN, entonces debe ser óptima ahora porque sé que MIN tiene una de las dos manos.

Más generalmente, MAX usa lo que podríamos llamar «hacer un promedio sobre la clarividencia». La idea es evaluar una línea de acción, cuando hay cartas no visibles, calculando primero el valor de minimax de esa acción para cada reparto posible de cartas, y luego calcular el valor esperado usando la probabilidad de cada reparto.

Si piensa que esto es razonable (o si no tiene ni idea porque no entiende el bridge), considere la siguiente historia:

Día 1: el Camino A conduce a un montón de piezas de oro; el camino B conduce a una bifurcación. Tome hacia la izquierda y encontrará un montón de joyas, pero tome hacia la derecha y será atropellado por un autobús.

Día 2: el Camino A conduce a un montón de piezas de oro; el camino B conduce a una bifurcación. Tome hacia la derecha y encontrará un montón de joyas, pero tome hacia la izquierda y será atropellado por un autobús.

Día 3: el camino A conduce a un montón de piezas de oro; el camino B conduce a una bifurcación. Adivine correctamente y encontrará un montón de joyas, pero adivine incorrectamente y será atropellado por un autobús.

Obviamente, no es irrazonable tomar el camino B durante los dos primeros días. Ninguna persona en su sano juicio, sin embargo, tomaría el camino B durante el Día 3. Esto es exactamente lo que sugiere un promedio sobre la clarividencia: el camino B es óptimo en las situaciones de Día 1 y Día 2; por lo tanto es óptimo durante el Día 3, porque una de las dos situaciones anteriores debe conseguirse. Volvamos al juego de cartas: después de que MAX salga con el $\clubsuit 9$, MIN gana con el $\clubsuit 10$. Como antes, MIN sale con el $\spadesuit 2$, y ahora MAX está como en la bifurcación del camino sin ninguna instrucción. Si MAX tira el $\heartsuit 6$ y MIN todavía tiene $\heartsuit 4$, el $\heartsuit 4$ se hace ganador y MAX pierde el juego. Del mismo modo, si MAX tira $\diamondsuit 6$ y MIN todavía tiene el $\diamondsuit 4$, MAX también pierde. Por lo tanto, jugar primero el $\clubsuit 9$ conduce a una situación donde MAX tiene una posibilidad del 50 por ciento de perder. (Sería mucho mejor jugar primero el $\heartsuit 6$ y el $\diamondsuit 6$, garantizando un empate.)

La lección que podemos sacar de todo esto es que cuando falla la información, hay que considerar *la información que se tendrá* en cada punto del juego. El problema con el algoritmo de MAX es que asume que en cada posible reparto, el juego procederá *como si todas las cartas fueran visibles*. Como muestra nuestro ejemplo, esto conduce a MAX a actuar como si toda la incertidumbre *futura* vaya a ser resuelta cuando aparezca. El algoritmo de MAX decidirá también no reunir la información (ni *proporcionar* información a un compañero), porque en cada reparto no hay ninguna necesidad de hacerlo; sin embargo en juegos como el bridge, es a menudo una buena idea jugar una carta que ayudará a descubrir cosas sobre las cartas del adversario o informar a su compañero sobre

sus propias cartas. Estas clases de comportamientos se generan automáticamente por un algoritmo óptimo para juegos de la información imperfecta. Tal algoritmo no busca en el espacio de estados del mundo (las manos de las cartas), sino en el espacio de **estados de creencia** (creencia de quién tiene qué cartas, con qué probabilidades). Seremos capaces de explicar el algoritmo correctamente en el Capítulo 17, una vez que hayamos desarrollado la maquinaria probabilística necesaria. En ese capítulo, ampliaremos también un punto final y muy importante: en juegos de información imperfecta, lo mejor es dar tan poca información al oponente como sea posible, y a menudo el mejor modo de hacerlo es actuar de manera *impredecible*. Por eso, los inspectores de sanidad hacen visitas de inspección aleatorias a los restaurantes.

6.6 Programas de juegos

Podríamos decir que jugar a juegos es a IA como el Gran Premio de carreras de automóviles es a la industria de coches: los programas de juegos son deslumbrantemente rápidos, máquinas increíblemente bien ajustadas que incorporan técnicas muy avanzadas de la ingeniería, pero no son de mucho uso para hacer la compra. Aunque algunos investigadores crean que jugar a juegos es algo irrelevante en la corriente principal de IA, se sigue generando entusiasmo y una corriente estable de innovaciones que se han adoptado por la comunidad.

AJEDREZ

Ajedrez: en 1957, Herbert Simon predijo que dentro de 10 años los computadores ganarían al campeón mundial humano. Cuarenta años más tarde, el programa Deep Blue derrotó a Garry Kasparov en un partido de exhibición a seis juegos. Simon se equivocó, pero sólo por un factor de 4. Kasparov escribió:

El juego decisivo del partido fue el juego 2, que dejó una huella en mi memoria... Vimos algo que fue más allá de nuestras expectativas más salvajes de cómo un computador sería capaz de prever las consecuencias posicionales a largo plazo de sus decisiones. La máquina rechazó moverse a una posición que tenía una ventaja decisiva a corto plazo, mostrando un sentido muy humano del peligro. (Kasparov, 1997)

Deep Blue fue desarrollado por Murray Campbell, Feng-Hsiung Hsu, y Joseph Hoane en IBM (véase Campbell *et al.*, 2002), construido sobre el diseño de Deep Thought desarrollado anteriormente por Campbell y Hsu en Carnegie Mellon. La máquina ganadora era un computador paralelo con 30 procesadores IBM RS/6000 que controlaba la «búsqueda software» y 480 procesadores VLSI, encargados para el ajedrez, que realizaron la generación de movimientos (incluso el movimiento de ordenación), la «búsqueda hardware» para los últimos niveles del árbol, y la evaluación de los nodos hoja. Deep Blue buscó 126 millones de nodos por segundo, como regla general, con una velocidad máxima de 330 millones de nodos por segundo. Generó hasta 30 billones de posiciones por movimiento, y alcanzó la profundidad 14 rutinariamente. El corazón de la máquina es una búsqueda alfa-beta estándar de profundidad iterativa con una tabla de transposiciones, pero la llave de su éxito parece haber estado en su capacidad de generar extensiones más allá del límite de profundidad para líneas suficientemente interesantes. En

algunos casos la búsqueda alcanzó una profundidad de 40 capas. La función de evaluación tenía más de 8.000 características, muchas de ellas describiendo modelos muy específicos de las piezas. Se usó una «salida de libro» de aproximadamente 4.000 posiciones, así como una base de datos de 700.000 jugadas de gran maestro de las cuales se podrían extraer algunas recomendaciones de consenso. El sistema también utilizó una gran base de datos de finales del juego de posiciones resueltas, conteniendo todas las posiciones con cinco piezas y muchas con seis piezas. Esta base de datos tiene el considerable efecto de ampliar la profundidad efectiva de búsqueda, permitiendo a Deep Blue jugar perfectamente en algunos casos aun cuando se aleja del jaque mate.

El éxito de Deep Blue reforzó la creencia de que el progreso en los juegos de computador ha venido principalmente del *hardware* cada vez más poderoso (animado por IBM). Los creadores de Deep Blue, por otra parte, declaran que las extensiones de búsqueda y la función de evaluación eran también críticas (Campbell *et al.*, 2002). Además, sabemos que mejoras algorítmicas recientes han permitido a programas, que se ejecutan sobre computadores personales, ganar cada Campeonato Mundial de Ajedrez de computadores desde 1992, a menudo derrotando a adversarios masivamente paralelos que podrían buscar 1.000 veces más nodos. Una variedad de poda heurística se utiliza para reducir el factor de ramificación efectivo a menos de 3 (comparado con el factor de ramificación actual de aproximadamente 35). Lo más importante de ésta es la heurística de **movimiento nulo**, que genera una cota inferior buena sobre el valor de una posición, usando una búsqueda superficial en la cual el adversario consigue moverse dos veces. Esta cota inferior a menudo permite la poda alfa-beta sin el costo de una búsqueda de profundidad completa. También es importante la **poda de inutilidad**, la cual ayuda ha decidir de antemano qué movimientos causarán un corte beta en los nodos sucesores.

El equipo de Deep Blue rehusó la posibilidad de un nuevo partido con Kasparov. En cambio, la competición principal más reciente de 2002 destacó el programa FRITZ contra el campeón mundial Vladimir Kramnik. El partido a ocho juegos terminó en un empate. Las condiciones del partido eran mucho más favorables al humano, y el *hardware* era un computador personal ordinario, no un supercomputador. De todos modos, Kramnik comentó que «está ahora claro que el programa y el campeón mundial son aproximadamente iguales».

MOVIMIENTO NULO

PODA DE INUTILIDAD

DAMAS

Damas: en 1952, Arthur Samuel de IBM, trabajando en sus ratos libres, desarrolló un programa de damas que aprendió su propia función de evaluación jugando con él mismo miles de veces. Describimos esta idea más detalladamente en el Capítulo 21. El programa de Samuel comenzó como un principiante, pero después, sólo unos días de auto-jugar, se había mejorado más allá del propio nivel de Samuel (aunque él no fuera un jugador fuerte). En 1962 derrotó a Robert Nealy, un campeón en «damas ciegas», por un error por su parte. Muchas personas señalaron que, en damas, los computadores eran superiores a la gente, pero no era la cuestión. De todos modos, cuando uno considera que el equipo calculador de Samuel (un IBM 704) tenía 10.000 palabras de memoria principal, cinta magnetofónica para el almacenaje a largo plazo y un procesador de ,000001 GHz, el triunfo sigue siendo un gran logro.

Pocas personas intentaron hacerlo mejor hasta que Jonathan Schaeffer y colegas desarrollaran Chinook, que se ejecuta sobre computadores personales y usa la búsqueda alfa-

beta. Chinook usa una base de datos precalculada de 444 billones de posiciones con ocho o menos piezas sobre el tablero para así hacer la fase final del juego de forma impecable. Chinook quedó segundo en el Abierto de Estados Unidos de 1990 y ganó el derecho a participar en el campeonato mundial. Entonces se ejecutó contra un problema, en la forma de Marion Tinsley. Dr. Tinsley había sido el campeón mundial durante más de 40 años, y había perdido sólo tres juegos en todo ese tiempo. En el primer partido contra Chinook, Tinsley sufrió sus cuartas y quintas derrotas, pero ganó el partido 20.5-18.5. El partido del campeonato mundial en agosto de 1994 entre Tinsley y Chinook se terminó prematuramente cuando Tinsley tuvo que retirarse por motivos de salud. Chinook se convirtió en el campeón mundial oficial.

Schaeffer cree que, con bastante poder calculador, la base de datos de fases finales podría ampliarse hasta el punto donde una búsqueda hacia delante desde la posición inicial alcanzaría siempre posiciones resueltas, es decir, las damas estarían completamente resueltas. (Chinook ha anunciado un triunfo tan sólo en cinco movimientos.) Esta clase de análisis exhaustivo puede hacerse a mano para tic-tac-toe 3x3 y se ha hecho con el computador para Qubic (tic-tac-toe 4x4x4), Go-Moku (cinco en fila), y Morris de nueve-hombres (Gasser, 1998). El trabajo notable de Ken Thompson y Lewis Stiller (1992) resolvió todo el ajedrez con cinco piezas y las fases finales de seis piezas, poniéndolas a disposición sobre Internet. Stiller descubrió un caso donde existía un jaque mate forzado, pero requirió 262 movimientos; esto causó alguna consternación porque las reglas del ajedrez requieren que ocurra algún «progreso» en 50 movimientos.

OTELO

Oteló: también llamado Reversi, es probablemente más popular como un juego de computador que como un juego de mesa. Tiene un espacio de búsqueda más pequeño que el ajedrez, por lo general de cinco a 15 movimientos legales, pero desde los comienzos se tuvo que desarrollar la evaluación experta. En 1997, el programa Logistello (Buro, 2002) derrotó al campeón mundial humano, Takeshi Murakami, por seis juegos a ninguno. Se reconoce, generalmente, que la gente no es igual a los computadores en Oteló.

BACKGAMMON

Backgammon: la Sección 6.5 explicó por qué la inclusión de incertidumbre, provocada por el lanzamiento de los dados, hace de la búsqueda un lujo costoso. La mayor parte de los trabajos sobre *backgammon* se han centrado en la mejora de la función de evaluación. Gerry Tesauro (1992) combinó el aprendizaje por refuerzo de Samuel con técnicas de redes neuronales (Capítulo 20) para desarrollar un evaluador notablemente exacto usado con una búsqueda a profundidad 2 o 3. Después de jugar más de un millón de juegos de entrenamiento contra él mismo, el programa de Tesauro, TD-GAMMON, se sitúa, seguramente, entre los tres primeros jugadores del mundo. Las opiniones del programa sobre los movimientos de apertura del juego han alterado radicalmente la sabiduría recibida.

GO

Go: es el juego de mesa más popular de Asia, requiriendo al menos tanta disciplina de sus profesionales como el ajedrez. Como el tablero es de 19x19, el factor de ramificación comienza en 361, que desalienta también a los métodos regulares de búsqueda. Hasta 1997 no había ningún programa competente, pero ahora los programas a menudo juegan respetablemente. La mayor parte de los mejores programas combinan técnicas de reconocimiento de modelos (cuando aparece el siguiente modelo de piezas, este

movimiento debe considerarse) con la búsqueda limitada (decide si estas piezas pueden ser capturadas, y quedan dentro del área local). Los programas más fuertes, en el momento de estar escribiendo, son probablemente Goemate de Chen Zhixing y Go4++ de Michael Reiss, cada uno valorado en alrededor de 10 kyu (aficionado débil). Go es un área que probablemente se beneficiará de la investigación intensiva que utiliza métodos de razonamiento más sofisticados. El éxito puede venir de encontrar modos de integrar varias líneas del razonamiento local sobre cada uno de los muchos «subjuegos» ligeramente conectados en los que Go se puede descomponer. Tales técnicas serían de enorme valor para sistemas inteligentes en general.

BRIDGE

Bridge: es un juego de información imperfecta: las cartas de un jugador se esconden de los otros jugadores. El bridge es también un juego *multijugador* con cuatro jugadores en vez de dos, aunque los jugadores se emparejen en dos equipos. Cuando lo vimos en la Sección 6.5, el juego óptimo en el bridge puede incluir elementos de reunión de información, comunicación, tirarse un farol, y el ponderado cuidadoso de probabilidades. Muchas de estas técnicas se usan en el programa de Bridge Baron™ (Smith *et al.*, 1998), que ganó el campeonato de bridge de computadores de 1997. Mientras no juega óptimamente, Bridge Baron es uno de los pocos sistemas de juegos en usar planes complejos y jerárquicos (*véase* el Capítulo 12) que implican ideas de alto nivel como **astucia y aprieto**, que son familiares para los jugadores de bridge.

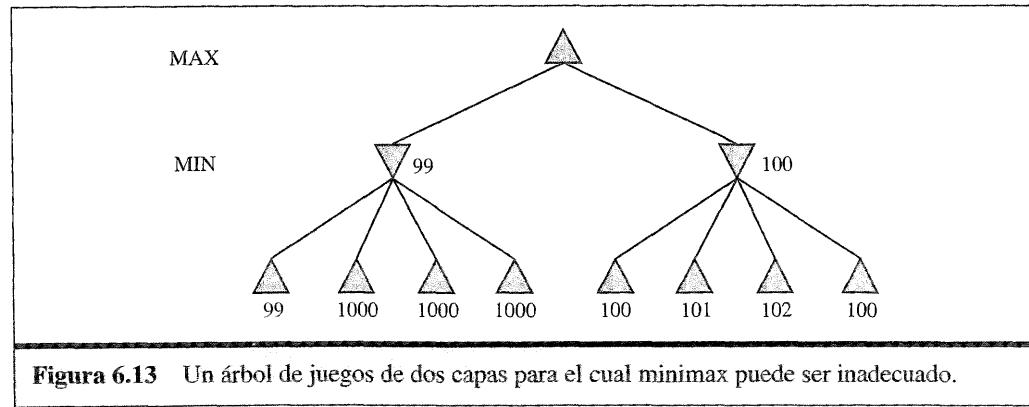
El programa GIB (Ginsberg, 1999) ganó el campeonato 2000 con decisión. GIB usa el método de «hacer un promedio sobre la clarividencia», con dos modificaciones cruciales. Primero, antes de examinar cuán bien trabaja cada opción para cada plan posible de las cartas escondidas, (de las cuales puede ser hasta 10 millones) examina una muestra aleatoria de 100 planes. Segundo, GIB usa la **generalización basada en explicaciones** para calcular y guardar las reglas generales para el juego óptimo en varias clases estándar de situaciones. Esto permite resolver cada reparto *exactamente*. La exactitud táctica del GIB compensa su inhabilidad de razonar sobre la información. Terminó el 12º de 35 en la competición de igualdad (implicando solamente el juego de una mano) en el campeonato mundial humano de 1998, excediendo las expectativas de muchos expertos humanos.

6.7 Discusión

Como el cálculo de decisiones óptimas en juegos es intratable en la mayoría de los casos, todos los algoritmos deben hacer algunas suposiciones y aproximaciones. La aproximación estándar, basada en minimax, funciones de evaluación y alfa-beta, es solamente un modo de hacerlo. Probablemente porque la propusieron tan pronto, la aproximación estándar fue desarrollada intensivamente y domina otros métodos en los juegos de turnos. Algunos en el campo creen que esto ha causado que jugar a juegos llegue a divorciarse de la corriente principal de investigación de IA, porque la aproximación estándar no proporciona mucho más espacio para nuevas perspicacias en cuestiones generales de la toma de decisiones. En esta sección, vemos las alternativas.

Primero, consideremos minimax. Minimax selecciona un movimiento óptimo en un árbol de búsqueda *a condición de que las evaluaciones de los nodos hoja sean exactamente correctas*. En realidad, las evaluaciones son generalmente estimaciones rudimentarias del valor de una posición y se consideran que tienen asociados errores grandes. La Figura 6.13 muestra un árbol de juegos de dos capas para el cual minimax parece inadecuado. Minimax aconseja tomar la rama derecha, mientras que es bastante probable que el valor real de la rama izquierda sea más alto. La opción de minimax confía suponiendo que *todos* los nodos etiquetados con valores 100, 101, 102 y 100 sean *realmente* mejores que el nodo etiquetado con el valor 99. Sin embargo, el hecho de que el nodo etiquetado con 99 tiene hermanos etiquetados con 1.000 sugiere que, de hecho, podría tener un valor real más alto. Un modo de tratar con este problema es tener una evaluación que devuelva una *distribución de probabilidad* sobre valores posibles. Entonces uno puede calcular la distribución de probabilidad del valor del padre usando técnicas estadísticas. Lamentablemente, los valores de los nodos hermanos están, por lo general, muy correlacionados, por consiguiente puede ser de cálculo costoso, requisito importante para obtener la información.

Después, consideraremos el algoritmo de búsqueda que genera el árbol. El objetivo de un diseñador de algoritmos es especificar un cálculo de ejecución rápida y que produzca un movimiento bueno. El problema más obvio con el algoritmo alfa-beta es que está diseñado, no solamente para seleccionar un movimiento bueno, sino también para calcular límites sobre los valores de todos los movimientos legales. Para ver por qué esta información suplementaria es innecesaria, consideremos una posición en la cual hay sólo un movimiento legal. La búsqueda alfa-beta todavía generará y evaluará un grande, y totalmente inútil, árbol de búsqueda. Desde luego, podemos insertar un test en el algoritmo, pero éste simplemente esconderá el problema subyacente: muchos de los cálculos hechos por alfa-beta son en gran parte irrelevantes. Tener sólo un movimiento legal no es mucho más diferente que tener varios movimientos legales, uno de los cuales es excelente y el resto obviamente desastroso. En una situación «favorita clara» como ésta, sería mejor alcanzar una decisión rápida después de una pequeña cantidad de búsqueda, que gastar el tiempo que podría ser más provechoso más tarde en una posición más problemática. Esto conduce a la idea de la *utilidad de una expansión de un nodo*. Un algoritmo de búsqueda bueno debería seleccionar expansiones de nodos de utilidad alta



(es decir que probablemente conducirán al descubrimiento de un movimiento considerablemente mejor). Si no hay ninguna expansión de nodos cuya utilidad sea más alta que su coste (en términos de tiempo), entonces el algoritmo debería dejar de buscar y hacer un movimiento. Notemos que esto funciona no solamente para situaciones favoritas claras, sino también para el caso de movimientos *simétricos*, para los cuales ninguna cantidad de la búsqueda mostrará que un movimiento es mejor que otro.

A esta clase de razonamiento, sobre qué cálculos hacer, se le llama **meta-razonamiento** (razonamiento sobre el razonamiento). Esto se aplica no solamente a juegos, sino a cualquier clase del razonamiento. Todos los cálculos se hacen para tratar de alcanzar mejores decisiones, todos tienen gastos, y todos tienen alguna probabilidad de causar una cierta mejora de la calidad de decisión. Alfa-beta incorpora la clase más simple de meta-razonamiento, un teorema en el sentido de que ciertas ramas del árbol pueden ignorarse sin perder. Es posible hacerlo mucho mejor. En el Capítulo 16, veremos cómo estas ideas pueden hacerse más precisas e implementables.

Finalmente, reexaminemos la naturaleza de la búsqueda en sí mismo. Algoritmos para la búsqueda heurística y para juegos trabajan generando secuencias de estados concretos, comenzando desde el estado inicial y luego aplicando una función de evaluación. Claramente, así no es como juega la gente. En el ajedrez, uno a menudo tiene en mente un objetivo particular (por ejemplo, atrapar la reina del adversario) y puede usar este objetivo para generar *selectivamente* el plan plausible para conseguirlo. Esta clase de **razonamiento dirigido por objetivos** o **planificación** a veces elimina, totalmente, la búsqueda combinatoria. (Véase la Parte IV.) PARADISE de David Wilkins (1980) es el único programa que ha usado el razonamiento dirigido por objetivos con éxito en el ajedrez: era capaz de resolver algunos problemas de ajedrez que requieren una combinación de 18 movimientos. Aún no es nada fácil entender cómo *combinar* las dos clases de algoritmos en un sistema robusto y eficiente, aunque Bridge Baron pudiera ser un paso en la dirección correcta. Un sistema totalmente integrado sería un logro significativo, no solamente para la investigación de juegos, sino también para la investigación de IA en general, porque esto sería una buena base para un agente general inteligente.

META-RAZONAMIENTO

6.8 Resumen

Hemos visto una variedad de juegos para entender qué significa jugar óptimamente y entender cómo jugar bien en la práctica. Las ideas más importantes son las siguientes:

- Un juego puede definirse por el **estado inicial** (como se establece en el tablero), las **acciones legales** en cada estado, un **test terminal** (que dice cuándo el juego está terminado), y una **función de utilidad** que se aplica a los estados terminales.
- En juegos de suma cero de dos jugadores con **información perfecta**, el algoritmo **minimax** puede seleccionar movimientos óptimos usando una enumeración primero en profundidad del árbol de juegos.
- El algoritmo de búsqueda **alfa-beta** calcula el mismo movimiento óptimo que el minimax, pero consigue una eficiencia mucho mayor, eliminando subárboles que son probablemente irrelevantes.

- Por lo general, no es factible considerar el árbol entero de juegos (hasta con alfa-beta), entonces tenemos que cortar la búsqueda en algún punto y aplicar una función de evaluación que dé una estimación de la utilidad de un estado.
- Los juegos de azar pueden manejarse con una extensión del algoritmo minimax que evalúa un nodo de posibilidad tomando la utilidad media de todos sus nodos hijos, ponderados por la probabilidad de cada hijo.
- El juego óptimo en juegos de información imperfecta, como el bridge, requiere el razonamiento sobre los estados de creencia actuales y futuros de cada jugador. Una aproximación simple puede obtenerse haciendo un promedio del valor de una acción sobre cada configuración posible de la información ausente.
- Los programas pueden equipararse o pueden ganar a los mejores jugadores humanos en damas, Otelo y *backgammon*, y están cercanos en bridge. Un programa ha ganado al campeón mundial de ajedrez en un partido de exhibición. Los programas permanecen en el nivel aficionado en Go.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

La temprana historia de los juegos mecánicos se estropeó por numerosos fraudes. El más célebre de estos fue «El Turco» de Baron Wolfgang von Kempelen (1734-1804), un supuesto autómata que jugaba al ajedrez, que derrotó a Napoleón antes de ser expuesto como la caja de bromas de un mago que escondía a un humano experto en ajedrez (véase Levitt, 2000). Jugó desde 1769 hasta 1854. En 1846, Charles Babbage (quien había sido fascinado por el Turco) parece haber contribuido a la primera discusión seria de la viabilidad del computador de ajedrez y de damas (Morrison y Morrison, 1961). Él también diseñó, pero no construyó, una máquina con destino especial para jugar a tic-tac-toe. La primera máquina real de juegos fue construida alrededor de 1890 por el ingeniero español Leonardo Torres y Quevedo. Se especializó en el «RTR» (rey y torre contra el rey), la fase final de ajedrez, garantizando un triunfo con el rey y torre desde cualquier posición.

El algoritmo minimax se remonta a un trabajo publicado en 1912 de Ernst Zermelo, el que desarrolló la teoría moderna de conjuntos. El trabajo, lamentablemente, tenía varios errores y no describió minimax correctamente. Un fundamento sólido, para la teoría de juegos, fue desarrollado en el trabajo seminal de *Theory of Games and Economic Behaviour* (von Neumann y Morgenstern, 1944), que incluyó un análisis en el que mostraba que algunos juegos *requieren* estrategias aleatorizadas (o imprevisibles). Véase el Capítulo 17 para más información.

Muchas figuras influyentes de los comienzos de los computadores, quedaron intrigadas por la posibilidad de jugar al ajedrez con un computador. Konrad Zuse (1945), la primera persona que diseñó un computador programable, desarrolló ideas bastante detalladas sobre cómo se podría hacer esto. El libro influyente de Norbert Wiener (1948), *Cybernetics*, habló de un diseño posible para un programa de ajedrez, incluso las ideas de búsqueda minimax, límites de profundidad, y funciones de evaluación. Claude Shannon (1950) presentó los principios básicos de programas modernos de juegos con mucho más detalle que Wiener. Él introdujo la idea de la búsqueda de estabilidad y describió algunas ideas para la búsqueda del árbol de juegos selectiva (no exhaustiva). Slater

(1950) y los que comentaron su artículo también exploraron las posibilidades para el juego de ajedrez por computador. En particular, I. J. Good (1950) desarrolló la noción de estabilidad independientemente de Shannon.

En 1951, Alan Turing escribió el primer programa de computador capaz de jugar un juego completo de ajedrez (véase Turing *et al.*, 1953). Pero el programa de Turing nunca se ejecutó sobre un computador; fue probado por simulación a mano contra un jugador muy débil humano, que lo derrotó. Mientras tanto D. G. Prinz (1952) escribió, y realmente ejecutó, un programa que resolvió problemas de ajedrez, aunque no jugara un juego completo. Alex Bernstein escribió el primer programa para jugar un juego completo de ajedrez estándar (Bernstein y Roberts, 1958; Bernstein *et al.*, 1958)³.

John McCarthy concibió la idea de la búsqueda alfa-beta en 1956, aunque él no lo publicara. El programa NSS de ajedrez (Newell *et al.*, 1958) usó una versión simplificada de alfa-beta; y fue el primer programa de ajedrez en hacerlo así. Según Nilsson (1971), el programa de damas de Arthur Samuel (Samuel, 1959, 1967) también usó alfa-beta, aunque Samuel no lo mencionara en los informes publicados sobre el sistema. Los trabajos que describen alfa-beta fueron publicados a principios de 1960 (Hart y Edwards, 1961; Brudno, 1963; Slagle, 1963b). Una implementación completa de alfa-beta está descrita por Slagle y Dixon (1969) en un programa para juegos de Kalah. Alfa-beta fue también utilizada por el programa «Kotok-McCarthy» de ajedrez escrito por un estudiante de John McCarthy (Kotok, 1962). Knuth y Moore (1975) proporcionan una historia de alfa-beta, junto con una demostración de su exactitud y un análisis de complejidad en tiempo. Su análisis de alfa-beta con un orden de sucesores aleatorio mostró una complejidad asintótica de $O((b/\log b)^d)$, que pareció bastante triste porque el factor de ramificación efectivo $b/\log b$ no es mucho menor que b . Ellos, entonces, se dieron cuenta que la fórmula asintótica es exacta sólo para $b > 1000$ más o menos, mientras que a menudo se aplica un $O(b^{3d/4})$ a la variedad de factores de ramificación encontrados en los juegos actuales. Pearl (1982b) muestra que alfa-beta es asintóticamente óptima entre todos los algoritmos de búsqueda de árbol de juegos de profundidad fija.

El primer partido de ajedrez de computador presentó al programa Kotok-McCarthy y al programa «ITEP» escrito a mediados de 1960 en el Instituto de Moscú de Física Teórica y Experimental (Adelson-Velsky *et al.*, 1970). Este partido intercontinental fue jugado por telégrafo. Se terminó con una victoria 3-1 para el programa ITEP en 1967. El primer programa de ajedrez que compitió con éxito con la gente fue MacHack 6 (Greenblatt *et al.*, 1967). Su grado de aproximadamente 1.400 estaba bien sobre el nivel de principiante de 1.000, pero era bajo comparado con el grado 2.800 o más que habría sido necesario para satisfacer la predicción de 1957 de Herb Simon de que un programa de computador sería el campeón mundial de ajedrez en el plazo de 10 años (Simon y Newell, 1958).

Comenzando con el primer Campeonato Norteamericano ACM de Ajedrez de computador en 1970, el concurso entre programas de ajedrez se hizo serio. Los programas a principios de 1970 se hicieron sumamente complicados, con varias clases de trucos para eliminar algunas ramas de búsqueda, para generar movimientos plausibles, etcétera.

³ Newell *et al.* (1958) mencionan un programa ruso, BESM, que puede haber precedido al programa de Bernstein.

En 1974, el primer Campeonato Mundial de Ajedrez de computador fue celebrado en Estocolmo y ganado por Kaissa (Adelson-Velsky *et al.*, 1975), otro programa de ITEP. Kaissa utilizó la aproximación mucho más directa de la búsqueda alfa-beta exhaustiva combinada con la búsqueda de estabilidad. El dominio de esta aproximación fue confirmado por la victoria convincente de CHESS 4.6 en el Campeonato Mundial de 1977 de Ajedrez de computador. CHESS 4.6 examinó hasta 400.000 posiciones por movimiento y tenía un grado de 1.900.

Una versión posterior de MacHack, de Greenblatt 6, fue el primer programa de ajedrez ejecutado sobre un *hardware* de encargo diseñado expresamente para el ajedrez (Moussouris *et al.*, 1979), pero el primer programa en conseguir éxito notable por el uso del *hardware* de encargo fue Belle (Condon y Thompson, 1982). El *hardware* de generación de movimientos y de la evaluación de la posición de Belle, le permitió explorar varios millones de posiciones por movimiento. Belle consiguió un grado de 2.250, y se hizo el primer programa de nivel maestro. El sistema HITECH, también un computador con propósito especial, fue diseñado por el antiguo Campeón de Ajedrez de Correspondencia Mundial Hans Berliner y su estudiante Carl Ebeling en CMU para permitir el cálculo rápido de la función de evaluación (Ebeling, 1987; Berliner y Ebeling, 1989). Generando aproximadamente 10 millones de posiciones por movimiento, HITECH se hizo el campeón norteamericano de computador en 1985 y fue el primer programa en derrotar a un gran maestro humano en 1987. Deep Thought, que fue también desarrollado en CMU, fue más lejos en la dirección de la velocidad pura de búsqueda (Hsu *et al.*, 1990). Consiguió un grado de 2.551 y fue el precursor de Deep Blue. El Premio de Fredkin, establecido en 1980, ofreció 5.000 dólares al primer programa en conseguir un grado de maestro, 10.000 dólares al primer programa en conseguir un grado FEUA (Federación de los Estados Unidos de Ajedrez) de 2.500 (cerca del nivel de gran maestro), y 100.000 dólares para el primer programa en derrotar al campeón humano mundial. El premio de 5.000 dólares fue reclamado por Belle en 1983, el premio de 10.000 dólares por Deep Thought en 1989, y el premio de 100.000 dólares por Deep Blue por su victoria sobre Garry Kasparov en 1997. Es importante recordar que el éxito de Deep Blue fue debido a mejoras algorítmicas y de *hardware* (Hsu, 1999; Campbell *et al.*, 2002). Las técnicas como la heurística de movimiento nulo (Beal, 1990) han conducido a programas que son completamente selectivos en sus búsquedas. Los tres últimos Campeonatos Mundiales de Ajedrez de computador en 1992, 1995 y 1999 fueron ganados por programas que se ejecutan sobre computadores personales. Probablemente la mayor parte de la descripción completa de un programa moderno de ajedrez la proporciona Ernst Heinz (2000), cuyo programa DARKTHOUGHT fue el programa de computador no comercial de rango más alto de los campeonatos mundiales de 1999.

Varias tentativas se han hecho para vencer los problemas «de la aproximación estándar» perfilados en la Sección 6.7. El primer algoritmo selectivo de búsqueda con un poco de base teórica fue probablemente B* (Berlinés, 1979), que intenta mantener límites de intervalos sobre el valor posible de un nodo en el árbol de juegos, más que darle una estimación valorada por un punto. Los nodos hoja son seleccionados para expansión en una tentativa de refinar los límites del nivel superior hasta que un movimiento sea «claramente mejor». Palay (1985) amplía la idea de B* para usar distribuciones de probabilidad en lugar de intervalos. La búsqueda del número de conspiración

de David McAllester (1988) expande los nodos hoja que, cambiando sus valores, podrían hacer que el programa prefiriera un nuevo movimiento en la raíz. MGSS* (Russell y Wefald, 1989) usa las técnicas teóricas de decisión del Capítulo 16 para estimar el valor de expansión de cada hoja en términos de mejora esperada de la calidad de decisión en la raíz. Jugó mejor que un algoritmo alfa-beta, en Otelo, a pesar de la búsqueda de un orden de magnitud de menos nodos. La aproximación MGSS* es, en principio, aplicable al control de cualquier forma de deliberación.

La búsqueda alfa-beta es, desde muchos puntos de vista, el análogo de dos jugadores al ramificar y acotar primero en profundidad, dominada por A* en el caso de agente simple. El algoritmo SSS* (Stockman, 1979) puede verse como A* de dos jugadores y nunca expande más nodos que alfa-beta para alcanzar la misma decisión. Las exigencias de memoria y los costos indirectos computacionales de la cola hacen que SSS* sea poco práctico en su forma original, pero se ha desarrollado una versión de espacio-lineal a partir del algoritmo RBFS (Korf y Chickering, 1996). Plaat *et al.* (1996) desarrollaron una nueva visión de SSS* como una combinación de alfa-beta y tablas de transposiciones, mostrando cómo vencer los inconvenientes del algoritmo original y desarrollando una nueva variante llamada MTD(f) que ha sido adoptada por varios programas superiores.

D. F. Beal (1980) y Dana Nau (1980, 1983) estudiaron las debilidades de minimax aplicado a la aproximación de las evaluaciones. Ellos mostraron que bajo ciertos axiomas de independencia sobre las distribuciones de los valores de las hojas, minimaximizar puede producir valores en la raíz que son realmente *menos* fiables que el uso directo de la función de evaluación. El libro de Pearl, *Heuristics* (1984), explica parcialmente esta paradoja aparente y analiza muchos algoritmos de juegos. Baum y Smith (1997) proponen una sustitución a base de probabilidad para minimax, y muestra que ésto causa mejores opciones en ciertos juegos. Hay todavía poca teoría sobre los efectos de cortar la búsqueda en niveles diferentes y aplicar funciones de evaluación.

El algoritmo minimax esperado fue propuesto por Donald Michie (1966), aunque por supuesto sigue directamente los principios de evaluación de los árboles de juegos debido a von Neumann y Morgenstern. Bruce Ballard (1983) amplió la poda alfa-beta para cubrir árboles de nodos de posibilidad. El primer programa de *backgammon* fue BKG (Berliner, 1977, 1980b); utilizó una función de evaluación compleja construida a mano y buscó sólo a profundidad 1. Fue el primer programa que derrotó a un campeón mundial humano en un juego clásico importante (Berliner, 1980a). Berliner reconoció que éste fue un partido de exhibición muy corto (no fue un partido del campeonato mundial) y que BKG tuvo mucha suerte con los dados. El trabajo de Gerry Tesauro, primero sobre NEUROGAMMON (Tesauro, 1989) y más tarde sobre TD-GAMMON (Tesauro, 1995), mostró que se pueden obtener muchos mejores resultados mediante el aprendizaje por refuerzo, que trataremos en el Capítulo 21.

Las damas, más que el ajedrez, fue el primer juego clásico jugado completamente por un computador. Christopher Strachey (1952) escribió el primer programa de funcionamiento para las damas. Schaeffer (1997) dio una muy legible, «con todas sus imperfecciones», cuenta del desarrollo de su programa de damas campeón del mundo Chinook.

Los primeros programas de Go fueron desarrollados algo más tarde que los de las damas y el ajedrez (Lefkovitz, 1960; Remus, 1962) y han progresado más despacio. Ryder (1971) usó una aproximación basada en la búsqueda pura con una variedad de métodos

de poda selectivos para vencer el enorme factor de ramificación. Zobrist (1970) usó las reglas condición-acción para sugerir movimientos plausibles cuando aparecieran los modelos conocidos. Reitman y Wilcox (1979) combinan reglas y búsqueda con efectos buenos, y los programas más modernos han seguido esta aproximación híbrida. Müller (2002) resume el estado del arte de la informatización de Go y proporciona una riqueza de referencias. Anshelevich (2000) utilizó las técnicas relacionadas para el juego Hex. *Computer Go Newsletter*, publicada por la Asociación de Go por computador, describe el desarrollo actual del juego.

Los trabajos sobre juegos de computador aparecen en multitud de sitios. La mal llamada conferencia *Heuristic Programming in Artificial Intelligence* hizo un informe sobre las Olimpiadas de Computador, que incluyen una amplia variedad de juegos. Hay también varias colecciones de trabajos importantes sobre la investigación en juegos (Levy, 1988a, 1988b; Marsland y Schaeffer, 1990). La Asociación Internacional de Ajedrez por Computador (ICCA), fundada en 1977, publica la revista trimestral *ICGA* (anteriormente la revista *ICCA*). Los trabajos importantes han sido publicados en la serie antológica *Advances in Computer Chess*, que comienza con Clarke (1977). El volumen 134 de la revista *Artificial Intelligence* (2002) contiene descripciones de programas para el ajedrez, Otelo, Hex, shogi, Go, *backgammon*, poker, Scrabble™ y otros juegos.



EJERCICIOS

6.1 En este problema se ejercitan los conceptos básicos de juegos, utilizando tic-tac-toe (tres en raya) como un ejemplo. Definimos X_n como el número de filas, columnas, o diagonales con exactamente n Xs y ningún O. Del mismo modo, O_n es el número de filas, columnas, o diagonales con solamente n Os. La función de utilidad asigna +1 a cualquier posición con $X_3 = 1$ y -1 a cualquier posición con $O_3 = 1$. Todas las otras posiciones terminales tienen utilidad 0. Para posiciones no terminales, usamos una función de evaluación lineal definida como $\text{Eval}(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$.

- a) ¿Aproximadamente cuántos juegos posibles de tic-tac-toe hay?
- b) Muestre el árbol de juegos entero hasta profundidad 2 (es decir, un X y un O sobre el tablero) comenzando con un tablero vacío, teniendo en cuenta las simetrías.
- c) Señale sobre el árbol las evaluaciones de todas las posiciones a profundidad 2.
- d) Usando el algoritmo minimax, marque sobre su árbol los valores hacia atrás para las posiciones de profundidades 1 y 0, y use esos valores para elegir el mejor movimiento de salida.
- e) Marque los nodos a profundidad 2 que no serían evaluados si se aplicara la poda alfa-beta, asumiendo que los nodos están generados *en orden óptimo por la poda alfa-beta*.

6.2 Demuestre la afirmación siguiente: para cada árbol de juegos, la utilidad obtenida por MAX usando las decisiones minimax contra MIN subóptimo nunca será inferior que la utilidad obtenida jugando contra MIN óptimo. ¿Puede proponer un árbol de juegos en el cual MAX puede mejorar utilizando una estrategia subóptima contra MIN subóptimo?

6.3 Considere el juego de dos jugadores descrito en la Figura 6.14.

- Dibuje el árbol de juegos completo, usando las convenciones siguientes:
 - Escriba cada estado como (s_A, s_B) donde s_A y s_B denotan las posiciones simbólicas.
 - Ponga cada estado terminal en una caja cuadrada y escriba su valor de juego en un círculo.
 - Ponga los *estados bucle* (estados que ya aparecen sobre el camino a la raíz) en dobles cajas cuadradas. Ya que no está claro cómo adjudicar valores a estados bucle, anote cada uno con un «?» en un círculo.
- Ahora marque cada nodo con su valor minimax hacia atrás (también en un círculo). Explique cómo maneja los valores «?» y por qué.
- Explique por qué el algoritmo minimax estándar fallaría sobre este árbol de juegos y brevemente esboce cómo podría arreglarlo, usando su respuesta en (b). ¿Su algoritmo modificado proporciona las decisiones óptimas para todos los juegos con bucles?
- Este juego de 4 cuadrados puede generalizarse a n cuadrados para cualquier $n > 2$. Demuestre que A gana si n es par y pierde si n es impar.



6.4 Implemente los generadores de movimiento y las funciones de evaluación para uno o varios de los juegos siguientes: Kalah, Otelo, damas y ajedrez. Construya un agente de juegos alfa-beta general que use su implementación. Compare el efecto de incrementar la profundidad de la búsqueda, mejorando el orden de movimientos, y la mejora de la función de evaluación. ¿Cuál es el factor de ramificación eficaz para el caso ideal de la ordenación perfecta de movimientos?

6.5 Desarrolle una demostración formal de la exactitud para la poda alfa-beta. Para hacer esto, considere la situación de la Figura 6.15. La pregunta es si hay que podar el nodo n_j , qué es un nodo max y un descendiente del nodo n_1 . La idea básica es podarlo si y sólo si el valor minimax de n_1 puede demostrarse que es independiente del valor de n_j .

- El valor de n_1 está dado por

$$n_1 = \min(n_2, n_{21}, \dots, n_{2b_2})$$

Encuentre una expresión similar para n_2 y de ahí una expresión para n_1 en términos de n_j .

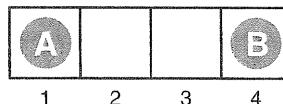


Figura 6.14 La posición de partida de un juego sencillo. El jugador A mueve primero. Los dos jugadores mueven por turno, y cada jugador debe mover su señal a un espacio vacío adyacente en *una u otra dirección*. Si el adversario ocupa un espacio adyacente, entonces un jugador puede saltar sobre el adversario al siguiente espacio vacío si existe. (Por ejemplo, si A está sobre 3 y B está sobre 2, entonces A puede mover hacia atrás a 1.) El juego se termina cuando un jugador alcanza el extremo opuesto del tablero. Si el jugador A alcanza el espacio 4 primero, el valor del juego a A es +1; si el jugador B alcanza el espacio 1 primero, entonces el valor del juego para A es -1.

- b) Sea l_i el valor mínimo (o máximo) de los nodos a la *izquierda* del nodo n_i a profundidad i , cuyo valor minimax es ya conocido. Del mismo modo, sea r_i el valor mínimo (o máximo) de los nodos *inexplorados* de la derecha de n_i a profundidad i . Rescriba la expresión para n_i en términos de los valores de r_i y l_i .
- c) Ahora reformule la expresión para demostrar que para afectar a n_i , n_j no debe exceder de una cierta cota obtenida de los valores de l_i .
- d) Repita el proceso para el caso donde n_i es un nodo min.



6.6 Implemente el algoritmo minimax esperado y el algoritmo *-alfa-beta, descrito por Ballard (1983), para podar árboles de juegos con nodos de posibilidad. Inténtelo sobre un juego como el *backgammon* y mida la eficacia de la poda *-alfa-beta.

6.7 Demuestre que con una transformación positiva lineal de valores de las hojas (es decir, transformando un valor x a $ax + b$ donde $a > 0$), la opción del movimiento permanece sin alterar en un árbol de juegos, aun cuando haya nodos posibilidad.

6.8 Considere el procedimiento siguiente para elegir movimientos en juegos con nodos de posibilidad:

- Genere algunas secuencias de lanzamientos de un dado (digamos, 50) a una profundidad conveniente (digamos, 8).
- Conocidos los lanzamientos del dado, el árbol de juegos se hace determinista. Para cada secuencia de lanzamientos del dado, resuelva el árbol de juegos determinista que ha resultado utilizando alfa-beta.
- Use los resultados para estimar el valor de cada movimiento y elegir el mejor.

¿Trabajará este procedimiento bien? ¿Por qué (no)?

6.9 Describa e implemente un entorno de juegos multijugador en tiempo real, donde el tiempo es parte del estado del ambiente y a los jugadores se le dan asignaciones de tiempo fijas.

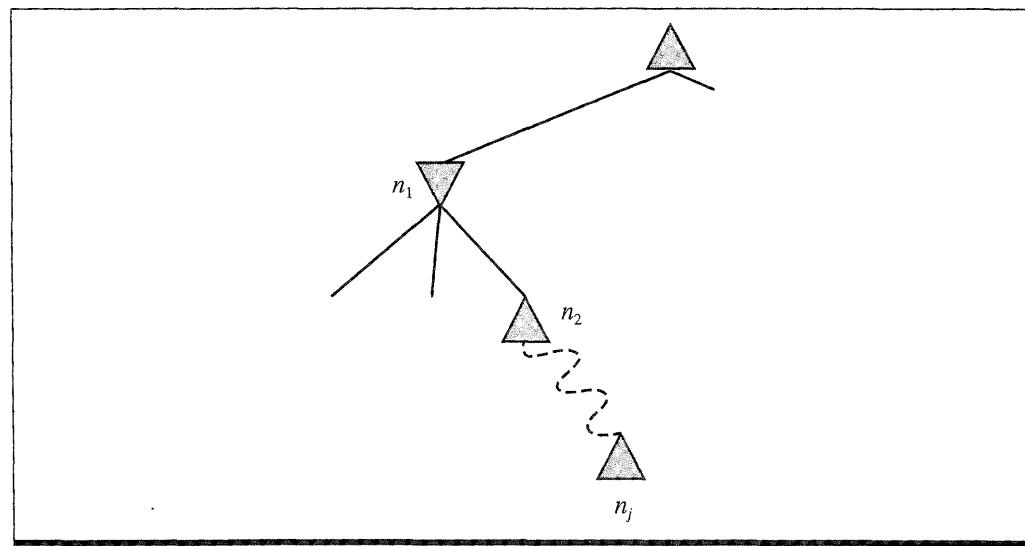


Figura 6.15 Situación cuando consideramos si hay que podar el nodo n_j .

6.10 Describa o implemente las descripciones de los estados, generadores de movimiento, test terminal, función de utilidad y funciones de evaluación para uno o varios de los juegos siguientes: monopoly, scrabble, bridge (asumiendo un contrato dado), y póker (elija su variedad favorita).

6.11 Considere con cuidado la interacción de acontecimientos de posibilidad e información parcial en cada uno de los juegos del Ejercicio 6.10.

- a) ¿Para cuáles es apropiado el minimax esperado estándar? Implemente el algoritmo y ejecútelo en su agente de juegos, con las modificaciones apropiadas al ambiente de juegos.
- b) ¿Para cuál es apropiado el esquema descrito en el Ejercicio 6.8?
- c) Discuta cómo podría tratar con el hecho que en algunos juegos, los jugadores no tienen el mismo conocimiento del estado actual.

6.12 El algoritmo minimax supone que los jugadores mueven por turnos, pero en juegos de cartas como *whist* y bridge, el ganador de la baza anterior juega primero sobre la siguiente baza.

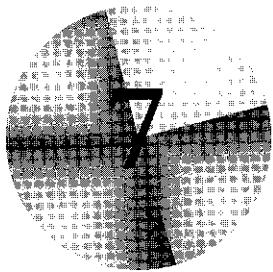
- a) Modifique el algoritmo para trabajar correctamente para estos juegos. Se supone que está disponible una función GANADOR(baza) que hace un informe sobre qué carta gana una baza.
- b) Dibuje el árbol de juegos para el primer par de manos de la página 200.

6.13 El programa de damas Chinook hace uso de bases de datos de final del juego, que proporcionan valores exactos para cada posición con ocho o menos piezas. ¿Cómo podrían generarse tales bases de datos de manera eficiente?

6.14 Discuta cómo la aproximación estándar de juegos se aplicaría a juegos como tenis, billar y croquet, que ocurren en un espacio de estado físico continuo.

6.15 Describa cómo los algoritmos minimax y alfa-beta cambian en **juegos de suma no cero** de dos jugadores en los que cada jugador tiene su propia función utilidad. Podríamos suponer que cada jugador sabe la función de utilidad del otro. Si no hay restricciones sobre las dos utilidades terminales, ¿es posible podar algún nodo con alfa-beta?

6.16 Suponga que tiene un programa de ajedrez que puede evaluar un millón de nodos por segundo. Decida una representación de un estado del juego para almacenarlo en una tabla de transposiciones. ¿Sobre cuántas entradas puede poner en una tabla de 500MB de memoria? ¿Será suficiente tres minutos de búsqueda para un movimiento? ¿Cuántas consultas de la tabla puede hacer en el tiempo utilizado para hacer una evaluación? Ahora suponga que la tabla de transposiciones es más grande que la que puede de caber en memoria. ¿Sobre cuántas evaluaciones podría hacer en el tiempo utilizado para realizar una búsqueda en disco con un disco estándar?



Agentes lógicos

Donde diseñaremos agentes que pueden construir representaciones del mundo, utilizar un proceso de inferencia para derivar nuevas representaciones del mundo, y emplear éstas para deducir qué hacer.

En este capítulo se introducen los agentes basados en conocimiento. Los conceptos que discutiremos (la *representación* del conocimiento y los procesos de *razonamiento* que permiten que éste evolucione) son centrales en todo el ámbito de la inteligencia artificial.

De algún modo, las personas conocen las cosas y realizan razonamientos. Tanto el conocimiento como el razonamiento son también importantes para los agentes artificiales, porque les permiten comportamientos con éxito que serían muy difíciles de alcanzar mediante otros mecanismos. Ya hemos visto cómo el conocimiento acerca de los efectos de las acciones permiten a los agentes que resuelven problemas actuar correctamente en entornos complejos. Un agente reflexivo sólo podría hallar un camino de Arad a Bucharest mediante la suerte del principiante. Sin embargo, el conocimiento de los agentes que resuelven problemas es muy específico e inflexible. Un programa de ajedrez puede calcular los movimientos permitidos de su rey, pero no puede saber de ninguna manera que una pieza no puede estar en dos casillas diferentes al mismo tiempo. Los agentes basados en conocimiento se pueden aprovechar del conocimiento expresado en formas muy genéricas, combinando y recombinando la información para adaptarse a diversos propósitos. A veces, este proceso puede apartarse bastante de las necesidades del momento (como cuando un matemático demuestra un teorema o un astrónomo calcula la esperanza de vida de la Tierra).

El conocimiento y el razonamiento juegan un papel importante cuando se trata con entornos parcialmente observables. Un agente basado en conocimiento puede combinar

el conocimiento general con las percepciones reales para inferir aspectos ocultos del estado del mundo, antes de seleccionar cualquier acción. Por ejemplo, un médico diagnostica a un paciente (es decir, infiere una enfermedad que no es directamente observable) antes de seleccionar un tratamiento. Parte del conocimiento que utiliza el médico está en forma de reglas que ha aprendido de los libros de texto y sus profesores, y parte en forma de patrones de asociación que el médico no es capaz de describir explícitamente. Si este conocimiento está en la cabeza del médico, es su conocimiento.

El entendimiento del lenguaje natural también necesita inferir estados ocultos, en concreto, la intención del que habla. Cuando escuchamos, «John vió el diamante a través de la ventana y lo codició», sabemos que «lo» se refiere al diamante y no a la ventana (quizá de forma inconsciente, razonamos con nuestro conocimiento acerca del papel relativo de las cosas). De forma similar, cuando escuchamos, «John lanzó el ladrillo a la ventana y se rompió», sabemos que «se» se refiere a la ventana. El razonamiento nos permite hacer frente a una variedad virtualmente infinita de manifestaciones utilizando un conjunto finito de conocimiento de sentido común. Los agentes que resuelven problemas presentan dificultades con este tipo de ambigüedad debido a que su representación de los problemas con contingencias es inherentemente exponencial.

Nuestra principal razón para estudiar los agentes basados en conocimiento es su flexibilidad. Ellos son capaces de aceptar tareas nuevas en forma de objetivos descritos explícitamente, pueden obtener rápidamente competencias informándose acerca del conocimiento del entorno o aprendiéndolo, y pueden adaptarse a los cambios del entorno actualizando el conocimiento relevante.

En la Sección 7.1 comenzamos con el diseño general del agente. En la Sección 7.2 se introduce un nuevo entorno muy sencillo, el mundo de *wumpus*, y se muestra la forma de actuar de un agente basado en conocimiento sin entrar en los detalles técnicos. Entonces, en la Sección 7.3, explicamos los principios generales de la **lógica**. La lógica será el instrumento principal para la representación del conocimiento en toda la Parte III de este libro. El conocimiento de los agentes lógicos siempre es *categórico* (cada proposición acerca del mundo es verdadera o falsa, si bien, el agente puede ser agnóstico acerca de algunas proposiciones).

La lógica presenta la ventaja pedagógica de ser un ejemplo sencillo de representación para los agentes basados en conocimiento, pero tiene serias limitaciones. En concreto, gran parte del razonamiento llevado a cabo por las personas y otros agentes en entornos parcialmente observables se basa en manejar conocimiento que es *incierto*. La lógica no puede representar bien esta incertidumbre, así que trataremos las probabilidades en la Parte V, que sí puede. Y en la Parte VI y la Parte VII trataremos otras representaciones, incluidas algunas basadas en matemáticas continuas como combinaciones de funciones Gaussianas, redes neuronales y otras representaciones.

En la Sección 7.4 de este capítulo se presenta una lógica muy sencilla denominada **lógica proposicional**. Aunque es mucho menos expresiva que la **lógica de primer orden** (Capítulo 8), la lógica proposicional nos permitirá ilustrar los conceptos fundamentales de la lógica. En las secciones 7.5 y 7.6 describiremos la tecnología, que está bastante desarrollada, para el razonamiento basado en lógica proposicional. Finalmente, en la sección 7.7 se combina el concepto de agente lógico con la tecnología de la lógica proposicional para la construcción de unos agentes muy sencillos en nuestro ejemplo.

del mundo de *wumpus*. Se identifican ciertas deficiencias de la lógica proposicional, que nos permitirán el desarrollo de lógicas más potentes en los capítulos siguientes.

7.1 Agentes basados en conocimiento

BASE DE CONOCIMIENTO
SENTENCIA
LENGUAJE DE REPRESENTACIÓN DEL CONOCIMIENTO
INFERENCIA
AGENTES LÓGICOS

CONOCIMIENTO DE ANTECEDENTES

El componente principal de un agente basado en conocimiento es su **base de conocimiento**, o BC. Informalmente, una base de conocimiento es un conjunto de **sentencias**. (Aquí «sentencia» se utiliza como un término técnico. Es parecido, pero no idéntico, a las sentencias en inglés u otros lenguajes naturales.) Cada sentencia se expresa en un lenguaje denominado **lenguaje de representación del conocimiento** y representa alguna aserción acerca del mundo.

Debe haber un mecanismo para añadir sentencias nuevas a la base de conocimiento, y uno para preguntar qué se sabe en la base de conocimiento. Los nombres estándar para estas dos tareas son DECIR y PREGUNTAR, respectivamente. Ambas tareas requieren realizar **inferencia**, es decir, derivar nuevas sentencias de las antiguas. En los **agentes lógicos**, que son el tema principal de estudio de este capítulo, la inferencia debe cumplir con el requisito esencial de que cuando se PREGUNTA a la base de conocimiento, la respuesta debe seguirse de lo que se HA DICHO a la base de conocimiento previamente. Más adelante, en el capítulo, seremos más precisos en cuanto a la palabra «seguirse». Por ahora, tómate su significado en el sentido de que la inferencia no se inventaría cosas poco a poco.

La Figura 7.1 muestra el esquema general de un programa de un agente basado en conocimiento. Al igual que todos nuestros agentes, éste recibe una percepción como entrada y devuelve una acción. El agente mantiene una base de conocimiento, BC, que inicialmente contiene algún **conocimiento de antecedentes**. Cada vez que el programa del agente es invocado, realiza dos cosas. Primero, DICE a la base de conocimiento lo que ha percibido. Segundo, PREGUNTA a la base de conocimiento qué acción debe ejecutar. En este segundo proceso de responder a la pregunta, se debe realizar un razonamiento extensivo acerca del estado actual del mundo, de los efectos de las posibles acciones, etcétera. Una vez se ha escogido la acción, el agente graba su elección mediante un DECIR y ejecuta la acción. Este segundo DECIR es necesario para permitirle a la base de conocimiento saber que la acción hipotética realmente se ha ejecutado.

**función AGENTE-BC(*percepción*) devuelve una acción
variables estáticas: BC, una base de conocimiento
t, un contador, inicializado a 0, que indica el tiempo**

```

DECIR(BC, CONSTRUIR-SENTENCIA-DE-PERCEPCIÓN(percepción, t))
acción ← PREGUNTAR(BC, PEDIR-Acción(tacción, t))
t ← t + 1
devolver acción

```

Figura 7.1 Un agente basado en conocimiento genérico.

Los detalles del lenguaje de representación están ocultos en las dos funciones que implementan la interfaz entre los sensores, los accionadores, el núcleo de representación y el sistema de razonamiento. **CONSTRUIR-SENTENCIA-DE-PERCEPCIÓN** toma una percepción y un instante de tiempo y devuelve una sentencia afirmando lo que el agente ha percibido en ese instante de tiempo. **PEDIR-ACCIÓN** toma un instante de tiempo como entrada y devuelve una sentencia para preguntarle a la base de conocimiento qué acción se debe realizar en ese instante de tiempo. Los detalles de los mecanismos de inferencia están ocultos en **DECIR** y **PREGUNTAR**. En las próximas secciones del capítulo se mostrarán estos detalles.

El agente de la Figura 7.1 se parece bastante a los agentes con estado interno descritos en el Capítulo 2. Pero gracias a las definiciones de **DECIR** y **PREGUNTAR**, el agente basado en conocimiento no obtiene las acciones mediante un proceso arbitrario. Es compatible con una descripción al **nivel de conocimiento**, en el que sólo necesitamos especificar lo que el agente sabe y los objetivos que tiene para establecer su comportamiento. Por ejemplo, un taxi automatizado podría tener el objetivo de llevar un pasajero al condado de Marin, y podría saber que está en San Francisco y que el puente Golden Gate es el único enlace entre las dos localizaciones. Entonces podemos esperar que el agente cruce el puente Golden Gate *porque él sabe que hacerlo le permitirá alcanzar su objetivo*. Fíjate que este análisis es independiente de cómo el taxi trabaja al **nivel de implementación**. Al agente no le debe importar si el conocimiento geográfico está implementado mediante listas enlazadas o mapas de píxeles, o si su razonamiento se realiza mediante la manipulación de textos o símbolos almacenados en registros, o mediante la propagación de señales en una red de neuronas.



Tal como comentamos en la introducción del capítulo, *uno puede construir un agente basado en conocimiento simplemente DICIÉNDOLE al agente lo que necesita saber*. El programa del agente, inicialmente, antes de que empiece a recibir percepciones, se construye mediante la adición, una a una, de las sentencias que representan el conocimiento del entorno que tiene el diseñador. El diseño del lenguaje de representación que permita, de forma más fácil, expresar este conocimiento mediante sentencias simplifica muchísimo el problema de la construcción del agente. Este enfoque en la construcción de sistemas se denomina **enfoque declarativo**. Por el contrario, el enfoque procedural codifica los comportamientos que se desean obtener directamente en código de programación; mediante la minimización del papel de la representación explícita y del razonamiento se pueden obtener sistemas mucho más eficientes. En la Sección 7.7 veremos agentes de ambos tipos. En los 70 y 80, defensores de los dos enfoques se enfrentaban en acalorados debates. Ahora sabemos que para que un agente tenga éxito su diseño debe combinar elementos declarativos y procedurales.

A parte de DECIRLE al agente lo que necesita saber, podemos proveer a un agente basado en conocimiento de los mecanismos que le permitan aprender por sí mismo. Estos mecanismos, que se verán en el Capítulo 18, crean un conocimiento general acerca del entorno con base en un conjunto de percepciones. Este conocimiento se puede incorporar a la base de conocimiento del agente y utilizar para su toma de decisiones. De esta manera, el agente puede ser totalmente autónomo.

Todas estas capacidades (representación, razonamiento y aprendizaje) se apoyan en la teoría y tecnología de la lógica, desarrolladas a lo largo de los siglos. Sin embargo,

NIVEL DE CONOCIMIENTO

NIVEL DE IMPLEMENTACIÓN

ENFOQUE DECLARATIVO

antes de explicar dichas teoría y tecnología, crearemos un mundo sencillo que nos permitirá ilustrar estos mecanismos.

7.2 El mundo de *wumpus*

MUNDO DE *WUMPUS*

El **mundo de *wumpus*** es una cueva que está compuesta por habitaciones conectadas mediante pasillos. Escondido en algún lugar de la cueva está el *wumpus*, una bestia que se come a cualquiera que entre en su habitación. El *wumpus* puede ser derribado por la flecha de un agente, y éste sólo dispone de una. Algunas habitaciones contienen hoyos sin fondo que atrapan a aquel que deambula por dichas habitaciones (menos al *wumpus*, que es demasiado grande para caer en ellos). El único premio de vivir en este entorno es la posibilidad de encontrar una pila de oro. Aunque el mundo de *wumpus* pertenece más al ámbito de los juegos por computador, es un entorno perfecto para evaluar los agentes inteligentes. Michael Genesereth fue el primero que lo propuso.

En la Figura 7.2 se muestra un ejemplo del mundo de *wumpus*. La definición precisa del entorno de trabajo, tal como sugerimos en el Capítulo 2, mediante la descripción REAS, es:

- **Rendimiento:** +1.000 por recoger el oro, -1.000 por caer en un hoyo o ser comido por el *wumpus*, -1 por cada acción que se realice y -10 por lanzar la flecha.
- **Entorno:** una matriz de 4×4 habitaciones. El agente siempre comienza en la casilla etiquetada por [1, 1], y orientado a la derecha. Las posiciones del oro y del *wumpus* se escogen de forma aleatoria, mediante una distribución uniforme, a partir de todas las casillas menos la de salida del agente. Además, con probabilidad 0,2, cada casilla puede tener un hoyo.
- **Actuadores:** el agente se puede mover hacia delante, girar a la izquierda 90° , o a la derecha 90° . El agente puede fallecer de muerte miserable si entra en una casilla en la que hay un hoyo o en la que está el *wumpus* vivo. (No sucede nada malo, aunque huele bastante mal, si el agente entra en una casilla con un *wumpus* muerto.) Si hay un muro en frente y el agente intenta avanzar, no sucede nada. La acción *Agarrar* se puede utilizar para tomar un objeto de la misma casilla en donde se encuentre el agente. La acción *Disparar* se puede utilizar para lanzar una flecha en línea recta, en la misma dirección y sentido en que se encuentra situado el agente. La flecha avanza hasta que se choca contra un muro o alcanza al *wumpus* (y entonces lo mata). El agente sólo dispone de una flecha, así que, sólo tiene efecto el primer *Disparo*.
- **Sensores:** el agente dispone de cinco sensores, y cada uno le da una pequeña información acerca del entorno.
 - El agente percibirá un mal hedor si se encuentra en la misma casilla que el *wumpus* o en las directamente adyacentes a él (no en diagonal).
 - El agente recibirá una pequeña brisa en las casillas directamente adyacentes donde hay un hoyo.
 - El agente verá un resplandor en las casillas donde está el oro.

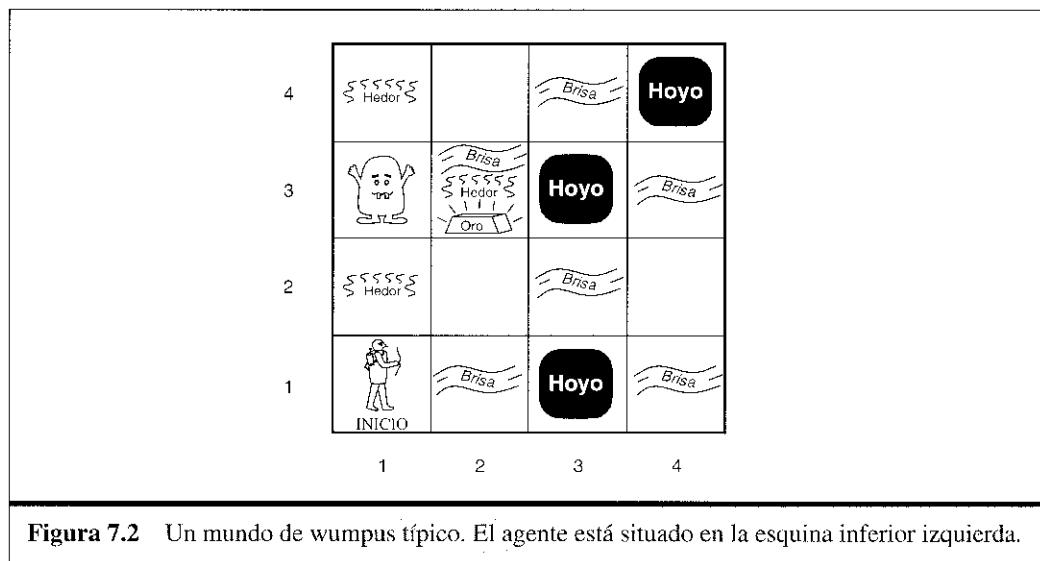
- Si el agente intenta atravesar un muro sentirá un golpe.
 - Cuando el *wumpus* es aniquilado emite un desconsolado grito que se puede oír en toda la cueva.

Las percepciones que recibirá el agente se representan mediante una lista de cinco símbolos: por ejemplo, si el agente percibe un mal hedor o una pequeña brisa, pero no ve un resplandor, no siente un golpe, ni oye un grito, el agente recibe la lista [*Hedor, Brisa, Nada, Nada, Nada*].

En el Ejercicio 7.1 se pide definir el entorno del *wumpus* a partir de las diferentes dimensiones tratadas en el Capítulo 2. La principal dificultad para el agente es su ignorancia inicial acerca de la configuración del entorno; para superar esta ignorancia parece que se requiere el razonamiento lógico. En muchos casos del mundo de *wumpus*, para el agente es posible obtener el oro de forma segura. En algunos casos, el agente debe escoger entre volver a casa con las manos vacías o arriesgarse para encontrar el oro. Cerca del 21 por ciento de los casos son completamente injustos, ya que el oro se encuentra en un hoyo o rodeado de ellos.

Vamos a ver un agente basado en conocimiento en el mundo de *wumpus*, explorando el entorno que se muestra en la Figura 7.2. La base de conocimiento inicial del agente contiene las reglas del entorno, tal como hemos listado anteriormente; en concreto, el agente sabe que se encuentra en la casilla [1, 1] y que ésta es una casilla segura. Vemos cómo su conocimiento evoluciona a medida que recibe nuevas percepciones y las acciones se van ejecutando.

La primera percepción es [*Nada*, *Nada*, *Nada*, *Nada*, *Nada*], de la cual, el agente puede concluir que las casillas vecinas son seguras. La Figura 7.3(a) muestra el conocimiento del estado del agente en ese momento. En esta figura mostramos (algunas de) las sentencias de la base de conocimiento utilizando letras como la *B* (de brisa) y *OK* (de casilla segura, no hay hoyo ni está el *wumpus*) situadas en las casillas adecuadas. En cambio, la Figura 7.2 muestra el mundo tal como es.



De los hechos, que no hay mal hedor ni brisa en la casilla [1, 1], el agente infiere que las casillas [1, 2] y [2, 1] están libres de peligro. Entonces las marca con *OK* para indicar esta conclusión. Un agente que sea cauto sólo se moverá hacia una casilla en la que él sabe que está *OK*. Supongamos que el agente decide moverse hacia delante a la casilla [2, 1], alcanzando la situación de la Figura 7.3(b).

El agente detecta una brisa en la casilla [2, 1], por lo tanto, debe haber un hoyo en alguna casilla vecina. El hoyo no puede estar en la casilla [1, 1], teniendo en cuenta las reglas del juego, así que debe haber uno en la casilla [2, 2] o en la [3, 1], o en ambas. La etiqueta *?P?* de la Figura 7.3(b), nos indica que puede haber un posible hoyo en estas casillas. En este momento, sólo se conoce una casilla que está *OK* y que no ha sido visitada aún. Así que el agente prudente girará para volver a la casilla [1, 1] y entonces se moverá a la [1, 2].

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1 A OK	2,1 OK	3,1	4,1

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2 ?P? OK	3,2	4,2
1,1 V OK	2,1 A B OK	3,1 ?P?	4,1

(a)
(b)

Figura 7.3 El primer paso dado por el agente en el mundo de *wumpus*. (a) La situación inicial, después de la percepción [*Nada*, *Nada*, *Nada*, *Nada*, *Nada*]. (b) Despues del primer movimiento, con la percepción [*Nada*, *Brisa*, *Nada*, *Nada*, *Nada*].

La nueva percepción en la casilla [1, 2] es [*Hedor*, *Nada*, *Nada*, *Nada*, *Nada*], obteniendo el estado de conocimiento que se muestra en la Figura 7.4(a). El mal hedor en la [1, 2] significa que debe haber un *wumpus* muy cerca. Pero el *wumpus* no puede estar en la [1, 1], teniendo en cuenta las reglas del juego, y tampoco puede estar en [2, 2] (o el agente habría detectado un mal hedor cuando estaba en la [2, 1]). Entonces el agente puede inferir que el *wumpus* se encuentra en la casilla [1, 3], que se indica con la etiqueta *?W!* Más aún, la ausencia de *Brisa* en la casilla [1, 2] implica que no hay un hoyo en la [2, 2]. Como ya habíamos inferido que debía haber un hoyo en la casilla [2, 2] o en la [3, 1], éste debe estar en la [3, 1]. Todo esto es un proceso de inferencia realmente costoso, ya que debe combinar el conocimiento adquirido en diferentes instantes de tiempo y en distintas situaciones, para resolver la falta de percepciones y poder realizar cualquier paso crucial. La inferencia pertenece a las habilidades de muchos animales, pero es típico del tipo de razonamiento que un agente lógico realiza.

Figura 7.4 Los dos últimos estados en el desarrollo del juego. (a) Despues del tercer movimiento, con la percepción [Hedor, Nada, Nada, Nada, Nada]. (b) Despues del quinto movimiento, con la percepción [Hedor, Brisa, Resplandor, Nada, Nada].

El agente ha demostrado en este momento que no hay ni un hoyo ni un *wumpus* en la casilla [2, 2], así que está *OK* para desplazarse a ella. No mostraremos el estado de conocimiento del agente en [2, 2]; asumimos que el agente gira y se desplaza a [2, 3], tal como se muestra en la Figura 7.4(b). En la casilla [2, 3] el agente detecta un resplandor, entonces el agente cogería el oro y acabaría el juego.



En cada caso en que el agente saca una conclusión a partir de la información que tiene disponible, se garantiza que dicha conclusión es correcta si la información disponible también lo es. Esta es una propiedad fundamental del razonamiento lógico. En lo que queda del capítulo vamos a describir cómo construir agentes lógicos que pueden representar la información necesaria para sacar conclusiones similares a las que hemos descrito en los párrafos anteriores.

7.3 Lógica

Esta sección presenta un repaso de todos los conceptos fundamentales de la representación y el razonamiento lógicos. Dejamos los detalles técnicos de cualquier clase concreta de lógica para la siguiente sección. En lugar de ello, utilizaremos ejemplos informales del mundo de *wumpus* y del ámbito familiar de la aritmética. Adoptamos este enfoque poco común, porque los conceptos de la lógica son bastante más generales y bellos de lo que se piensa a priori.

En la sección 7.1 dijimos que las bases de conocimiento se componen de sentencias. Estas sentencias se expresan de acuerdo a la **sintaxis** del lenguaje de representación, que especifica todas las sentencias que están bien formadas. El concepto de sintaxis está suficientemente claro en la aritmética; « $x + y = 4$ » es una sentencia bien formada, mien-

tras que « $x2y + =$ » no lo es. Por lo general, la sintaxis de los lenguajes lógicos (y la de los aritméticos, en cuanto al mismo tema) está diseñada para escribir libros y artículos. Hay literalmente docenas de diferentes sintaxis, algunas que utilizan muchas letras griegas y símbolos matemáticos complejos, otras basadas en diagramas con flechas y burbujas visualmente muy atractivas. Y sin embargo, en todos estos casos, las sentencias de la base de conocimiento del agente son configuraciones físicas reales (de las partes) del agente. El razonamiento implica generar y manipular estas configuraciones.

SEMÁNTICA

VALOR DE VERDAD

MUNDO POSIBLE

MODELO

IMPLICACIÓN

Una lógica también debe definir la **semántica** del lenguaje. Si lo relacionamos con el lenguaje hablado, la semántica trata el «significado» de las sentencias. En lógica, esta definición es bastante más precisa. La semántica del lenguaje define el **valor de verdad** de cada sentencia respecto a cada **mundo posible**. Por ejemplo, la semántica que se utiliza en la aritmética especifica que la sentencia « $x + y = 4$ » es verdadera en un mundo en el que x sea 2 e y sea 2, pero falsa en uno en el que x sea 1 e y sea 1¹. En las lógicas clásicas cada sentencia debe ser o bien verdadera o bien falsa en cada mundo posible, no puede ser lo uno y lo otro².

Cuando necesitemos ser más precisos, utilizaremos el término **modelo** en lugar del de «mundo posible». (También utilizaremos la frase « m es un modelo de α » para indicar que la sentencia α es verdadera en el modelo m .) Siempre que podamos pensar en los mundos posibles como en (potencialmente) entornos reales en los que el agente pueda o no estar, los modelos son abstracciones matemáticas que simplemente nos permiten definir la verdad o falsedad de cada sentencia que sea relevante. Informalmente podemos pensar, por ejemplo, en que x e y son el número de hombres y mujeres que están sentados en una mesa jugando una partida de *bridge*, y que la sentencia $x + y = 4$ es verdadera cuando los que están jugando son cuatro en total; formalmente, los modelos posibles son justamente todas aquellas posibles asignaciones de números a las variables x e y . Cada una de estas asignaciones indica el valor de verdad de cualquier sentencia aritmética cuyas variables son x e y .

Ahora que ya disponemos del concepto de valor de verdad, ya estamos preparados para hablar acerca del razonamiento lógico. Éste requiere de la relación de **implicación** lógica entre las sentencias (la idea de que una sentencia *se sigue lógicamente* de otra sentencia). Su notación matemática es

$$\alpha \models \beta$$

para significar que la sentencia α implica la sentencia β . La definición formal de implicación es esta: $\alpha \models \beta$ si y sólo si en cada modelo en el que α es verdadera, β también lo es. Otra forma de definirla es que si α es verdadera, β también lo debe ser. Informalmente, el valor de verdad de β «está contenido» en el valor de verdad de α . La relación de implicación nos es familiar en la aritmética; no nos disgusta la idea de que la sentencia $x + y = 4$ implica la sentencia $4 = x + y$. Es obvio que en cada modelo en

¹ El lector se habrá dado cuenta de la semejanza entre el concepto de valor de verdad de las sentencias y la satisfacción de restricciones del Capítulo 5. No es casualidad (los lenguajes de restricciones son en efecto lógicas y la resolución de restricciones un tipo de razonamiento lógico).

² La **lógica difusa**, que se verá en el Capítulo 14, nos permitirá tratar con grados de valores de verdad.

el que $x + y = 4$ (como lo es el modelo en el que x es 2 e y es 2) también lo es para $4 = x + y$. Pronto veremos que una base de conocimiento puede ser considerada como una afirmación, y a menudo hablaremos de que una base de conocimiento implica una sentencia.

Ahora podemos aplicar el mismo tipo de análisis que utilizamos en la sección anterior al mundo del *wumpus*. Si tomamos la situación de la Figura 7.3(b): el agente no ha detectado nada en la casilla [1, 1], y ha detectado una brisa en la [2, 1]. Estas percepciones, combinadas con el conocimiento del agente sobre las reglas que definen el funcionamiento del mundo de *wumpus* (la descripción REAS de la página 221), constituyen su BC. El agente está interesado (entre otras cosas) en si las casillas adyacentes [1, 2], [2, 2] y [3, 1] tienen hoyos sin fondo. Cada una de las tres casillas pueden o no tener un hoyo, por lo tanto (al menos en este ejemplo) hay $3^3 = 8$ modelos posibles. Tal como se muestran en la Figura 7.5³.

La BC es falsa en los modelos que contradicen lo que el agente sabe (por ejemplo, la BC es falsa en cualquier modelo en el que la casilla [1, 2] tenga un hoyo), porque no ha detectado ninguna brisa en la casilla [1, 1]. De hecho, hay tres modelos en los que la BC es verdadera, los que se muestran como subconjunto de los modelos de la Figura 7.5. Ahora consideremos las dos conclusiones:

$$\alpha_1 = \text{«No hay un hoyo en la casilla [1, 2]»}.$$

$$\alpha_2 = \text{«No hay un hoyo en la casilla [2, 2]»}.$$

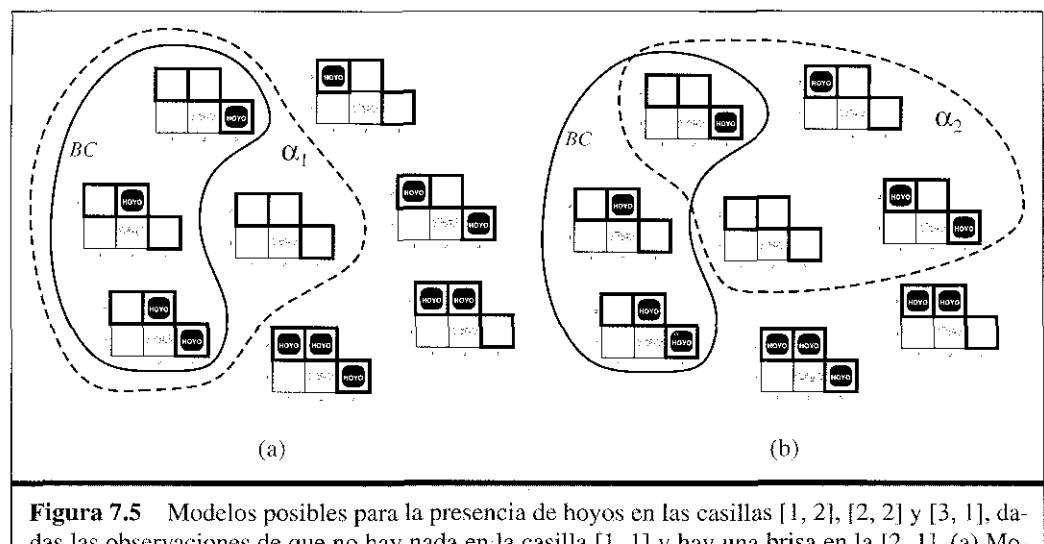


Figura 7.5 Modelos posibles para la presencia de hoyos en las casillas [1, 2], [2, 2] y [3, 1], dadas las observaciones de que no hay nada en la casilla [1, 1] y hay una brisa en la [2, 1]. (a) Modelos de la base de conocimiento y α_1 (no hay un hoyo en [1, 2]). (b) Modelos de la base de conocimiento y α_2 (no hay un hoyo en [2, 2]).

³ En la Figura 7.5 los modelos se muestran como mundos parciales, porque en realidad tan sólo son asignaciones de *verdadero* y *falso* a sentencias como «hay un hoyo en la casilla [1, 2]», etc. Los modelos, desde el punto de vista matemático, no necesitan tener horribles wumpus etéreos ambulando en ellos.

Hemos rodeado (con línea discontinua) los modelos de α_1 y α_2 en las Figuras 7.5(a) y 7.5(b) respectivamente. Si observamos, podemos ver lo siguiente:

en cada modelo en el que la BC es verdadera, α_1 también lo es.

De aquí que $BC \models \alpha_1$: no hay un hoyo en la casilla [1, 2]. También podemos ver que

en algunos modelos en los que la BC es verdadera, α_2 es falsa.

De aquí que $BC \not\models \alpha_2$: el agente no puede concluir que no haya un hoyo en la casilla [2, 2]. (Ni tampoco puede concluir que lo haya.⁴)

El ejemplo anterior no sólo nos muestra el concepto de implicación, sino, también cómo el concepto de implicación se puede aplicar para derivar conclusiones, es decir, llevar a cabo la **inferencia lógica**. El algoritmo de inferencia que se muestra en la Figura 7.5 se denomina **comprobación de modelos** porque enumera todos los modelos posibles y comprueba si α es verdadera en todos los modelos en los que la BC es verdadera.

Para entender la implicación y la inferencia nos puede ayudar pensar en el conjunto de todas las consecuencias de la BC como en un pajar, y en α como en una aguja. La implicación es como la aguja que se encuentra en el pajar, y la inferencia consiste en encontrarla. Esta distinción se expresa mediante una notación formal: si el algoritmo de inferencia i puede derivar α de la BC , entonces escribimos

$$BC \vdash_i \alpha,$$

que se pronuncia como «« α se deriva de la BC mediante i » o «« i deriva α de la BC ».

Se dice que un algoritmo de inferencia que deriva sólo sentencias implicadas es **sólido** o que **mantiene la verdad**. La solidez es una propiedad muy deseable. Un procedimiento de inferencia no sólido tan sólo se inventaría cosas poco a poco (anunciaría el descubrimiento de agujas que no existirían). Se puede observar fácilmente que la comprobación de modelos, cuando es aplicable⁵, es un procedimiento sólido.

También es muy deseable la propiedad de **completitud**: un algoritmo de inferencia es completo si puede derivar cualquier sentencia que está implicada. En los pajares reales, que son de tamaño finito, parece obvio que un examen sistemático siempre permite decidir si hay una aguja en el pajar. Sin embargo, en muchas bases de conocimiento, el pajar de las consecuencias es infinito, y la completitud pasa a ser una problemática importante⁶. Por suerte, hay procedimientos de inferencia completos para las lógicas que son suficientemente expresivas para manejar muchas bases de conocimiento.

⁴ El agente podría calcular la *probabilidad* de que haya un hoyo en la casilla [2, 2]; en el Capítulo 13 lo veremos.

⁵ La comprobación de modelos trabaja bien cuando el espacio de los modelos es finito (por ejemplo, en un mundo del *wumpus* de tamaño de casillas fijo). Por el otro lado, en la aritmética, el espacio de modelos es infinito: aun limitándonos a los enteros, hay infinitos pares de valores para x e y en la sentencia $x + y = 4$.

⁶ Compárello con el caso de la búsqueda en espacios infinitos de estados del Capítulo 3, en donde la búsqueda del primero en profundidad no es completa.



Hemos descrito un proceso de razonamiento en el que se garantiza que las conclusiones sean verdaderas en cualquier mundo en el que las premisas lo sean; en concreto, *si una BC es verdadera en el mundo real, entonces cualquier sentencia α que se derive de la BC mediante un procedimiento de inferencia sólido también será verdadera en el mundo real*. Así, mientras que un proceso de inferencia opera con la «sintaxis» (las configuraciones físicas internas, tales como los bits en los registros o los patrones de impulsos eléctricos en el cerebro) el proceso se corresponde con la relación del mundo real según la cual algún aspecto del mundo real es cierto⁷ en virtud de que otros aspectos del mundo real lo son. En la Figura 7.6 se ilustra esta correspondencia entre el mundo y la representación.

DENOTACIÓN

El último asunto que debe ser tratado mediante una computación basada en agentes lógicos es el de la **denotación** (la conexión, si la hay, entre los procesos de razonamiento lógico y el entorno real en el que se encuentra el agente). En concreto, *¿cómo sabemos que la BC es verdadera en el mundo real?* (Después de todo, la BC sólo es «sintaxis» dentro de la cabeza del agente.) Ésta es una cuestión filosófica acerca de la cual se han escrito muchos, muchísimos libros. (Ver Capítulo 26.) Una respuesta sencilla es que los sensores del agente crean la conexión. Por ejemplo, nuestro agente del mundo de *wumpus* dispone de un sensor de olores. El programa del agente crea una sentencia adecuada siempre que hay un olor. Entonces, siempre que esa sentencia esté en la base de conocimiento será verdadera en el mundo real. Así, el significado y el valor de verdad de las sentencias de las percepciones se definen mediante el proceso de los sensores y el de la construcción de las sentencias, activada por el proceso previo. ¿Qué sucede con el resto del conocimiento del agente, tal como sus creencias acerca de que el *wumpus* causa mal hedor en las casillas adyacentes? Ésta no es una representación directa de una simple percepción, pero sí es una regla general (derivada, quizás, de la experiencia de las percepciones aunque no idéntica a una afirmación de dicha experiencia). Las reglas generales como ésta se generan mediante un proceso de construcción de sentencias denominado **aprendizaje**, que es el tema que trataremos en la Parte VI. El aprendizaje es falible. Puede darse el caso en el que el *wumpus* cause mal hedor *excepto el 29 de febrero en años bisiestos*, que

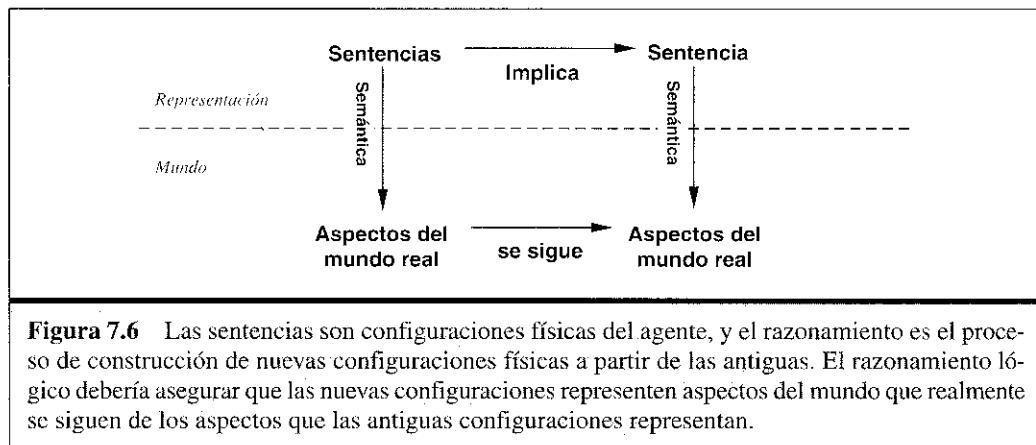


Figura 7.6 Las sentencias son configuraciones físicas del agente, y el razonamiento es el proceso de construcción de nuevas configuraciones físicas a partir de las antiguas. El razonamiento lógico debería asegurar que las nuevas configuraciones representen aspectos del mundo que realmente se siguen de los aspectos que las antiguas configuraciones representan.

⁷ Tal como escribió Wittgenstein (1922) en su famoso *Tractatus*: «El mundo es cada cosa que es cierta».

es cuando toma su baño. Así, la *BC* no sería verdadera en el mundo real, sin embargo, mediante procedimientos de aprendizaje buenos no hace falta ser tan pesimistas.

7.4 Lógica proposicional: una lógica muy sencilla

LÓGICA PROPOSICIONAL

Ahora vamos a presentar una lógica muy sencilla llamada **Lógica proposicional**⁸. Vamos a cubrir tanto la sintaxis como la semántica (la manera como se define el valor de verdad de las sentencias) de la lógica proposicional. Luego trataremos la **implicación** (la relación entre una sentencia y la que se sigue de ésta) y veremos cómo todo ello nos lleva a un algoritmo de inferencia lógica muy sencillo. Todo ello tratado, por supuesto, en el mundo de *wumpus*.

Sintaxis

SENTENCIAS ATÓMICAS

SÍMBOLO PROPOSICIONAL

SENTENCIAS COMPLEJAS

CONECTIVAS LÓGICAS

NEGACIÓN

LITERAL

CONJUNCIÓN

DISYUNCIÓN

IMPlicación

PREMISA

CONCLUSIÓN

La **sintaxis** de la lógica proposicional nos define las sentencias que se pueden construir. Las **sentencias atómicas** (es decir, los elementos sintácticos indivisibles) se componen de un único **símbolo proposicional**. Cada uno de estos símbolos representa una proposición que puede ser verdadera o falsa. Utilizaremos letras mayúsculas para estos símbolos: *P*, *Q*, *R*, y siguientes. Los nombres de los símbolos suelen ser arbitrarios pero a menudo se escogen de manera que tengan algún sentido mnemotécnico para el lector. Por ejemplo, podríamos utilizar $W_{1,3}$ para representar que el *wumpus* se encuentra en la casilla [1, 3]. (Recuerde que los símbolos como $W_{1,3}$ son *atómicos*, esto es, *W*, 1, y 3 no son partes significantes del símbolo.) Hay dos símbolos proposicionales con significado fijado: *Verdadero*, que es la proposición que siempre es verdadera; y *Falso*, que es la proposición que siempre es falsa.

Las **sentencias complejas** se construyen a partir de sentencias más simples mediante el uso de las **conectivas lógicas**, que son las siguientes cinco:

- ¬ (no). Una sentencia como $\neg W_{1,3}$ se denomina **negación** de $W_{1,3}$. Un **literal** puede de ser una sentencia atómica (un **literal positivo**) o una sentencia atómica negada (un **literal negativo**).
- ∧ (y). Una sentencia que tenga como conectiva principal \wedge , como es $W_{1,3} \wedge H_{3,1}$, se denomina **conjunción**; sus componentes son los **conjuntores**.
- ∨ (o). Una sentencia que utiliza la conectiva \vee , como es $(W_{1,3} \wedge H_{3,1}) \vee W_{2,2}$, es una **disyunción de los disyuntores** ($W_{1,3} \wedge H_{3,1}$) y $W_{2,2}$. (Históricamente, la conectiva \vee proviene de «vel» en Latín, que significa «o». Para mucha gente, es más fácil recordarla como la conjunción al revés.)
- \Rightarrow (implica). Una sentencia como $(W_{1,3} \wedge H_{3,1}) \Rightarrow \neg W_{2,2}$ se denomina **implicación** (o condicional). Su **premisa o antecedente** es $(W_{1,3} \wedge H_{3,1})$, y su **conclusión o conseciente** es $\neg W_{2,2}$. Las implicaciones también se conocen como **reglas** o afir-

⁸ A la lógica proposicional también se le denomina **Lógica Booleana**, por el matemático George Boole (1815-1864).

BICONDICIONAL

maciones **si-entonces**. Algunas veces, en otros libros, el símbolo de la implicación se representa mediante \supset o \rightarrow .

\Leftrightarrow (si y sólo si). La sentencia $W_{1,3} \Leftrightarrow \neg W_{2,2}$ es una **bicondicional**.

En la Figura 7.7 se muestra una gramática formal de la lógica proposicional; mira la página 984 si no estás familiarizado con la notación BNF.

$\text{Sentencia} \rightarrow \text{Sentencia Atómica} \mid \text{Sentencia Compleja}$ $\text{Sentencia Atómica} \rightarrow \text{Verdadero} \mid \text{Falso} \mid \text{Símbolo Proposicional}$ $\text{Símbolo Proposicional} \rightarrow \mathbf{P} \mid \mathbf{Q} \mid \mathbf{R} \mid \dots$ $\text{Sentencia Compleja} \rightarrow \neg \text{Sentencia}$
$\mid (\text{Sentencia} \wedge \text{Sentencia})$ $\mid (\text{Sentencia} \vee \text{Sentencia})$ $\mid (\text{Sentencia} \Rightarrow \text{Sentencia})$ $\mid (\text{Sentencia} \Leftrightarrow \text{Sentencia})$

Figura 7.7 Una gramática BNF (Backus-Naur Form) de sentencias en lógica proposicional.

Fíjese en que la gramática es muy estricta respecto al uso de los paréntesis: cada sentencia construida a partir de conectivas binarias debe estar encerrada en paréntesis. Esto asegura que la gramática no sea ambigua. También significa que tenemos que escribir, por ejemplo, $((A \wedge B) \Rightarrow C)$ en vez de $A \wedge B \Rightarrow C$. Para mejorar la legibilidad, a menudo omitiremos paréntesis, apoyándonos en un orden de precedencia de las conectivas. Es una precedencia similar a la utilizada en la aritmética (por ejemplo, $ab + c$ se lee $((ab) + c)$ porque la multiplicación tiene mayor precedencia que la suma). El orden de precedencia en la lógica proposicional (de mayor a menor) es: $\neg, \wedge, \vee, \Rightarrow$ y \Leftrightarrow . Así, la sentencia

$$\neg P \vee Q \wedge R \Rightarrow S$$

es equivalente a la sentencia

$$((\neg P) \vee (Q \wedge R)) \Rightarrow S$$

La precedencia entre las conectivas no resuelve la ambigüedad en sentencias como $A \wedge B \wedge C$, que se podría leer como $((A \wedge B) \wedge C)$ o como $(A \wedge (B \wedge C))$. Como estas dos lecturas significan lo mismo según la semántica que mostraremos en la siguiente sección, se permiten este tipo de sentencias. También se permiten sentencias como $A \vee B \vee C$ o $A \Leftrightarrow B \Leftrightarrow C$. Sin embargo, las sentencias como $A \Rightarrow B \Rightarrow C$ no se permiten, ya que su lectura en una dirección y su opuesta tienen significados muy diferentes; en este caso insistimos en la utilización de los paréntesis. Por último, a veces utilizaremos corchetes, en vez de paréntesis, para conseguir una lectura de la sentencia más clara.

Semántica

Una vez especificada la sintaxis de la lógica proposicional, vamos a definir su semántica. La semántica define las reglas para determinar el valor de verdad de una sentencia respecto a un modelo en concreto. En la lógica proposicional un modelo define el va-

lor de verdad (*verdadero* o *falso*). Por ejemplo, si las sentencias de la base de conocimiento utilizan los símbolos proposicionales $H_{1,2}$, $H_{2,2}$, y $H_{3,1}$, entonces un modelo posible sería

$$m_1 = \{H_{1,2} = \text{falso}, H_{2,2} = \text{falso}, H_{3,1} = \text{verdadero}\}$$

Con tres símbolos proposicionales hay $2^3 = 8$ modelos posibles, exactamente los que aparecen en la Figura 7.5. Sin embargo, fíjese en que gracias a que hemos concretado la sintaxis, los modelos se convierten en objetos puramente matemáticos sin tener necesariamente una conexión al mundo de *wumpus*. $H_{1,2}$ es sólo un símbolo, podría denotar tanto «hay un hoyo en la casilla [1, 2]», como «estaré en París hoy y mañana».

La semántica en lógica proposicional debe especificar cómo obtener el valor de verdad de *cualquier* sentencia, dado un modelo. Este proceso se realiza de forma recursiva. Todas las sentencias se construyen a partir de las sentencias atómicas y las cinco conectivas lógicas; entonces necesitamos establecer cómo definir el valor de verdad de las sentencias atómicas y cómo calcular el valor de verdad de las sentencias construidas con las cinco conectivas lógicas. Para las sentencias atómicas es sencillo:

- *Verdadero* es verdadero en todos los modelos y *Falso* es falso en todos los modelos.
- El valor de verdad de cada símbolo proposicional se debe especificar directamente para cada modelo. Por ejemplo, en el modelo anterior m_1 , $H_{1,2}$ es falso.

Para las sentencias complejas, tenemos reglas como la siguiente

- Para toda sentencia s y todo modelo m , la sentencia $\neg s$ es verdadera en m si y sólo si s es falsa en m .

Este tipo de reglas reducen el cálculo del valor de verdad de una sentencia compleja al valor de verdad de las sentencias más simples. Las reglas para las conectivas se pueden resumir en una **tabla de verdad** que especifica el valor de verdad de cada sentencia compleja según la posible asignación de valores de verdad realizada a sus componentes. En la Figura 7.8 se muestra la tabla de verdad de las cinco conectivas lógicas. Utilizando estas tablas de verdad, se puede obtener el valor de verdad de cualquier sentencia s según un modelo m mediante un proceso de evaluación recursiva muy sencillo. Por ejemplo, la sentencia $\neg H_{1,2} \wedge (H_{2,2} \vee H_{3,1})$ evaluada según m_1 , da *verdadero*.

TABLA DE VERDAD

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
<i>falso</i>	<i>falso</i>	<i>verdadero</i>	<i>falso</i>	<i>falso</i>	<i>verdadero</i>	<i>verdadero</i>
<i>falso</i>	<i>verdadero</i>	<i>verdadero</i>	<i>falso</i>	<i>verdadero</i>	<i>verdadero</i>	<i>falso</i>
<i>verdadero</i>	<i>falso</i>	<i>falso</i>	<i>falso</i>	<i>verdadero</i>	<i>falso</i>	<i>falso</i>
<i>verdadero</i>	<i>verdadero</i>	<i>falso</i>	<i>verdadero</i>	<i>verdadero</i>	<i>verdadero</i>	<i>verdadero</i>

Figura 7.8 Tablas de verdad para las cinco conectivas lógicas. Para utilizar la tabla, por ejemplo, para calcular el valor de $P \vee Q$, cuando P es verdadera y Q falso, primero mire a la izquierda en donde P es *verdadera* y Q es *falsa* (la tercera fila). Entonces mire en esa fila justo en la columna de $P \vee Q$ para ver el resultado: *verdadero*. Otra forma de verlo es pensar en cada fila como en un modelo, y que sus entradas en cada fila dicen para cada columna si la sentencia es verdadera en ese modelo.

$dero \wedge (falso \vee verdadero) = verdadero \wedge verdadero = verdadero$. El Ejercicio 7.3 pide que escriba el algoritmo $\text{¿V-VERDAD-LP?}(s, m)$ que debe obtener el valor de verdad de una sentencia s en lógica proposicional según el modelo m .

Ya hemos comentado que una base de conocimiento está compuesta por sentencias. Ahora podemos observar que esa base de conocimiento lógica es una conjunción de dichas sentencias. Es decir, si comenzamos con una BC vacía y ejecutamos $\text{DECIR}(BC, S_1) \dots \text{DECIR}(BC, S_n)$ entonces tenemos $BC = S_1 \wedge \dots \wedge S_n$. Esto significa que podemos manejar bases de conocimiento y sentencias de manera intercambiable.

Los valores de verdad de «y», «o» y «no» concuerdan con nuestra intuición, cuando los utilizamos en lenguaje natural. El principal punto de confusión puede presentarse cuando $P \vee Q$ es verdadero porque P lo es, Q lo es, o *ambos* lo son. Hay una conectiva diferente denominada «o exclusiva» («xor» para abreviar) que es falsa cuando los dos disyuntores son verdaderos⁹. No hay consenso respecto al símbolo que representa la o exclusiva, siendo las dos alternativas $\dot{\vee}$ y \oplus .

El valor de verdad de la conectiva \Rightarrow puede parecer incomprendible al principio, ya que no encaja en nuestra comprensión intuitiva acerca de « P implica Q » o de «si P entonces Q ». Para una cosa, la lógica proposicional no requiere de una relación de causalidad o relevancia entre P y Q . La sentencia «que 5 sea impar implica que Tokio es la capital de Japón» es una sentencia verdadera en lógica proposicional (bajo una interpretación normal), aunque pensándolo es, decididamente, una frase muy rara. Otro punto de confusión es que cualquier implicación es verdadera siempre que su antecedente sea falso. Por ejemplo, «que 5 sea par implica que Sam es astuto» es verdadera, independientemente de que Sam sea o no astuto. Parece algo extraño, pero tiene sentido si piensa acerca de « $P \Rightarrow Q$ » como si dijera, «si P es verdadero, entonces estoy afirmando que Q es verdadero. De otro modo, no estoy haciendo ninguna afirmación.» La única manera de hacer esta sentencia falsa es haciendo que P sea cierta y Q falsa.

La tabla de verdad de la bicondicional $P \Leftrightarrow Q$ muestra que la sentencia es verdadera siempre que $P \Rightarrow Q$ y $Q \Rightarrow P$ lo son. En lenguaje natural a menudo se escribe como « P si y sólo si Q » o « P si Q ». Las reglas del mundo de *wumpus* se describen mejor utilizando la conectiva \Leftrightarrow . Por ejemplo, una casilla tiene corriente de aire *si* alguna casilla vecina tiene un hoyo, y una casilla tiene corriente de aire *sólo si* una casilla vecina tiene un hoyo. De esta manera necesitamos bicondicionales como

$$B_{1,1} \Leftrightarrow (H_{1,2} \vee H_{2,1}),$$

en donde $B_{1,1}$ significa que hay una brisa en la casilla [1,1]. Fíjese en que la implicación

$$B_{1,1} \Rightarrow (H_{1,2} \vee H_{2,1})$$

es verdadera, aunque incompleta, en el mundo de *wumpus*. Esta implicación no descarta modelos en los que $B_{1,1}$ sea falso y $H_{1,2}$ sea verdadero, hecho que violaría las reglas del mundo de *wumpus*. Otra forma de observar esta incompletitud es que la implicación necesita la presencia de hoyos si hay una corriente de aire, mientras que la bicondicional además necesita la ausencia de hoyos si no hay ninguna corriente de aire.

⁹ En latín está la palabra específica *aut* para la o exclusiva.

Una base de conocimiento sencilla

Ahora que ya hemos definido la semántica de la lógica proposicional, podemos construir una base de conocimiento para el mundo de *wumpus*. Para simplificar, sólo trataremos con hechos y reglas acerca de hoyos; dejamos el tratamiento del *wumpus* como ejercicio. Vamos a proporcionar el conocimiento suficiente para llevar a cabo la inferencia que se trató en la Sección 7.3.

Primero de todo, necesitamos escoger nuestro vocabulario de símbolos proposicionales. Para cada i, j :

- Hacemos que $H_{i,j}$ sea verdadero si hay un hoyo en la casilla $[i, j]$.
- Hacemos que $B_{i,j}$ sea verdadero si hay una corriente de aire (una brisa) en la casilla $[i, j]$.

La base de conocimiento contiene, cada una etiquetada con un identificador, las siguientes sentencias:

- No hay ningún hoyo en la casilla $[1, 1]$.

$$R_1: \neg H_{1,1}$$

- En una casilla se siente una brisa si y sólo si hay un hoyo en una casilla vecina. Esta regla se ha de especificar para cada casilla; por ahora, tan sólo incluimos las casillas que son relevantes:

$$\begin{aligned} R_2: B_{1,1} &\Leftrightarrow (H_{1,2} \vee H_{2,1}) \\ R_3: B_{2,1} &\Leftrightarrow (H_{1,1} \vee H_{2,2} \vee H_{3,1}) \end{aligned}$$

- Las sentencias anteriores son verdaderas en todos los mundos de *wumpus*. Ahora incluimos las percepciones de brisa para las dos primeras casillas visitadas en el mundo concreto en donde se encuentra el agente, llegando a la situación que se muestra en la Figura 7.3(b).

$$\begin{aligned} R_4: \neg B_{1,1} \\ R_5: B_{2,1} \end{aligned}$$

Entonces, la base de conocimiento está compuesta por las sentencias R_1 hasta R_5 . La BC también se puede representar mediante una única sentencia (la conjunción $R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$) porque dicha sentencia aserta que todas las sentencias son verdaderas.

Inferencia

Recordemos que el objetivo de la inferencia lógica es decidir si $BC \models \alpha$ para alguna sentencia α . Por ejemplo, si se deduce $H_{2,2}$. Nuestro primer algoritmo para la inferencia será una implementación directa del concepto de implicación: enumerar los modelos, y averiguar si α es verdadera en cada modelo en el que la BC es verdadera. En la lógica proposicional los modelos son asignaciones de los valores *verdadero* y *falso* sobre cada símbolo proposicional. Volviendo a nuestro ejemplo del mundo de *wumpus*, los símbolos proposicionales relevantes son $B_{1,1}, B_{2,1}, H_{1,1}, H_{1,2}, H_{2,1}, H_{2,2}$ y $H_{3,1}$. Con es-

tos siete símbolos, tenemos $2^7 = 128$ modelos posibles; y en tres de estos modelos, la BC es verdadera (Figura 7.9). En esos tres modelos $\neg H_{1,2}$ es verdadera, por lo tanto, no hay un hoyo en la casilla [1, 2]. Por el otro lado, $H_{2,2}$ es verdadera en dos de esos tres modelos y falsa en el tercero, entonces todavía no podemos decir si hay un hoyo en la casilla [2, 2].

La Figura 7.9 reproduce más detalladamente el razonamiento que se mostraba en la Figura 7.5. En la Figura 7.10 se muestra un algoritmo general para averiguar la implicación en lógica proposicional. De forma similar al algoritmo de BÚSQUEDA-CON-BACKTRACKING de la página 86, ¿IMPLICACIÓN-EN-TV? Realiza una enumeración recursiva de un espacio finito de asignaciones a variables. El algoritmo es **sólido** porque implemen-

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	BC
falso	verdadero	verdadero	verdadero	verdadero	falso	falso						
falso	verdadero	verdadero	verdadero	verdadero	falso	falso						
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
falso	verdadero	falso	falso	falso	falso	falso	verdadero	verdadero	falso	verdadero	verdadero	falso
falso	verdadero	falso	falso	falso	falso	falso	verdadero	verdadero	verdadero	verdadero	verdadero	verdadero
falso	verdadero	falso	falso	falso	falso	falso	verdadero	verdadero	verdadero	verdadero	verdadero	verdadero
falso	verdadero	falso	falso	falso	falso	falso	verdadero	verdadero	verdadero	verdadero	verdadero	verdadero
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
verdadero	falso											

Figura 7.9 Una tabla de verdad construida para la base de conocimiento del ejemplo. La BC es verdadera si R_1 hasta R_5 son verdaderas, cosa que sucede en tres de las 128 filas. En estas tres filas, $H_{1,2}$ es falsa, así que no hay ningún hoyo en la casilla [1, 2]. Por otro lado, puede haber (o no) un hoyo en la casilla [2, 2].

función ¿IMPLICACIÓN-EN-TV?(BC, α) **devuelve** verdadero o falso

entradas: BC , la base de conocimiento, una sentencia en lógica proposicional α , la sentencia implicada, una sentencia en lógica proposicional

símbolos \leftarrow una lista de símbolos proposicionales de la BC y α

devuelve COMPROBAR-TV($BC, \alpha, \text{símbolos}, []$)

función COMPROBAR-TV($BC, \alpha, \text{símbolos}, \text{modelo}$) **devuelve** verdadero o falso

si ¿VACÍA?(símbolos) **entonces**

si ¿VERDADERO-LP?(BC, modelo) **entonces devuelve** ¿VERDADERO-LP?(α, modelo)

sino devuelve verdadero

sino hacer

$P \leftarrow \text{PRIMERO}(\text{símbolos}); resto \leftarrow \text{RESTO}(\text{símbolos})$

devuelve CHEQUEAR-TV($BC, \alpha, resto, \text{EXTENDER}(P, \text{verdadero}, \text{modelo})$) y
COMPROBAR-TV($BC, \alpha, resto, \text{EXTENDER}(P, \text{falso}, \text{modelo})$)

Figura 7.10 Un algoritmo de enumeración de una tabla de verdad para averiguar la implicación proposicional. TV viene de tabla de verdad. ¿VERDADERO-LP? Devuelve verdadero si una sentencia es verdadera en un modelo. La variable *modelo* representa un modelo parcial (una asignación realizada a un subconjunto de las variables). La llamada a la función EXTENDER(*P*, *verdadero*, *modelo*) devuelve un modelo parcial nuevo en el que *P* tiene el valor de verdad *verdadero*.

ta de forma directa la definición de implicación, y es **completo** porque trabaja para cualquier BC y sentencia α , y siempre finaliza (sólo hay un conjunto finito de modelos a ser examinados).

Por supuesto, que «conjunto finito» no siempre es lo mismo que «pequeño». Si la BC y α contienen en total n símbolos, entonces tenemos 2^n modelos posibles. Así, la complejidad temporal del algoritmo es $O(2^n)$. (La complejidad espacial sólo es $O(n)$ porque la enumeración es en primero en profundidad.) Más adelante, en este capítulo, veremos algoritmos que en la práctica son mucho más eficientes. Desafortunadamente, *cada algoritmo de inferencia que se conoce en lógica proposicional tiene un caso peor, cuya complejidad es exponencial respecto al tamaño de la entrada*. No esperamos mejorarlo, ya que demostrar la implicación en lógica proposicional es un problema co-NP-completo. (Véase Apéndice A.)



EQUIVALENCIA LÓGICA

Equivalencia, validez y satisfacibilidad

Antes de que nos sumerjamos en los detalles de los algoritmos de inferencia lógica necesitaremos algunos conceptos adicionales relacionados con la implicación. Al igual que la implicación, estos conceptos se aplican a todos los tipos de lógica, sin embargo, se entienden más fácilmente para una en concreto, como es el caso de la lógica proposicional.

El primer concepto es la **equivalencia lógica**: dos sentencias α y β son equivalentes lógicamente si tienen los mismos valores de verdad en el mismo conjunto de modelos. Este concepto lo representamos con $\alpha \Leftrightarrow \beta$. Por ejemplo, podemos observar fácilmente (mediante una tabla de verdad) que $P \wedge Q$ y $Q \wedge P$ son equivalentes lógicamente. En la Figura 7.11 se muestran otras equivalencias. Éstas juegan el mismo papel en la lógica que las igualdades en las matemáticas. Una definición alternativa de equivalencia es la siguiente: para dos sentencias α y β cualesquiera,

$$\alpha \equiv \beta \quad \text{si y sólo si} \quad \alpha \models \beta \text{ y } \beta \models \alpha$$

(Recuerde que \models significa implicación.)

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	Commutatividad de \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	Commutatividad de \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	Asociatividad de \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	Asociatividad de \vee
$\neg(\neg\alpha) \equiv \alpha$	Eliminación de la doble negación
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	Contraposición
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	Eliminación de la implicación
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	Eliminación de la bicondicional
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	Ley de Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	Ley de Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	Distribución de \wedge respecto a \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	Distribución de \vee respecto a \wedge

Figura 7.11 Equivalencias lógicas. Los símbolos α , β y γ se pueden sustituir por cualquier sentencia en lógica proposicional.

VALIDEZ

TAUTOLOGÍA

TEOREMA DE LA DEDUCCIÓN



SATISFACIBILIDAD

SATISFACE

REDUCTIO AD ABSURDUM

REFUTACIÓN

El segundo concepto que necesitaremos es el de **validez**. Una sentencia es válida si es verdadera en *todos* los modelos. Por ejemplo, la sentencia $P \vee \neg P$ es una sentencia válida. Las sentencias válidas también se conocen como **tautologías**, son *necesariamente* verdaderas y por lo tanto vacías de significado. Como la sentencia *Verdadero* es verdadera en todos los modelos, toda sentencia válida es lógicamente equivalente a *Verdadero*.

¿Qué utilidad tienen las sentencias válidas? De nuestra definición de implicación podemos derivar el **teorema de la deducción**, que ya se conocía por los Griegos antiguos:

Para cualquier sentencia α y β , $\alpha \models \beta$ si y sólo si la sentencia $(\alpha \Rightarrow \beta)$ es válida.

(En el Ejercicio 7.4 se pide demostrar una serie de aserciones.) Podemos pensar en el algoritmo de inferencia de la Figura 7.10 como en un proceso para averiguar la validez de $(BC \Rightarrow \alpha)$. A la inversa, cada sentencia que es una implicación válida representa una inferencia correcta.

El último concepto que necesitaremos es el de **satisfacibilidad**. Una sentencia es satisfactoria si es verdadera para *algún* modelo. Por ejemplo, en la base de conocimiento ya mostrada, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$ es *satisfacible* porque hay tres modelos en los que es verdadera, tal como se muestra en la Figura 7.9. Si una sentencia α es verdadera en un modelo m , entonces decimos que m **satisface** α , o que m es **un modelo de** α . La *satisfacibilidad* se puede averiguar enumerando los modelos posibles hasta que uno satisface la sentencia. La determinación de la *satisfacibilidad* de sentencias en lógica proposicional fue el primer problema que se demostró que era NP-completo.

Muchos problemas en las ciencias de la computación son en realidad problemas de *satisfacibilidad*. Por ejemplo, todos los problemas de satisfacción de restricciones del Capítulo 5 se preguntan esencialmente si un conjunto de restricciones se satisfacen dada una asignación. Con algunas transformaciones adecuadas, los problemas de búsqueda también se pueden resolver mediante *satisfacibilidad*. La validez y la *satisfacible* están íntimamente relacionadas: α es válida si y sólo si $\neg\alpha$ es *insatisfacible*; en contraposición, α es *satisfacible* si y sólo si $\neg\alpha$ no es válida.

$\alpha \models \beta$ si y sólo si la sentencia $(\alpha \wedge \neg\beta)$ es *insatisfactoria*.

La demostración de β a partir de α averiguando la insatisfacibilidad de $(\alpha \wedge \neg\beta)$ se corresponde exactamente con la técnica de demostración en matemáticas de la *reductio ad absurdum* (que literalmente se traduce como «reducción al absurdo»). Esta técnica también se denomina demostración mediante **refutación** o demostración por **contradicción**. Asumimos que la sentencia β es falsa y observamos si se llega a una contradicción con las premisas en α . Dicha contradicción es justamente lo que queremos expresar cuando decimos que la sentencia $(\alpha \wedge \neg\beta)$ es *insatisfacible*.



7.5 Patrones de razonamiento en lógica proposicional

Esta sección cubre los patrones estándar de inferencia que se pueden aplicar para derivar cadenas de conclusiones que nos llevan al objetivo deseado. Estos patrones de infe-

rencia se denominan **reglas de inferencia**. La regla más conocida es la llamada **Modus Ponens** que se escribe como sigue:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

La notación nos dice que, cada vez que encontramos dos sentencias en la forma $\alpha \Rightarrow \beta$ y α , entonces la sentencia β puede ser inferida. Por ejemplo, si tenemos $(WumpusEnFrente \wedge WumpusVivo) \Rightarrow Disparar$ y $(WumpusEnFrente \wedge WumpusVivo)$, entonces se puede inferir *Disparar*.

Otra regla de inferencia útil es la **Eliminación- \wedge** , que expresa que, de una conjunción se puede inferir cualquiera de sus conjuntores:

$$\frac{\alpha \wedge \beta}{\alpha}$$

Por ejemplo, de $(WumpusEnFrente \wedge WumpusVivo)$, se puede inferir *WumpusVivo*.

Teniendo en cuenta los posibles valores de verdad de α y β se puede observar fácilmente, de una sola vez, que el Modus Ponens y la Eliminación- \wedge son reglas sólidas. Estas reglas se pueden utilizar sobre cualquier instancia en la que es aplicable, generando inferencias sólidas, sin la necesidad de enumerar todos los modelos.

Todas las equivalencias lógicas de la Figura 7.11 se pueden utilizar como reglas de inferencia. Por ejemplo, la equivalencia de la eliminación de la bicondicional nos lleva a las dos reglas de inferencia

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{y} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Pero no todas las reglas de inferencia se pueden usar, como ésta, en ambas direcciones. Por ejemplo, no podemos utilizar el Modus Ponens en la dirección opuesta para obtener $\alpha \Rightarrow \beta$ y α a partir de β .

Veamos cómo se pueden usar estas reglas de inferencia y equivalencias en el mundo de *wumpus*. Comenzamos con la base de conocimiento contenido en R_1 a R_5 , y mostramos cómo demostrar $\neg H_{1,2}$, es decir, que no hay un hoyo en la casilla [1, 2]. Primero aplicamos la eliminación de la bicondicional a R_2 para obtener

$$R_6: (B_{1,1} \Rightarrow (H_{1,2} \vee H_{2,1})) \wedge ((H_{1,2} \vee H_{2,1}) \Rightarrow B_{1,1})$$

Entonces aplicamos la Eliminación- \wedge a R_6 para obtener

$$R_7: ((H_{1,2} \vee H_{2,1}) \Rightarrow B_{1,1})$$

Y por la equivalencia lógica de contraposición obtenemos

$$R_8: (\neg B_{1,1} \Rightarrow \neg(H_{1,2} \vee H_{2,1}))$$

Ahora aplicamos el Modus Ponens con R_8 y la percepción R_4 (por ejemplo, $\neg B_{1,1}$), para obtener

$$R_9: \neg(H_{1,2} \vee H_{2,1})$$

Finalmente, aplicamos la ley de Morgan, obteniendo la conclusión

$$R_{10}: \neg H_{1,2} \wedge \neg H_{2,1}$$

Es decir, ni la casilla [1, 2] ni la [2, 1] contienen un hoyo.

PRUEBA

A la derivación que hemos realizado (una secuencia de aplicaciones de reglas de inferencia) se le denomina una **prueba** (o **demonstración**). Obtener una prueba es muy semejante a encontrar una solución en un problema de búsqueda. De hecho, si la función sucesor se define para generar todas las aplicaciones posibles de las reglas de inferencia, entonces todos los algoritmos de búsqueda del Capítulo 3 se pueden utilizar para obtener una prueba. De esta manera, la búsqueda de pruebas es una alternativa a tener que enumerar los modelos. La búsqueda se puede realizar hacia delante a partir de la base de conocimiento inicial, aplicando las reglas de inferencia para derivar la sentencia objetivo, o hacia atrás, desde la sentencia objetivo, intentando encontrar una cadena de reglas de inferencia que nos lleven a la base de conocimiento inicial. Más adelante, en esta sección, veremos dos familias de algoritmos que utilizan estas técnicas.

 El hecho de que la inferencia en lógica proposicional sea un problema NP-completo nos hace pensar que, en el peor de los casos, la búsqueda de pruebas va a ser no mucho más eficiente que la enumeración de modelos. Sin embargo, en muchos casos prácticos, *encontrar una prueba puede ser altamente eficiente simplemente porque el proceso puede ignorar las proposiciones irrelevantes, sin importar cuántas de éstas haya*. Por ejemplo, la prueba que hemos visto que nos llevaba a $\neg H_{1,2} \wedge \neg H_{2,1}$ no utiliza las proposiciones $B_{2,1}$, $H_{1,1}$, $H_{2,2}$ o $H_{3,1}$. Estas proposiciones se pueden ignorar porque la proposición objetivo $H_{1,2}$ sólo aparece en la sentencia R_4 , y la otra proposición de R_4 sólo aparece también en R_2 ; por lo tanto, R_1 , R_3 y R_5 no juegan ningún papel en la prueba. Sucedería lo mismo aunque añadiésemos un millón de sentencias a la base de conocimiento; por el otro lado, el algoritmo de la tabla de verdad, aunque sencillo, quedaría saturado por la explosión exponencial de los modelos.

MONÓTONO

Esta propiedad de los sistemas lógicos en realidad proviene de una característica mucho más fundamental, denominada **monótono**. La característica de monotonismo nos dice que el conjunto de sentencias implicadas sólo puede *aumentar* (pero no cambiar) al añadirse información a la base de conocimiento¹⁰. Para cualquier sentencia α y β ,

$$\text{si } BC \models \alpha \text{ entonces } BC \wedge \beta \models \alpha$$

Por ejemplo, supongamos que la base de conocimiento contiene una asercción adicional β , que nos dice que hay exactamente ocho hoyos en el escenario. Este conocimiento podría ayudar al agente a obtener conclusiones *adicionales*, pero no puede invalidar ninguna conclusión α ya inferida (como la conclusión de que no hay un hoyo en la casilla [1, 2]). El monotonismo permite que las reglas de inferencia se puedan aplicar siempre que se hallen premisas aplicables en la base de conocimiento; la conclusión de la regla debe permanecer *sin hacer caso de qué más hay en la base de conocimiento*.

¹⁰ Las lógicas **No Monótonas**, que violan la propiedad de monotonismo, modelan una característica propia del razonamiento humano: cambiar de opinión. Estas lógicas se verán en la Sección 10.7.

Resolución

Hemos argumentado que las reglas de inferencia vistas hasta aquí son *sólidas*, pero no hemos visto la cuestión acerca de lo *completo* de los algoritmos de inferencia que las utilizan. Los algoritmos de búsqueda como el de búsqueda en profundidad iterativa (página 87) son completos en el sentido de que éstos encontrarán cualquier objetivo alcanzable, pero si las reglas de inferencia no son adecuadas, entonces el objetivo no es alcanzable; no existe una prueba que utilice sólo esas reglas de inferencia. Por ejemplo, si suprimimos la regla de eliminación de la bicondicional la prueba de la sección anterior no avanzaría. En esta sección se introduce una regla de inferencia sencilla, la **resolución**, que nos lleva a un algoritmo de inferencia completo cuando se empareja a un algoritmo de búsqueda completo.

Comenzaremos utilizando una versión sencilla de la resolución aplicada al mundo de *wumpus*. Consideremos los pasos que nos llevaban a la Figura 7.4(a): el agente vuelve de la casilla [2, 1] a la [1, 1] y entonces va a la casilla [1, 2], donde percibe un hedor, pero no percibe una corriente de aire. Ahora añadimos los siguientes hechos a la base de conocimiento:

$$\begin{aligned} R_{11}: \quad & \neg B_{1,2} \\ R_{12}: \quad & B_{1,2} \Leftrightarrow (H_{1,1} \vee H_{2,2} \vee H_{1,3}) \end{aligned}$$

Mediante el mismo proceso que nos llevó antes a R_{10} , podemos derivar que no hay ningún hoyo en la casilla [2, 2] o en la [1, 3] (recuerde que se sabe que en la casilla no había ninguna percepción de hoyos):

$$\begin{aligned} R_{13}: \quad & \neg H_{2,2} \\ R_{14}: \quad & \neg H_{1,3} \end{aligned}$$

También podemos aplicar la eliminación de la bicondicional a la R_3 , seguido del Modus Ponens con la R_5 , para obtener el hecho de que puede haber un hoyo en la casilla [1, 1], la [2, 2] o la [3, 1]:

$$R_{15}: \quad H_{1,1} \vee H_{2,2} \vee H_{3,1}$$

Ahora viene la primera aplicación de la regla de resolución: el literal $\neg H_{2,2}$ de la R_{13} se *resuelve con* el literal $H_{2,2}$ de la R_{15} , dando el *resolvente*

$$R_{16}: \quad H_{1,1} \vee H_{3,1}$$

En lenguaje natural: si hay un hoyo en la casilla [1, 1], o en la [2, 2], o en la [3, 1], y no hay ninguno en la [2, 2], entonces hay uno en la [1, 1] o en la [3, 1]. De forma parecida, el literal $\neg H_{1,1}$ de la R_1 se resuelve con el literal $H_{1,1}$ de la R_{16} , dando

$$R_{17}: \quad H_{3,1}$$

RESOLUCIÓN
UNITARIALITERALES
COMPLEMENTARIOS

CLÁUSULA

CLÁUSULA UNITARIA

FACTORIZACIÓN



En lenguaje natural: si hay un hoyo en la casilla [1, 1] o en la [3, 1], y no hay ninguno en la [1, 1], entonces hay uno en la [3, 1]. Los últimos dos pasos de inferencia son ejemplo de la regla de inferencia de **resolución unitaria**,

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k}$$

en donde cada ℓ es un literal y ℓ_i y m son **literales complementarios** (por ejemplo, uno es la negación del otro). Así, la resolución unitaria toma una **cláusula** (una disyunción de literales) y un literal para producir una nueva cláusula. Fíjese en que un literal se puede ver como una disyunción con un solo literal, conocido como **cláusula unitaria**.

La regla de resolución unitaria se puede generalizar a la regla general de **resolución**,

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m_1 \vee \dots \vee m_n}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}$$

donde ℓ_i y m_j son literales complementarios. Si sólo tratáramos con cláusulas de longitud dos, podríamos escribir la regla así

$$\frac{\ell_1 \vee \ell_2, \quad \neg \ell_2 \vee \ell_3}{\ell_1 \vee \ell_3}$$

Es decir, la resolución toma dos cláusulas y genera una cláusula nueva con los literales de las dos cláusulas originales *menos* los literales complementarios. Por ejemplo, tendríamos

$$\frac{P_{1,1} \vee P_{3,1}, \quad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}$$

Hay otro aspecto técnico relacionado con la regla de resolución: la cláusula resultante debería contener sólo una copia de cada literal¹¹. Se le llama **factorización** al proceso de eliminar las copias múltiples de los literales. Por ejemplo, si resolvemos $(A \vee B)$ con $(A \vee \neg B)$ obtenemos $(A \vee A)$, que se reduce a A .

La *sólitez* de la regla de resolución se puede ver fácilmente si consideramos el literal ℓ_i . Si ℓ_i es verdadero, entonces m_j es falso, y de aquí $m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n$ debe ser verdadero, porque se da $m_1 \vee \dots \vee m_n$. Si ℓ_i es falso, entonces $\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k$ debe ser verdadero, porque se da $\ell_1 \vee \dots \vee \ell_k$. Entonces ℓ_i es o bien verdadero o bien falso, y así, se obtiene una de las dos conclusiones, exactamente tal cómo establece la regla de resolución.

Lo que es más sorprendente de la regla de resolución es que crea la base para una familia de procedimientos de inferencia *completos*. *Cualquier algoritmo de búsqueda completa, aplicando sólo la regla de resolución, puede derivar cualquier conclusión implicada por cualquier base de conocimiento en lógica proposicional*. Pero hay una advertencia: la resolución es completa en un sentido muy especializado. Dado que A sea

¹¹ Si una cláusula se ve como un conjunto de literales, entonces esta restricción se respeta de forma automática. Utilizar la notación de conjuntos para representar cláusulas hace que la regla de resolución sea más clara, con el coste de introducir una notación adicional.

verdadero, no podemos utilizar la resolución para generar de forma automática la consecuencia $A \vee B$. Sin embargo, podemos utilizar la resolución para responder a la pregunta de si $A \vee B$ es verdadero. Este hecho se denomina **completitud de la resolución**, que indica que la resolución se puede utilizar siempre para confirmar o refutar una sentencia, pero no se puede usar para enumerar sentencias verdaderas. En las dos siguientes secciones explicamos cómo la resolución lleva a cabo este proceso.

Forma normal conjuntiva

La regla de resolución sólo se puede aplicar a disyunciones de literales, por lo tanto, sería muy importante que la base de conocimiento y las preguntas a ésta estén formadas por disyunciones. Entonces, ¿cómo nos lleva esto a un procedimiento de inferencia completa para toda la lógica proposicional? La respuesta es que *toda sentencia en lógica proposicional es equivalente lógicamente a una conjunción de disyunciones de literales*. Una sentencia representada mediante una conjunción de disyunciones de literales se dice que está en **forma normal conjuntiva** o FNC. Que lo consideraremos bastante útil más tarde, al tratar la reducida familia de sentencias **k -FNC**. Una sentencia k -FNC tiene exactamente k literales por cláusula:

$$(\ell_{1,1} \vee \dots \vee \ell_{1,k}) \wedge \dots \wedge (\ell_{n,1} \vee \dots \vee \ell_{n,k})$$

De manera que se puede transformar cada sentencia en una sentencia de tipo 3-FNC, la cual tiene un conjunto de modelos equivalente.

Mejor que demostrar estas afirmaciones (véase el Ejercicio 7.10), vamos a describir un procedimiento de conversión muy sencillo. Vamos a ilustrar el procedimiento con la conversión de R_2 , la sentencia $B_{1,1} \Leftrightarrow (H_{1,2} \vee H_{2,1})$, a FNC. Los pasos a seguir son los siguientes:

1. Eliminar \Leftrightarrow , sustituyendo $\alpha \Leftrightarrow \beta$ por $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (H_{1,2} \vee H_{2,1})) \wedge ((H_{1,2} \vee H_{2,1}) \Rightarrow B_{1,1})$$

2. Eliminar \Rightarrow , sustituyendo $\alpha \Rightarrow \beta$ por $\neg\alpha \vee \beta$.

$$(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1}) \wedge (\neg(H_{1,2} \vee H_{2,1}) \vee B_{1,1})$$

3. Una FNC requiere que la \neg se aplique sólo a los literales, por lo tanto, debemos «anidar las \neg » mediante la aplicación reiterada de las siguientes equivalencias (sacadas de la Figura 7.11).

$$\neg(\neg\alpha) \equiv \alpha \text{ (eliminación de la doble negación)}$$

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \text{ (de Morgan)}$$

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \text{ (de Morgan)}$$

En el ejemplo, sólo necesitamos una aplicación de la última regla:

$$(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1}) \wedge ((\neg H_{1,2} \wedge \neg H_{2,1}) \vee B_{1,1})$$



FORMA NORMAL CONJUNTIVA

k -FNC

4. Ahora tenemos una sentencia que tiene una \wedge con operadores de \vee anidados, aplicados a literales y a una \wedge anidada. Aplicamos la ley de distributividad de la Figura 7.11, distribuyendo la \vee sobre la \wedge cuando nos es posible.

$$(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1}) \wedge (\neg H_{1,2} \vee B_{1,1}) \wedge (\neg H_{2,1} \vee B_{1,1})$$

La sentencia inicial ahora está en FNC, una conjunción con tres cláusulas. Es más difícil de leer pero se puede utilizar como entrada en el procedimiento de resolución.

Un algoritmo de resolución

Los procedimientos de inferencia basados en la resolución trabajan utilizando el principio de prueba mediante contradicción que vimos al final de la Sección 7.4. Es decir, para demostrar que $BC \models \alpha$, demostramos que $(BC \wedge \neg\alpha)$ es *insatisfacible*. Lo hacemos demostrando una contradicción.

En la Figura 7.12 se muestra un algoritmo de resolución. Primero se convierte $(BC \wedge \neg\alpha)$ a FNC. Entonces, se aplica la regla de resolución a las cláusulas obtenidas. Cada par que contiene literales complementarios se resuelve para generar una nueva cláusula, que se añade al conjunto de cláusulas si no estaba ya presente. El proceso continúa hasta que sucede una de estas dos cosas:

- No hay nuevas cláusulas que se puedan añadir, en cuyo caso α no implica β , o
- Se deriva la cláusula vacía de una aplicación de la regla de resolución, en cuyo caso α implica β .

La cláusula vacía (una disyunción sin disyuntores) es equivalente a *Falso* porque una disyunción es verdadera sólo si al menos uno de sus disyuntores es verdadero. Otra forma de ver que la cláusula vacía representa una contradicción es observar que se presenta sólo si se resuelven dos cláusulas unitarias complementarias, tales como P y $\neg P$.

```

función RESOLUCIÓN-LP( $BC, \alpha$ ) devuelve verdadero o falso
entradas:  $BC$ , la base de conocimiento, una sentencia en lógica proposicional
 $\alpha$ , la petición, una sentencia en lógica proposicional
cláusulas  $\leftarrow$  el conjunto de cláusulas de  $BC \wedge \neg\alpha$  en representación FNC
nueva  $\leftarrow \{ \}$ 
bucle hacer
  para cada  $C_i, C_j$  en cláusulas hacer
    resolventes  $\leftarrow$  RESUELVE-LP( $C_i, C_j$ )
    si resolventes contiene la cláusula vacía entonces devolver verdadero
    nueva  $\leftarrow$  nueva  $\cup$  resolventes
  si nueva  $\subseteq$  cláusulas entonces devolver falso
  cláusulas  $\leftarrow$  cláusulas  $\cup$  nueva

```

Figura 7.12 Un algoritmo sencillo de resolución para la lógica proposicional. La función RESUELVE-LP devuelve el conjunto de todas las cláusulas posibles que se obtienen de resolver las dos entradas.

Ahora podemos aplicar el procedimiento de resolución a una inferencia sencilla del mundo de *wumpus*. Cuando el agente está en la casilla [1, 1] no percibe ninguna brisa, por lo tanto no puede haber hoyos en las casillas vecinas. Las sentencias relevantes en la base de conocimiento son

$$BC = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (H_{1,2} \vee H_{2,1})) \wedge \neg B_{1,1}$$

y deseamos demostrar α , es decir $\neg H_{1,2}$. Cuando convertimos $(BC \wedge \neg \alpha)$ a FNC obtenemos las cláusulas que se muestran en la fila superior de la Figura 7.13. La segunda fila en la figura muestra todas las cláusulas obtenidas resolviendo parejas de la primera fila. Entonces, cuando $H_{1,2}$ se resuelve con $\neg H_{1,2}$ obtenemos la cláusula vacía, representada mediante un cuadrado pequeño. Una revisión de la Figura 7.13 nos revela que muchos pasos de resolución no nos sirven de nada. Por ejemplo, la cláusula $B_{1,1} \vee \neg B_{1,1} \vee H_{1,2}$ es equivalente a *Verdadero* $\vee H_{1,2}$, que es también equivalente a *Verdadero*. Deducir que *Verdadero* es verdadero no nos es muy útil. Por lo tanto, se puede descartar cualquier cláusula que contenga dos literales complementarios.

Completitud de la resolución

Para concluir con nuestro debate acerca de la resolución, ahora vamos a demostrar por qué es completo el procedimiento RESOLUCIÓN-LP. Para hacerlo nos vendrá bien introducir el concepto de **cierre de la resolución** $CR(S)$ del conjunto de cláusulas S , que es el conjunto de todas las cláusulas derivables, obtenidas mediante la aplicación repetida de la regla de resolución a las cláusulas de S o a las derivadas de éstas. El cierre de la resolución es lo que calcula el procedimiento RESOLUCIÓN-LP y asigna como valor final a la variable *cláusulas*. Es fácil ver que $CR(S)$ debe ser finito, porque sólo hay un conjunto finito de las diferentes cláusulas que se pueden generar a partir del conjunto de símbolos P_1, \dots, P_k que aparecen en S , (fíjese que esto no sería cierto si no aplicáramos el procedimiento de factorización, que elimina las copias múltiples de un literal). Por eso, el procedimiento RESOLUCIÓN-LP siempre termina.

CIERRE DE LA RESOLUCIÓN

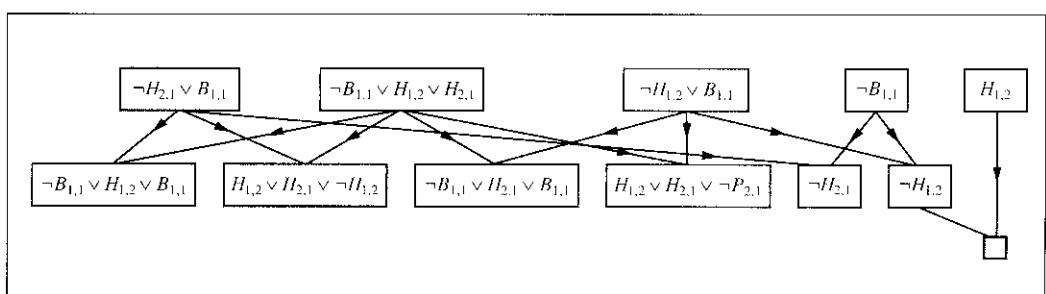


Figura 7.13 Aplicación parcial de RESOLUCIÓN-LP a una inferencia sencilla en el mundo de *wumpus*. Se observa que $\neg H_{1,2}$ se sigue de las cláusulas 3.^a y 4.^a de la fila superior.

TEOREMA FUNDAMENTAL DE LA RESOLUCIÓN

El teorema de la completitud para la resolución en lógica proposicional se denomina **teorema fundamental de la resolución**:

Si un conjunto de cláusulas es *insatisfacible*, entonces el cierre de la resolución de esas cláusulas contiene la cláusula vacía.

Vamos a probar este teorema demostrando su contraposición: si el cierre $CR(S)$ no contiene la cláusula vacía, entonces S es *satisfacible*. De hecho, podemos construir un modelo de S con los valores de verdad adecuados para P_1, \dots, P_k . El procedimiento de construcción es como sigue:

Para i de 1 a k ,

- Si hay una cláusula en $CR(S)$ que contenga el literal $\neg P_i$, tal que todos los demás literales de la cláusula sean falsos bajo la asignación escogida para P_1, \dots, P_{i-1} , entonces asignar a P_i el valor de *falso*.
- En otro caso, asignar a P_i el valor de *verdadero*.

Queda por demostrar que esta asignación a P_1, \dots, P_k es un modelo de S , a condición de que $CR(S)$ se cierre bajo la resolución y no contenga la cláusula vacía. Esta demostración se deja como ejercicio.

Encadenamiento hacia delante y hacia atrás

La completitud de la resolución hace que ésta sea un método de inferencia muy importante. Sin embargo, en muchos casos prácticos no se necesita todo el poder de la resolución. Las bases de conocimiento en el mundo real a menudo contienen sólo cláusulas, de un tipo restringido, denominadas **cláusulas de Horn**. Una cláusula de Horn es una disyunción de literales de los cuales, *como mucho uno es positivo*. Por ejemplo, la cláusula $(\neg L_{1,1} \vee \neg Brisa \vee B_{1,1})$, en donde $L_{1,1}$ representa que el agente está en la casilla [1, 1], es una cláusula de Horn, mientras que la cláusula $(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1})$ no lo es.

La restricción de que haya sólo un literal positivo puede parecer algo arbitraria y sin interés, pero realmente es muy importante, debido a tres razones:

1. Cada cláusula de Horn se puede escribir como una implicación cuya premisa sea una conjunción de literales positivos y cuya conclusión sea un único literal positivo. (Véase el Ejercicio 7.12.) Por ejemplo, la cláusula de Horn $(\neg L_{1,1} \vee \neg Brisa \vee B_{1,1})$ se puede describir como la implicación $(L_{1,1} \wedge Brisa) \Rightarrow B_{1,1}$. La sentencia es más fácil de leer en la última representación: ésta dice que si el agente está en la casilla [1, 1] y percibe una brisa, entonces la casilla [1, 1] tiene una corriente de aire. La gente encuentra más fácil esta forma de leer y escribir sentencias para muchos dominios del conocimiento.

Las cláusulas de Horn como ésta, con *exactamente* un literal positivo, se denominan **cláusulas positivas**. El literal positivo se denomina **cabeza**, y la disyunción de literales negativos **cuerpo** de la cláusula. Una cláusula positiva que no tiene literales negativos simplemente aserta una proposición dada, que algunas veces se le denomina **hecho**. Las cláusulas positivas forman la base de la

CLÁUSULAS DE HORN

CLÁUSULAS POSITIVAS

CABEZA

CUERPO

HECHO

RESTRICCIONES DE INTEGRIDAD

ENCADENAMIENTO HACIA DELANTE

ENCADENAMIENTO HACIA ATRÁS

GRAFO Y-O

PUNTO FIJO

programación lógica, que se verá en el Capítulo 9. Una cláusula de Horn *sin* literales positivos se puede escribir como una implicación cuya conclusión es el literal *Falso*. Por ejemplo, la cláusula $(\neg W_{1,1} \vee \neg W_{1,2})$ (*el wumpus* no puede estar en la casilla [1, 1] y la [1, 2] a la vez) es equivalente a $W_{1,1} \wedge W_{1,2} \Rightarrow \text{Falso}$. A este tipo de sentencias se las llama **restricciones de integridad** en el mundo de las bases de datos, donde se utilizan para indicar errores entre los datos. En los algoritmos siguientes asumimos, para simplificar, que la base de conocimiento sólo contiene cláusulas positivas y que no dispone de restricciones de integridad. Entonces decimos que estas bases de conocimiento están en forma de Horn.

2. La inferencia con cláusulas de Horn se puede realizar mediante los algoritmos de **encadenamiento hacia delante** y de **encadenamiento hacia atrás**, que en breve explicaremos. Ambos algoritmos son muy naturales, en el sentido de que los pasos de inferencia son obvios y fáciles de seguir por las personas.
3. Averiguar si hay o no implicación con las cláusulas de Horn se puede realizar en un tiempo que es *lineal* respecto al tamaño de la base de conocimiento.

Este último hecho es una grata sorpresa. Esto significa que la inferencia lógica es un proceso barato para muchas bases de conocimiento en lógica proposicional que se encuentran en el mundo real.

El algoritmo de encadenamiento hacia delante ¿IMPLICACIÓN-EHD-LP?(BC, q) determina si un símbolo proposicional q (la petición) se deduce de una base de conocimiento compuesta por cláusulas de Horn. El algoritmo comienza a partir de los hechos conocidos (literales positivos) de la base de conocimiento. Si todas las premisas de una implicación se conocen, entonces la conclusión se añade al conjunto de hechos conocidos. Por ejemplo, si $L_{1,1}$ y *Brisa* se conocen y $(L_{1,1} \wedge \text{Brisa}) \Rightarrow B_{1,1}$ está en la base de conocimiento, entonces se puede añadir $B_{1,1}$ a ésta. Este proceso continúa hasta que la petición q es añadida o hasta que no se pueden realizar más inferencias. En la Figura 7.14 se muestra el algoritmo detallado. El principal punto a recordar es que el algoritmo se ejecuta en tiempo lineal.

La mejor manera de entender el algoritmo es mediante un ejemplo y un diagrama. La Figura 7.15(a) muestra una base de conocimiento sencilla con cláusulas de Horn, en donde A y B se conocen como hechos. La Figura 7.15(b) muestra la misma base de conocimiento representada mediante un **grafo Y-O**. En los grafos Y-O múltiples enlaces se juntan mediante un arco para indicar una disyunción (cualquier enlace se puede probar). Es fácil ver cómo el encadenamiento hacia delante trabaja sobre el grafo. Se seleccionan los hechos conocidos (aquí A y B) y la inferencia se propaga hacia arriba tanto como se pueda. Siempre que aparece una conjunción, la propagación se para hasta que todos los conjuntos sean conocidos para seguir a continuación. Se anima al lector a que desarrolle el proceso en detalle a partir del ejemplo.

Es fácil descubrir que el encadenamiento hacia delante es un proceso **sólido**: cada inferencia es esencialmente una aplicación del Modus Ponens. El encadenamiento hacia delante también es **completo**: cada sentencia atómica implicada será derivada. La forma más fácil de verlo es considerando el estado final de la tabla *inferido* (después de que el algoritmo encuentra un **punto fijo** a partir del cual no es posible realizar nuevas inferencias).

función *¿IMPLICACIÓN-EHD-LP?*(*BC*, *q*) **devuelve** verdadero o falso

entradas: *BC*, la base de conocimiento, un conjunto de cláusulas de Horn en Lógica Proposicional
q, la petición, un símbolo proposicional

variables locales: *cuenta*, una tabla ordenada por cláusula, inicializada al número de cláusulas
inferido, una tabla, ordenada por símbolo, cada entrada inicializada a *falso*
agenda, una lista de símbolos, inicializada con los símbolos de la *BC* que se sabe que son verdaderos

mientras *agenda* no esté vacía **hacer**

p \leftarrow POP(*agenda*)

a menos que *inferido*[*p*] **hacer**

inferido[*p*] \leftarrow verdadero

para cada cláusula de Horn *c* en la que aparezca la premisa *p* **hacer**

 reducir *cuenta*[*c*]

si *cuenta*[*c*] = 0 **entonces hacer**

si CABEZA[*c*] = *q* **entonces devolver** verdadero

 PUSH(CABEZA[*c*], *agenda*)

devolver falso

Figura 7.14 El algoritmo de encadenamiento hacia delante para la lógica proposicional. La variable *agenda* almacena la pista de los símbolos que se saben son verdaderos pero no han sido «procesados» todavía. La tabla *cuenta* guarda la pista de las premisas de cada implicación que aún son desconocidas. Siempre que se procesa un símbolo *p* de la agenda la cuenta se reduce en uno para cada implicación en la que aparece la premisa *p*. (Este proceso se puede realizar en un tiempo constante si la *BC* se ordena de forma adecuada.) Si la cuenta llega a cero, todas las premisas de la implicación son conocidas, y por tanto, la conclusión se puede añadir a la agenda. Por último, necesitamos guardar la pista de qué símbolos han sido procesados; no se necesita añadir un símbolo *inferido* si ha sido procesado previamente. Este proceso nos evita un trabajo redundante, y también nos prevé de los bucles infinitos que podrían causarse por implicaciones tales como $P \Rightarrow Q$ y $Q \Rightarrow P$.

$$P \Rightarrow Q$$

$$L \wedge M \Rightarrow P$$

$$B \wedge L \Rightarrow M$$

$$A \wedge P \Rightarrow L$$

$$A \wedge B \Rightarrow L$$

$$A$$

$$B$$

(a)

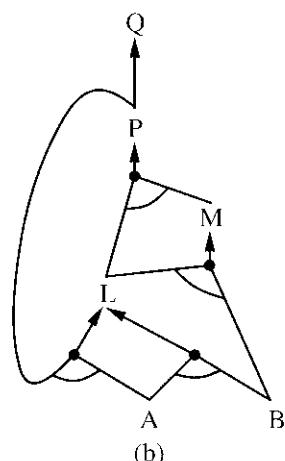


Figura 7.15 (a) Una base de conocimiento sencilla con cláusulas de Horn. (b) Su correspondiente grafo Y-O.

 La tabla contiene el valor *verdadero* para cada símbolo inferido en el proceso, y el valor *falso* para los demás símbolos. Podemos interpretar la tabla como un modelo lógico, más aún, *cada cláusula positiva de la BC original es verdadera en este modelo*. Para ver esto asumamos lo opuesto, en concreto, que alguna cláusula $a_1 \wedge \dots \wedge a_k \Rightarrow b$ sea falsa en el modelo. Entonces $a_1 \wedge \dots \wedge a_k$ debe ser verdadero en el modelo y b debe ser falso. ¡Pero esto contradice nuestra asunción de que el algoritmo ha encontrado un punto fijo! Por lo tanto, podemos concluir que el conjunto de sentencias atómicas inferidas hasta el punto fijo define un modelo de la *BC* original. Además, cualquier sentencia atómica q que se implica de la *BC* debe ser cierta en todos los modelos y en este modelo en particular. Por lo tanto, cada sentencia implicada q debe ser inferida por el algoritmo.

DIRIGIDO POR LOS DATOS

El encadenamiento hacia delante es un ejemplo del concepto general de razonamiento **dirigido por los datos**, es decir, un razonamiento en el que el foco de atención parte de los datos conocidos. Este razonamiento se puede utilizar en un agente para derivar conclusiones a partir de percepciones recibidas, a menudo, sin la necesidad de una petición concreta. Por ejemplo, el agente de *wumpus* podría DECIR sus percepciones a la base de conocimiento utilizando un algoritmo de encadenamiento hacia delante de tipo incremental, en el que los hechos se pueden añadir a la agenda para iniciar nuevas inferencias. A las personas, a medida que les llega nueva información, se les activa una gran cantidad de razonamiento dirigido por los datos. Por ejemplo, si estoy en casa y oigo que comienza a llover, podría sucederme que la merienda quedé cancelada. Con todo esto, no será muy probable que el pétalo diecisieteavo de la rosa más alta del jardín de mi vecino se haya mojado. Las personas llevan a cabo un encadenamiento hacia delante con un control cuidadoso, a fin de no hundirse en consecuencias irrelevantes.

El algoritmo de encadenamiento hacia atrás, tal como sugiere su nombre, trabaja hacia atrás a partir de la petición. Si se sabe que la petición q es verdadera, entonces no se requiere realizar ningún trabajo. En el otro caso, el algoritmo encuentra aquellas implicaciones de la base de conocimiento de las que se concluye q . Si se puede probar que todas las premisas de una de esas implicaciones son verdaderas (mediante un encadenamiento hacia atrás), entonces q es verdadera. Cuando se aplica a la petición Q de la Figura 7.15, el algoritmo retrocede hacia abajo por el grafo hasta que encuentra un conjunto de hechos conocidos que forma la base de la demostración. El algoritmo detallado se deja como ejercicio. Al igual que en el encadenamiento hacia delante, una implementación eficiente se ejecuta en tiempo lineal.

RAZONAMIENTO DIRIGIDO POR EL OBJETIVO

El encadenamiento hacia atrás es un tipo de **razonamiento dirigido por el objetivo**. Este tipo de razonamiento es útil para responder a peticiones tales como «¿Qué debo hacer ahora?» y «¿Dónde están mis llaves?». A menudo, el coste del encadenamiento hacia atrás es *mucho menor* que el orden lineal respecto al tamaño de la base de conocimiento, porque el proceso sólo trabaja con los hechos relevantes. Por lo general, un agente debería repartir su trabajo entre el razonamiento hacia delante y el razonamiento hacia atrás, limitando el razonamiento hacia delante a la generación de los hechos que sea probable que sean relevantes para las peticiones, y éstas se resolverán mediante el encadenamiento hacia atrás.

7.6 Inferencia proposicional efectiva

En esta sección vamos a describir dos familias de algoritmos eficientes para la inferencia en lógica proposicional, basadas en la comprobación de modelos: un enfoque basado en la búsqueda con *backtracking*, y el otro en la búsqueda basada en la escalada de colina. Estos algoritmos forman parte de la «tecnología» de la lógica proposicional. Esta sección se puede tan sólo oír en una primera lectura del capítulo.

Los algoritmos que vamos a describir se utilizan para la comprobación de la *satisfacibilidad*. Ya hemos mencionado la conexión entre encontrar un modelo satisfacible para una sentencia lógica y encontrar una solución para un problema de satisfacción de restricciones, entonces, quizás no sorprenda que las dos familias de algoritmos se asemejen bastante a los algoritmos con *backtracking* de la Sección 5.2 y a los algoritmos de búsqueda local de la Sección 5.3. Sin embargo, éstos son extremadamente importantes por su propio derecho, porque muchos problemas combinatorios en las ciencias de la computación se pueden reducir a la comprobación de la *satisfacibilidad* de una sentencia proposicional. Cualquier mejora en los algoritmos de *satisfacibilidad* presenta enormes consecuencias para nuestra habilidad de manejar la complejidad en general.

Un algoritmo completo con *backtracking* («vuelta atrás»)

ALGORITMO DE DAVIS Y PUTNAM

El primer algoritmo que vamos a tratar se le llama a menudo **algoritmo de Davis y Putnam**, después del artículo de gran influencia que escribieron Martin Davis y Hilary Putnam (1960). De hecho, el algoritmo es la versión descrita por Davis, Logemann y Loveland (1962), así que lo llamaremos DPLL, las iniciales de los cuatro autores. DPLL toma como entrada una sentencia en forma normal conjuntiva (un conjunto de cláusulas). Del mismo modo que la BÚSQUEDA-CON-BACKTRACKING e ¿IMPLICACIÓN-TV?, DPLL realiza una enumeración esencialmente recursiva, mediante el primero en profundidad, de los posibles modelos. El algoritmo incorpora tres mejoras al sencillo esquema del ¿IMPLICACIÓN-TV?

- *Terminación anticipada*: el algoritmo detecta si la sentencia debe ser verdadera o falsa, aun con un modelo completado parcialmente. Una cláusula es verdadera si *cualquier* literal es verdadero, aun si los otros literales todavía no tienen valores de verdad; así la sentencia, entendida como un todo, puede evaluarse como verdadera aun antes de que el modelo sea completado. Por ejemplo, la sentencia $(A \vee B) \wedge (A \vee C)$ es verdadera si A lo es, sin tener en cuenta los valores de B y C . De forma similar, una sentencia es falsa si *cualquier* cláusula lo es, caso que ocurre cuando cada uno de sus literales lo son. Del mismo modo, esto puede ocurrir mucho antes de que el modelo sea completado. La terminación anticipada evita la evaluación íntegra de los subárboles en el espacio de búsqueda.
- *Heurística de símbolo puro*: un **símbolo puro** es un símbolo que aparece siempre con el mismo «signo» en todas las cláusulas. Por ejemplo, en las tres cláusulas $(A \vee \neg B)$, $(\neg B \vee \neg C)$, y $(C \vee A)$, el símbolo A es puro, porque sólo aparece el literal positivo, B también es puro porque sólo aparece el literal negativo, y C es un sím-

SÍMBOLO PURO

bolo impuro. Es fácil observar que si una sentencia tiene un modelo, entonces tiene un modelo con símbolos puros asignados para hacer que sus literales sean *verdaderos*, porque al hacerse así una cláusula nunca puede ser falsa. Fíjese en que, al determinar la pureza de un símbolo, el algoritmo puede ignorar las cláusulas que ya se sabe que son verdaderas en el modelo construido hasta ahora. Por ejemplo, si el modelo contiene $B = \text{falso}$, entonces la cláusula $(\neg B \vee \neg C)$ ya es verdadera, y C pasa a ser un símbolo puro, ya que sólo aparecerá en la cláusula $(C \vee A)$.

- **Heurística de cláusula unitaria:** anteriormente había sido definida una **cláusula unitaria** como aquella que tiene sólo un literal. En el contexto del DPLL, este concepto también determina a aquellas cláusulas en las que todos los literales, menos uno, tienen asignado el valor *falso* en el modelo. Por ejemplo, si el modelo contiene $B = \text{falso}$, entonces $(B \vee \neg C)$ pasa a ser una cláusula unitaria, porque es equivalente a $(\text{Falso} \vee \neg C)$, o justamente $\neg C$. Obviamente, para que esta cláusula sea verdadera, a C se le debe asignar *falso*. La heurística de cláusula unitaria asigna dichos símbolos antes de realizar la ramificación restante. Una consecuencia importante de la heurística es que cualquier intento de demostrar (mediante refutación) un literal que ya esté en la base de conocimiento, tendrá éxito inmediatamente (Ejercicio 7.16). Fíjese también en que la asignación a una cláusula unitaria puede crear otra cláusula unitaria (por ejemplo, cuando a C se le asigna *falso*, $(C \vee A)$ pasa a ser una cláusula unitaria, causando que le sea asignado a A el valor verdadero). A esta «cascada» de asignaciones forzadas se la denomina **propagación unitaria**. Este proceso se asemeja al encadenamiento hacia delante con las cláusulas de Horn, y de hecho, si una expresión en FNC sólo contiene cláusulas de Horn, entonces el DPLL esencialmente reproduce el encadenamiento hacia delante. (Véase el Ejercicio 7.17.)

PROPAGACIÓN UNITARIA

En la Figura 7.16 se muestra el algoritmo DPLL. Sólo hemos puesto la estructura básica del algoritmo, que describe, en sí mismo, el proceso de búsqueda. No hemos descrito las estructuras de datos que se deben utilizar para hacer que cada paso de la búsqueda sea eficiente, ni los trucos que se pueden añadir para mejorar su comportamiento: aprendizaje de cláusulas, heurísticas para la selección de variables y reinicialización aleatoria. Cuando estas mejoras se añaden, el DPLL es uno de los algoritmos de *satisfacibilidad* más rápidos, a pesar de su antigüedad. La implementación CHAFF se utiliza para resolver problemas de verificación de *hardware* con un millón de variables.

Algoritmos de búsqueda local

Hasta ahora, en este libro hemos visto varios algoritmos de búsqueda local, incluyendo la ASCENSIÓN-DE-COLINA (página 126) y el TEMPLADO-SIMULADO (página 130). Estos algoritmos se pueden aplicar directamente a los problemas de *satisfacibilidad*, a condición de que elijamos la correcta función de evaluación. Como el objetivo es encontrar una asignación que satisfaga todas las cláusulas, una función de evaluación que cuente el número de cláusulas *insatisfacibles* hará bien el trabajo. De hecho, ésta es exactamente la medida utilizada en el algoritmo MIN-CONFLICTOS para los PSR (página 170). Todos estos algoritmos realizan los pasos en el espacio de asignaciones completas,

función *¿SATISFACIBLE-DPLL?(s)* **devuelve** verdadero o falso
entradas: *s*, una sentencia en lógica proposicional

cláusulas \leftarrow el conjunto de cláusulas de *s* en representación FNC
símbolos \leftarrow una lista de los símbolos proposicionales de *s*
devolver DPLL(*cláusulas*, *símbolos*, [])

función DPLL(*cláusulas*, *símbolos*, *modelo*) **devuelve** verdadero o falso

si cada cláusula en *cláusulas* es verdadera en el *modelo* **entonces devolver** verdadero
 si alguna cláusula en *cláusulas* es falsa en el *modelo* **entonces devolver** falso
 $P, \text{valor} \leftarrow \text{ENCONTRAR-SÍMBOLO-PURO}(\text{símbolos}, \text{cláusulas}, \text{modelo})$
 si P no está vacío **entonces devolver**
 DPLL(*cláusulas*, *símbolos* - P , EXTENDER(P, valor , *modelo*))
 $P, \text{valor} \leftarrow \text{ENCONTRAR-CLÁUSULA-UNITARIA}(\text{cláusulas}, \text{modelo})$
 si P no está vacío **entonces devolver**
 DPLL(*cláusulas*, *símbolos* - P , EXTENDER(P, valor , *modelo*))
 $P \leftarrow \text{PRIMERO}(\text{símbolos}); \text{resto} \leftarrow \text{RESTO}(\text{símbolos})$
devolver DPLL(*cláusulas*, *resto*, EXTENDER($P, \text{verdadero}$, *modelo*)) o
 DPLL(*cláusulas*, *resto*, EXTENDER(P, falso , *modelo*))

Figura 7.16 El algoritmo DPLL para la comprobación de la *satisfacibilidad* de una sentencia en lógica proposicional. En el texto se describen ENCONTRAR-SÍMBOLO-PURO y ENCONTRAR-CLÁUSULA-UNITARIA; cada una devuelve un símbolo (o ninguno) y un valor de verdad para asignar a dicho símbolo. Al igual que ¿IMPLICACIÓN-TV?, este algoritmo trabaja sobre modelos parciales.

intercambiando el valor de verdad de un símbolo a la vez. El espacio generalmente contiene muchos mínimos locales, requiriendo diversos métodos de alcatoriedad para escapar de ellos. En los últimos años se han realizado una gran cantidad de experimentos para encontrar un buen equilibrio entre la voracidad y la aleatoriedad.

Uno de los algoritmos más sencillos y eficientes que han surgido de todo este trabajo es el denominado SAT-CAMINAR (Figura 7.17). En cada iteración, el algoritmo selecciona una cláusula insatisfecha y un símbolo de la cláusula para intercambiarlo. El algoritmo escoge aleatoriamente entre dos métodos para seleccionar el símbolo a intercambiar: (1) un paso de «min-conflictos» que minimiza el número de cláusulas insatisfichas en el nuevo estado, y (2) un paso «pasada-aleatoria» que selecciona de forma aleatoria el símbolo.

¿El SAT-CAMINAR realmente trabaja bien? De forma clara, si el algoritmo devuelve un modelo, entonces la sentencia de entrada de hecho es *satisfacible*. ¿Qué sucede si el algoritmo devuelve *falso*? En ese caso, no podemos decir si la sentencia es *insatisfacible* o si necesitamos darle más tiempo al algoritmo. Podríamos intentar asignarle a *max_intercambios* el valor infinito. En ese caso, es fácil ver que con el tiempo SAT-CAMINAR nos devolverá un modelo (si existe alguno) a condición de que la probabilidad $p > 0$. Esto es porque siempre hay una secuencia de intercambios que nos lleva a una asignación satisfactoria, y al final, los sucesivos pasos de movimientos aleatorios generarán dicha secuencia. Ahora bien, si *max_intercambios* es infinito y la sentencia es *insatisfacible*, entonces la ejecución del algoritmo nunca finalizará.

función SAT-CAMINAR(*cláusulas*, *p*, *max_intercambios*) devuelve un modelo satisfactorio o *fallo*

entradas: *cláusulas*, un conjunto de cláusulas en lógica proposicional
p, la probabilidad de escoger un movimiento «aleatorio», generalmente alrededor de 0,5
max_intercambios el número de intercambios permitidos antes de abandonar

modelo \leftarrow una asignación aleatoria de *verdadero/falso* a los símbolos de las *cláusulas*

para *i* = 1 **hasta** *max_intercambios* **hacer**

- si** *modelo* satisface las *cláusulas* **entonces devolver** *modelo*
- cláusula* \leftarrow una cláusula seleccionada aleatoriamente de *cláusulas* que es falsa en el *modelo*
- con probabilidad** *p* intercambia el valor en el *modelo* de un símbolo seleccionado aleatoriamente de la *cláusula*
- sino** intercambia el valor de cualquier símbolo de la *cláusula* para que maximice el número de cláusulas *satisfacibles*

devolver *fallo*

Figura 7.17 El algoritmo SAT-CAMINAR para la comprobación de la *satisfacibilidad* mediante intercambio aleatorio de los valores de las variables. Existen muchas versiones de este algoritmo.

Lo que sugiere este problema es que los algoritmos de búsqueda local, como el SAT-CAMINAR, son más útiles cuando esperamos que haya una solución (por ejemplo, los problemas de los que hablamos en los Capítulos 3 y 5, por lo general, tienen solución). Por otro lado, los algoritmos de búsqueda local no detectan siempre la *insatisfacibilidad*, algo que se requiere para decidir si hay relación de implicación. Por ejemplo, un agente no puede utilizar la búsqueda local para demostrar, de forma fiable, si una casilla es segura en el mundo de *wumpus*. En lugar de ello, el agente puede decir «he pensado acerca de ello durante una hora y no he podido hallar ningún mundo posible en el que la casilla *no sea segura*». Si el algoritmo de búsqueda local es por lo general realmente más rápido para encontrar un modelo cuando éste existe, el agente se podría justificar asumiendo que el impedimento para encontrar un modelo indica *insatisfacibilidad*. Desde luego que esto no es lo mismo que una demostración, y el agente se lo debería pensar dos veces antes de apostar su vida en ello.

Problemas duros de *satisfacibilidad*

Vamos a ver ahora cómo trabaja en la práctica el DPLL. En concreto, estamos interesados en los problemas *duros* (o *complejos*), porque los problemas *fáciles* se pueden resolver con cualquier algoritmo antiguo. En el Capítulo 5 hicimos algunos descubrimientos sorprendentes acerca de cierto tipo de problemas. Por ejemplo, el problema de las *n-reinas* (pensado como un problema absolutamente difícil para los algoritmos de búsqueda con *backtracking*) resultó ser trivialmente sencillo para los métodos de búsqueda local, como el min-conflictos. Esto es a causa de que las soluciones están distribuidas muy densamente en el espacio de asignaciones, y está garantizado que cualquier asignación inicial tenga cerca una solución. Así, el problema de las *n-reinas* es sencillo porque está **bajo restricciones**.

Cuando observamos los problemas de *satisfacibilidad* en forma normal conjuntiva, un problema *bajo restricciones* es aquel que tiene relativamente *pocas* cláusulas res-

tringiendo las variables. Por ejemplo, aquí tenemos una sentencia en FNC-3 con cinco símbolos, y cinco cláusulas generadas aleatoriamente¹²:

$$(\neg D \vee \neg B \vee C) \wedge (D \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$$

16 de las 32 posibles asignaciones son modelos de esta sentencia, por lo tanto, de media, sólo se requerirían dos pasos para encontrar un modelo.

Entonces, ¿dónde se encuentran los problemas duros? Presumiblemente, si *incrementamos* el número de cláusulas, manteniendo fijo el número de símbolos, hacemos que el problema esté más restringido, y que sea más difícil encontrar las soluciones. Permitamos que m sea el número de cláusulas y n el número de símbolos. La Figura 7.18(a) muestra la probabilidad de que una sentencia en FNC-3 sea *satisfacible*, como una función de la relación cláusula/símbolo, m/n , con n fijado a 50. Tal como esperábamos, para una m/n pequeña, la probabilidad se acerca a 1, y para una m/n grande, la probabilidad se acerca a 0. La probabilidad cae de forma bastante brusca alrededor del valor de $m/n = 4,3$. Las sentencias en FNC que están cerca de este **punto crítico** se podrían definir como «casi satisfacibles» o «casi insatisfacibles». ¿Es en este punto donde encontramos los problemas duros?

La Figura 7.18(b) muestra los tiempos de ejecución de los algoritmos DPLL y SAT-CAMINAR alrededor de este punto crítico, en donde hemos restringido nuestra atención sólo a los problemas *satisfacibles*. Tres cosas están claras: primero, los problemas que están cerca del punto crítico son *mucho* más difíciles que los otros problemas aleatorios. Segundo, aun con los problemas más duros, el DPLL es bastante efectivo (una media de unos pocos miles de pasos comparados con $2^{50} \approx 10^{15}$ en la enumeración de tablas de verdad). Tercero, SAT-CAMINAR es mucho más rápido que el DPLL en todo el rango.

PUNTO CRÍTICO

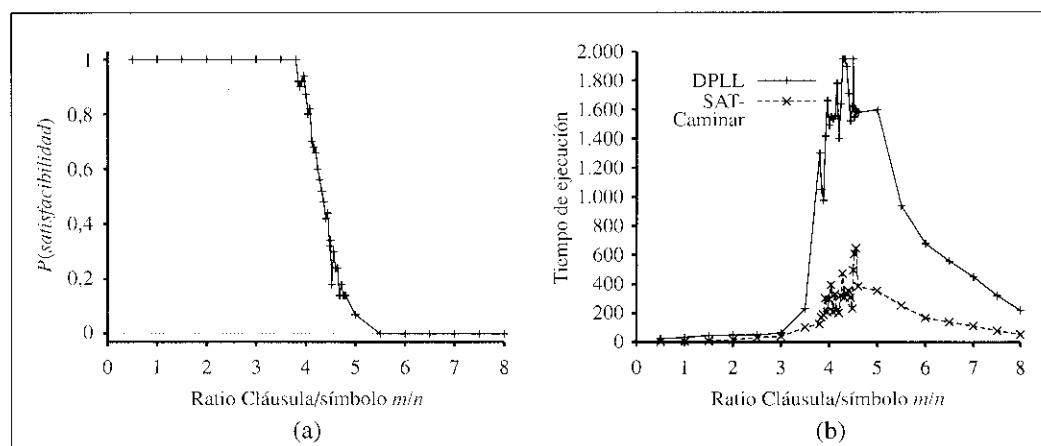


Figura 7.18 (a) Gráfico que muestra la probabilidad de que una sentencia en FNC-3 con $n = 50$ símbolos sea *satisfacible*, en función del ratio cláusula/símbolo m/n . (b) Gráfico del tiempo de ejecución promedio del DPLL y del SAT-CAMINAR sobre 100 sentencias en FNC-3 aleatorias con $n = 50$, para un rango reducido de valores de m/n alrededor del punto crítico.

¹² Cada cláusula contiene tres símbolos *diferentes* seleccionados aleatoriamente, cada uno de ellos negado con una probabilidad del 50%.

Por supuesto, estos resultados sólo son para los problemas generados aleatoriamente. Los problemas reales no tienen necesariamente la misma estructura (en términos de proporciones entre literales positivos y negativos, densidades de conexiones entre cláusulas, etcétera) que los problemas aleatorios. Pero todavía en la práctica, el SAT-CAMINAR y otros algoritmos de la misma familia son muy buenos para resolver problemas reales (a menudo, tan buenos como el mejor algoritmo de propósito específico para esas tareas). Problemas con miles de símbolos y millones de cláusulas se tratan de forma rutinaria con resolutores como el CHAFF. Estas observaciones nos sugieren que alguna combinación de los comportamientos de la heurística de min-conflictos y de pasada-aleatoria nos proporciona una gran capacidad de *propósito-general* para resolver muchas situaciones en las que se requiere el razonamiento combinatorio.

7.7 Agentes basados en lógica proposicional

En esta sección, vamos a juntar lo que hemos aprendido hasta ahora para construir agentes que funcionan utilizando la lógica proposicional. Veremos dos tipos de agentes: aquellos que utilizan algoritmos de inferencia y una base de conocimiento, como el agente basado en conocimiento genérico de la Figura 7.1, y aquellos que evalúan directamente expresiones lógicas en forma de circuitos. Aplicaremos ambos tipos de agentes en el mundo de *wumpus*, y encontraremos que ambos sufren de serias desventajas.

Encontrar hoyos y *wumpus* utilizando la inferencia lógica

Permítanos empezar con un agente que razona mediante la lógica acerca de las localizaciones de los hoyos, de los *wumpus* y de la seguridad de las casillas. El agente comienza con una base de conocimiento que describe la «física» del mundo de *wumpus*. El agente sabe que la casilla [1, 1] no tiene ningún hoyo ni ningún *wumpus*; es decir, $\neg H_{1,1}$ y $\neg W_{1,1}$. El agente también conoce una sentencia que indica cómo se percibe una brisa en una casilla $[x, y]$:

$$B_{x,y} \Leftrightarrow (H_{x,y+1} \vee H_{x,y-1} \vee H_{x+1,y} \vee H_{x-1,y}) \quad (7.1)$$

El agente conoce una sentencia que indica cómo se percibe el hedor en una casilla $[x, y]$:

$$M_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y}) \quad (7.2)$$

Finalmente, el agente sabe que sólo hay un *wumpus*. Esto se expresa de dos maneras. En la primera, debemos definir que hay *por lo menos* un *wumpus*:

$$W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,3} \vee W_{4,4}$$

Entonces, debemos definir que *como mucho* hay un *wumpus*. Una manera de definirlo es diciendo que para dos casillas cualesquiera, una de ellas debe estar libre de un *wumpus*:

wumpus. Con n casillas, tenemos $n(n - 1)/2$ sentencias del tipo $\neg W_{i,1} \vee \neg W_{i,2}$. Entonces, para un mundo de 4×4 , comenzamos con un total de 155 sentencias conteniendo 64 símbolos diferentes.

El programa del agente, que se muestra en la Figura 7.19 DICE a su base de conocimiento cualquier nueva percepción acerca de una brisa o un mal hedor. (También actualiza algunas variables del programa para guardar la pista de dónde se encuentra y qué casillas ha visitado. Estos últimos datos los necesitará más adelante.) Entonces el programa escoge dónde observar antes de entre las casillas que rodean al agente, es decir, las casillas adyacentes a aquellas ya visitadas. Una casilla $[i, j]$ que rodea al agente es *probable que sea segura* si la sentencia $(\neg H_{ij} \wedge \neg W_{ij})$ se deduce de la base de conocimiento. La siguiente alternativa mejor es una casilla *posiblemente segura*, o sea, aquella casilla para la cual el agente no puede demostrar si hay un hoyo o un *wumpus*, es decir, aquella para la que *no* se deduce $(H_{ij} \vee W_{ij})$.

El cálculo para averiguar la implicación mediante PREGUNTAR se puede implementar utilizando cualquiera de los métodos descritos anteriormente en este capítulo. ¿IMPLICACIÓN-TV? (Figura 7.10) es obviamente impracticable, ya que debería enumerar 2^{64} filas. El DPLL (Figura 7.16) realiza las inferencias necesarias en pocos milisegundos, principalmente gracias a la heurística de propagación unitaria. El SAT-CAMINAR también se puede utilizar, teniendo en cuenta la advertencia acerca de la incompletitud. En el mundo de *wumpus*, los obstáculos para encontrar un modelo, realizados 10.000 intercambios,

función AGENTE-WUMPUS-LP(percepción) devuelve una acción

entradas: *percepción*, una lista, [*hedor*, *brisa*, *resplandor*]

variables estáticas: *BC*, una base de conocimiento, inicialmente conteniendo la «física» del mundo de *wumpus*

x, *y*, *orientación*, la posición del agente (inicialmente en 1, 1) y su orientación (inicialmente *derecha*)

visitada, una matriz indicando qué casillas han sido visitadas, inicialmente *falso*

acción, la acción más reciente del agente, inicialmente *null*

plan, una secuencia de acciones, inicialmente vacía

actualiza *x*, *y*, *orientación*, *visitada* basada en *acción*

si *hedor* **entonces** DECIR(*BC*, $M_{x,y}$) **sino** DECIR(*BC*, $\neg M_{x,y}$)

si *brisa* **entonces** DECIR(*BC*, $B_{x,y}$) **sino** DECIR(*BC*, $\neg B_{x,y}$)

si *resplandor* **entonces** *acción* \leftarrow *agarrar*

sino **si** *plan* no está vacío **entonces** *acción* \leftarrow POP(*plan*)

sino **si** para alguna casilla $[i, j]$ que nos rodea es *verdadero* PREGUNTAR(*BC*, $(\neg H_{ij} \wedge \neg W_{ij})$)

o es *falso* PREGUNTAR(*BC*, $(H_{ij} \vee W_{ij})$) **entonces** *hacer*

plan \leftarrow BUSQUEDA-GRAFO-A*(PROBLEMA-RUTA($[x, y]$, *orientación*, $[i, j]$, *visitada*))

acción \leftarrow POP(*plan*)

sino *acción* \leftarrow un movimiento escogido al azar

devolver *acción*

Figura 7.19 Un agente en el mundo de *wumpus* que utiliza la lógica proposicional para identificar hoyos, *wumpus* y casillas seguras. La subrutina PROBLEMA-RUTA construye un problema de búsqueda cuya solución es una secuencia de acciones que nos llevan de $[x, y]$ a $[i, j]$ pasando sólo a través de las casillas previamente visitadas.

se corresponden invariablemente con la *insatisfacibilidad*, así que los errores no se deben probablemente a la incompletitud.

El programa AGENTE-WUMPUS-LP se comporta bastante bien en un mundo de *wumpus* pequeño. Sin embargo, sucede algo profundamente insatisfactorio respecto a la base de conocimiento del agente. La BC contiene las sentencias que definen la «física» en la forma dada en las Ecuaciones (7.1) y (7.2) para cada casilla individual. Cuanto más grande sea el entorno, más grande necesita ser la base de conocimiento inicial. Preferiríamos en mayor medida disponer sólo de las dos sentencias para definir cómo se presenta una brisa o un hedor en *cualquier* casilla. Esto está más allá del poder de expresión de la lógica proposicional. En el próximo capítulo veremos un lenguaje lógico más expresivo mediante el cual es más fácil expresar este tipo de sentencias.

Guardar la pista acerca de la localización y la orientación del agente

El programa del agente de la Figura 7.19 «hace trampa» porque guarda la pista de su localización *fuerza* de la base de conocimiento, en vez de utilizar el razonamiento lógico¹³. Para hacerlo «correctamente» necesitaremos proposiciones acerca de la localización. Una primera aproximación podría consistir en utilizar un símbolo como $L_{1,1}$ para indicar que el agente se encuentra en la casilla [1, 1]. Entonces la base de conocimiento inicial podría incluir sentencias como

$$L_{1,1} \wedge \text{OrientadoDerecha} \wedge \text{Avanzar} \Rightarrow L_{2,1}$$

Vemos inmediatamente que esto no funciona correctamente. Si el agente comienza en la casilla [1, 1] orientado a la derecha y avanza, de la base de conocimiento se deduciría $L_{1,1}$ (la localización original del agente) y $L_{2,1}$ (su nueva localización). ¡Aunque estas dos proposiciones no pueden ser verdaderas a la vez! El problema es que las proposiciones de localización deberían referirse a dos instantes de tiempo diferentes. Necesitamos $L_{1,1}^1$ para indicar que el agente se encuentra en la casilla [1, 1] en el instante de tiempo 1, $L_{2,1}^2$ para indicar que el agente se encuentra en la casilla [2, 1] en el instante de tiempo 2, etcétera. Las proposiciones acerca de la orientación y la acción también necesitan depender del tiempo. Por lo tanto, la sentencia correcta es

$$\begin{aligned} L_{1,1}^1 \wedge \text{OrientadoDerecha}^1 \wedge \text{Avanzar}^1 &\Rightarrow L_{2,1}^2 \\ \text{OrientadoDerecha} \wedge \text{GirarIzquierda}^1 &\Rightarrow \text{OrientadoArriba}^2 \end{aligned}$$

De esta manera resulta bastante difícil construir una base de conocimiento completa y correcta para guardar la pista de cada cosa que sucede en el mundo de *wumpus*; pospondremos la discusión en su detalle para el Capítulo 10. Lo que nos proponemos hacer aquí es que la base de conocimiento inicial contenga sentencias como los dos ejemplos anteriores para cada instante de tiempo t , así como para cada localización. Es

¹³ El lector que sea observador se habrá dado cuenta de que esto nos posibilitó afinar la conexión que hay entre las percepciones, como *Brisa* y la proposiciones acerca de la localización específica, como $B_{1,1}$.

decir, que para cada instante de tiempo t y localización $[x, y]$, la base de conocimiento contenga una sentencia del tipo

$$L'_{x,y} \wedge OrientadoDerecha' \wedge Avanzar' \Rightarrow L'^{+1}_{x+1,y} \quad (7.3)$$

Aunque pongamos un límite superior de pasos permitidos en el tiempo (por ejemplo 100) acabamos con decenas de miles de sentencias. Se presenta el mismo problema si añadimos las sentencias «a medida que las necesitemos» en cada paso en el tiempo. Esta proliferación de cláusulas hace que la base de conocimiento sea ilegible para las personas, sin embargo, los resolutores rápidos en lógica proposicional aún pueden manejar un mundo de *wumpus* de 4×4 con cierta facilidad (éstos encuentran su límite en los mundos de 100×100 casillas). Los agentes basados en circuitos de la siguiente sección ofrecen una solución parcial a este problema de proliferación de las cláusulas, pero la solución íntegra deberá esperar hasta que hayamos desarrollado la lógica de primer orden en el Capítulo 8.

Agentes basados en circuitos

AGENTE BASADO EN CIRCUITOS

CIRCUITO SECUENCIAL

PUERTAS

REGISTROS

Un **agente basado en circuitos** es un tipo particular de agente reflexivo con estado interno, tal como se definió en el Capítulo 2. Las percepciones son las entradas de un **círculo secuencial** (una red de **puertas**, cada una de ellas implementa una conectiva lógica, y de **registros**, cada uno de ellos almacena el valor lógico de una proposición atómica). Las salidas del circuito son los registros que se corresponden con las acciones, por ejemplo, la salida *Agarrar* se asigna a *verdadero* si el agente quiere coger algo. Si la entrada *Resplandor* se conecta directamente a la salida *Agarrar*, el agente cogerá el objetivo (el objeto *oro*) siempre que vea el resplandor. (Véase Figura 7.20.)

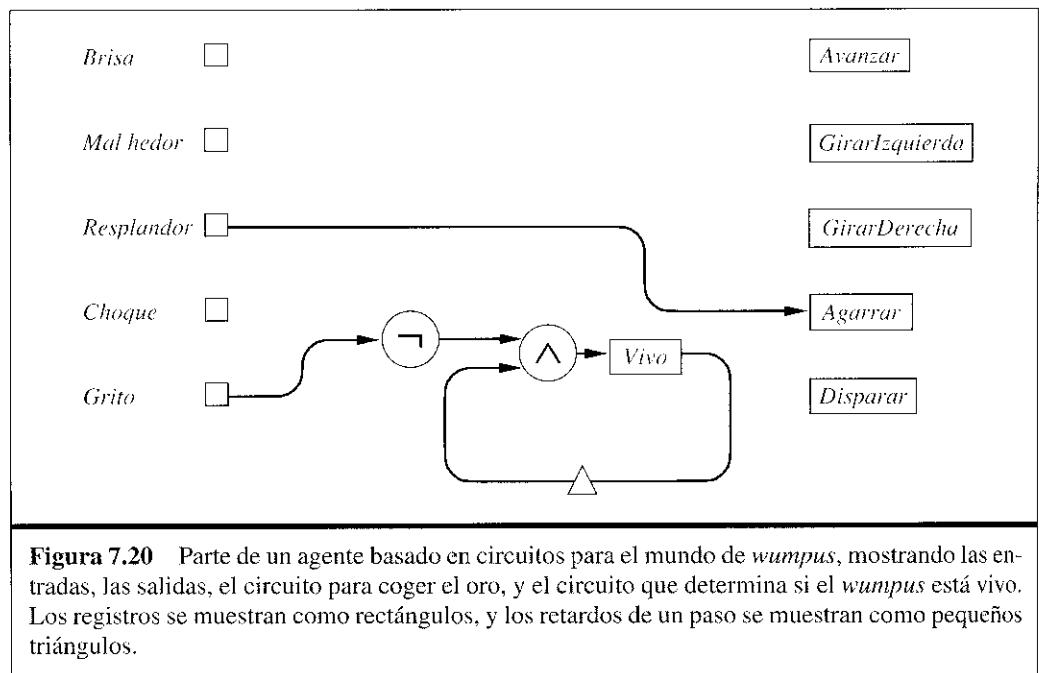


Figura 7.20 Parte de un agente basado en circuitos para el mundo de *wumpus*, mostrando las entradas, las salidas, el circuito para coger el oro, y el circuito que determina si el *wumpus* está vivo. Los registros se muestran como rectángulos, y los retardos de un paso se muestran como pequeños triángulos.

FLUJO DE DATOS

Los circuitos se evalúan de igual modo que los **flujos de datos**: en cada instante de tiempo, se asignan las entradas y se propagan las señales a través del circuito. Siempre que una puerta disponga de todas sus entradas, ésta produce una salida. Este proceso está íntimamente relacionado con el de encadenamiento hacia delante en un grafo Y-O, como el de la Figura 7.15(b).

En la sección precedente dijimos que los agentes basados en circuitos manejan el tiempo de forma más satisfactoria que los agentes basados en la inferencia proposicional. Esto se debe a que cada registro nos da el valor de verdad de su correspondiente símbolo proposicional *en el instante de tiempo actual t*, en vez de disponer de una copia diferente para cada instante de tiempo. Por ejemplo, podríamos tener un registro para *Vivo* que debería contener el valor *verdadero* cuando el *wumpus* esté vivo, y *falso* cuando esté muerto. Este registro se corresponde con el símbolo proposicional *Vivo'*, de esta manera, en cada instante de tiempo el registro se refiere a una proposición diferente. El estado interno del agente, es decir, su memoria, mantiene conectando hacia atrás la salida de un registro con el circuito mediante una **línea de retardo**. Este mecanismo nos da el valor del registro en el instante de tiempo previo. En la Figura 7.20 se muestra un ejemplo. El valor del registro *Vivo* se obtiene de la conjunción de la negación del registro *Grito* y de su propio valor anterior. En términos de proposiciones, el circuito del registro *Vivo* implementa la siguiente conectiva bicondicional

$$\text{Vivo}' \Leftrightarrow \neg\text{Grito}' \wedge \text{Vivo}^{t-1} \quad (7.4)$$

que nos dice que el *wumpus* está vivo en el instante *t* si y sólo si no se percibe ningún grito en el instante *t* y estaba vivo en el instante *t - 1*. Asumimos que el circuito se inicializa con el registro *Vivo* asignado a *verdadero*. Por lo tanto, *Vivo* permanecerá siendo verdadero hasta que haya un grito, con lo que se convertirá y se mantendrá en el valor falso. Esto es exactamente lo que deseamos.

La localización del agente se puede tratar, en mucho, de la misma forma que la salud del *wumpus*. Necesitamos un registro $L_{x,y}$ para cada *x* e *y*; su valor debería ser *verdadero* si el agente se encuentra en la casilla $[x, y]$. Sin embargo, el circuito que asigna el valor de $L_{x,y}$ es mucho más complicado que el circuito para el registro *Vivo*. Por ejemplo, el agente está en la casilla $[1, 1]$ en el instante *t* si: (a) estaba allí en el instante *t - 1* y no se movió hacia delante o lo intentó pero tropezó con un muro; o (b) estaba en la casilla $[1, 2]$ orientado hacia abajo y avanzó; o (c) estaba en la casilla $[2, 1]$ orientado a la izquierda y avanzó:

$$\begin{aligned} L'_{1,1} \Leftrightarrow & (L'^{-1}_{1,1} \wedge (\neg\text{Avanzar}^{t-1} \vee \text{Tropezar}')) \\ & \vee (L'^{-1}_{1,2} \wedge (\text{OrientadoAbajo}^{t-1} \wedge \text{Avanzar}^{t-1})) \\ & \vee (L'^{-1}_{2,1} \wedge (\text{OrientadoIzquierda}^{t-1} \wedge \text{Avanzar}^{t-1})) \end{aligned} \quad (7.5)$$

En la Figura 7.21 se muestra el circuito para $L_{1,1}$. Cada registro de localización tiene enlazado un circuito similar a éste. En el Ejercicio 7.13(b) se pide diseñar un circuito para las proposiciones de orientación.

Los circuitos de las Figuras 7.20 y 7.21 mantienen los valores correctos de los registros *Vivo* y $L_{x,y}$ en todo momento. Sin embargo, estas proposiciones son extrañas, en el sentido de que *sus valores de verdad correctos siempre se pueden averiguar*. En lu-

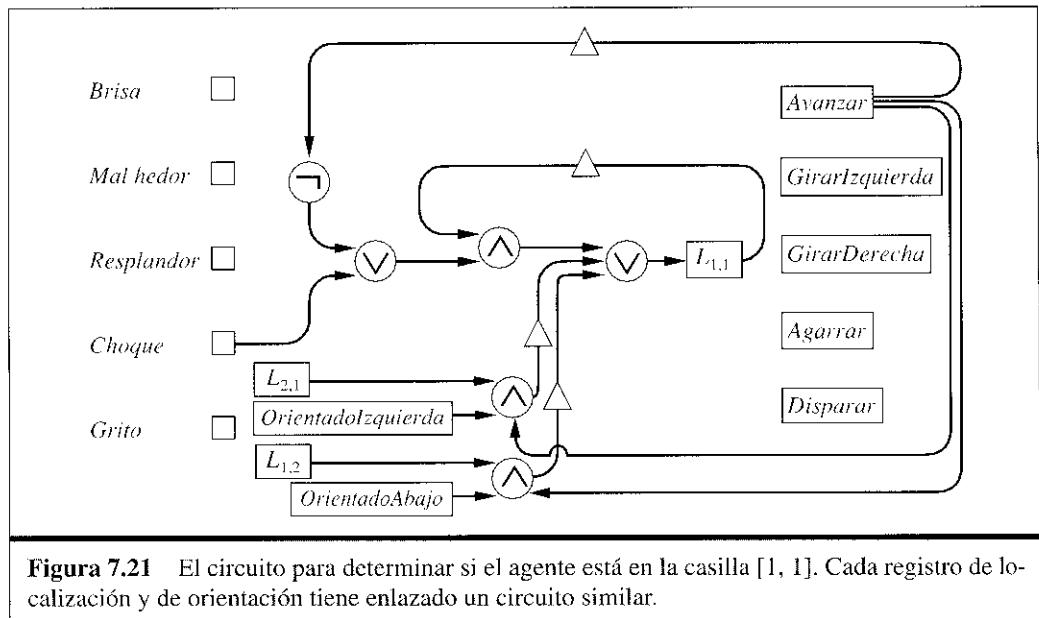


Figura 7.21 El circuito para determinar si el agente está en la casilla [1, 1]. Cada registro de localización y de orientación tiene enlazado un circuito similar.

gar de ello, consideremos la proposición $B_{4,4}$: en la casilla [4, 4] se percibe una brisa. Aunque el valor de verdad de esta proposición se mantiene fijo, el agente no puede aprender su valor hasta que haya visitado la casilla [4, 4] (o haya deducido que hay un hoyo cercano). Las lógicas proposicional y de primer orden están diseñadas para representar proposiciones verdaderas, falsas, o inciertas, de forma automática. Los circuitos no pueden hacerlo: el registro $B_{4,4}$ debe contener *algún* valor, bien *verdadero* o *falso*, aun antes de que se descubra su valor de verdad. El valor del registro bien podría ser el erróneo, y esto nos llevaría a que el agente se extraviara. En otras palabras, necesitamos representar tres posibles estados (se sabe que $B_{4,4}$ es verdadero, falso, o desconocido) y sólo tenemos un *bit* para representar estos estados.

La solución a este problema es utilizar dos bits en vez de uno. $B_{4,4}$ se puede representar mediante dos registros, a los que llamaremos $K(B_{4,4})$ y $K(\neg B_{4,4})$, en donde K significa «conocido». (¡Tenga en cuenta que esta representación consiste tan sólo en unos símbolos con nombres complicados, aunque parezcan expresiones estructuradas!) Cuando ambas, $K(B_{4,4})$ y $K(\neg B_{4,4})$, son falsas, significa que el valor de verdad de $B_{4,4}$ es desconocido. (¡Si ambas son ciertas, entonces la base de conocimiento tiene un fallo!) A partir de ahora, utilizaríamos $K(B_{4,4})$ en vez de $B_{4,4}$, siempre que ésta aparezca en alguna parte del circuito. Por lo general, representamos cada proposición que es potencialmente indeterminada mediante dos **proposiciones acerca del conocimiento** para especificar si la proposición subyacente se sabe que es verdadera y se sabe que es falsa.

En breve veremos un ejemplo de cómo utilizar las proposiciones acerca del conocimiento. Primero necesitamos resolver cómo determinar los valores de verdad de las propias proposiciones acerca del conocimiento. Fíjese en que, mientras $B_{4,4}$ tiene un valor de verdad fijo, $K(B_{4,4})$ y $K(\neg B_{4,4})$ cambian a medida que el agente descubre más cosas acerca del mundo. Por ejemplo, $K(B_{4,4})$ comienza siendo falsa y entonces se transforma en verdadera tan pronto como se puede determinar $B_{4,4}$ que es verdadera (es decir, cuan-

do el agente está en la casilla [4, 4] y detecta una brisa). A partir de entonces permanece siendo verdadera. Así tenemos

$$K(B_{4,4})^t \Leftrightarrow K(B_{4,4})^{t-1} \vee (L_{4,4}^t \wedge Brisa^t) \quad (7.6)$$

Se puede escribir una ecuación similar para $K(\neg B_{4,4})^t$.

Ahora que el agente sabe acerca de las casillas con brisa, puede ocuparse de los hoyos. La ausencia de un hoyo en una casilla se puede averiguar si y sólo si se sabe que en una de sus casillas vecinas no hay ninguna brisa. Por ejemplo, tenemos

$$K(\neg H_{4,4})^t \Leftrightarrow K(\neg B_{3,4})^t \vee K(\neg B_{4,3})^t \quad (7.7)$$

Determinar si *hay* un hoyo en una casilla es más difícil, debe haber una brisa en una casilla adyacente que no sea causada por otro hoyo:

$$\begin{aligned} K(H_{4,4})^t &\Leftrightarrow (K(B_{3,4})^t \wedge K(\neg H_{2,4})^t \wedge K(\neg H_{3,3})^t) \\ &\vee (K(B_{4,3})^t \wedge K(\neg H_{4,2})^t \wedge K(\neg H_{3,3})^t) \end{aligned} \quad (7.8)$$

Mientras que utilizar los circuitos para determinar la presencia o ausencia de hoyos puede resultar algo peliagudo, *sólo se necesitan un número constante de puertas para cada casilla*. Esta propiedad es esencial si debemos construir agentes basados en circuitos que se pueden ampliar de forma razonable. En realidad ésta es una propiedad del propio mundo de *wumpus*; decimos que un entorno manifiesta **localidad** si el valor de verdad de una proposición relevante se puede obtener observando sólo un número constante de otras proposiciones. La localidad es muy sensible a la «física» precisa del entorno. Por ejemplo, el dominio de los dragaminas (Ejercicio 7.11) no es un dominio localista, porque determinar si una mina se encuentra en una casilla dada puede requerir observar las casillas que están arbitrariamente lejos. Los agentes basados en circuitos no son siempre practicables para los dominios no localistas.



ACÍCLICO

Hay un asunto por el que hemos caminado de puntillas y con mucho cuidado: el tema es la propiedad de ser **acíclico**. Un circuito es acíclico si cada uno de sus caminos en el que se conecta hacia atrás la salida de un registro con su entrada se realiza mediante un elemento de retardo. ¡Necesitamos que todos los circuitos sean acíclicos porque los que no lo son, al igual que los artefactos físicos, no funcionan! Éstos pueden entrar en oscilaciones inestables produciendo valores indefinidos. Como ejemplo de un circuito cíclico, tenga en cuenta la siguiente ampliación de la Ecuación (7.6):

$$K(B_{4,4})^t \Leftrightarrow K(B_{4,4})^{t-1} \vee (L_{4,4}^t \wedge Brisa^t) \vee K(H_{3,4})^t \vee K(H_{4,3})^t \quad (7.9)$$

Los disyuntores extras, $K(H_{3,4})^t$ y $K(H_{4,3})^t$, permiten al agente determinar si hay brisa a partir del conocimiento de la presencia de hoyos en las casillas adyacentes, algo que parece totalmente razonable. Pero ahora, desafortunadamente, tenemos que la detección de la brisa depende de los hoyos adyacentes, y la detección de los hoyos depende de las brisas adyacentes, si tenemos en cuenta la Ecuación (7.8). Por lo tanto el circuito, en su conjunto, tendría ciclos.

La dificultad no es que la Ecuación (7.9) ampliada sea *incorrecta*. Más bien, el problema consiste en que las dependencias entrelazadas que se presentan en estas ecuaciones no se pueden resolver por el simple mecanismo de la propagación de los valores de

verdad en el correspondiente circuito Booleano. La versión acíclica utilizando la Ecuación (7.6), que determina si hay brisa sólo a partir de las observaciones directas en la casilla es *incompleta*, en el sentido de que en algunos puntos, el agente basado en circuitos podría saber menos que un agente basado en inferencia utilizando un procedimiento de inferencia completo. Por ejemplo, si hay una brisa en la casilla [1, 1], el agente basado en inferencia puede concluir que también hay una brisa en la casilla [2, 2], mientras que el agente basado en circuitos no puede hacerlo, utilizando la Ecuación (7.6). Se *puede* construir un circuito completo (después de todo, los circuitos secuenciales pueden emular cualquier computador digital) pero sería algo significativamente más complejo.

Una comparación

Los agentes basados en inferencia y los basados en circuitos representan los extremos declarativo y procesal en el diseño de agentes. Se pueden comparar según diversas dimensiones:

- *Precisión*: el agente basado en circuitos, a diferencia del agente basado en inferencia, no necesita disponer de copias diferentes de su «conocimiento» para cada instante de tiempo. En vez de ello, éste sólo se refiere al instante actual y al previo. Ambos agentes necesitan copias de la «física» de cada casilla (expresada mediante sentencias o circuitos), y por lo tanto, no se amplían adecuadamente a entornos mayores. En entornos con muchos objetos relacionados de forma compleja el número de proposiciones abrumará a cualquier agente proposicional. Este tipo de entornos requiere del poder expresivo de la lógica de primer orden. (Véase Capítulo 8.) Además, ambos tipos de agente proposicional están poco preparados para expresar o resolver el problema de encontrar un camino a una casilla segura que esté cercana. (Por esta razón, el AGENTE-WUMPUS-LP recurre a un algoritmo de búsqueda.)
- *Eficiencia computacional*: en el *peor de los casos*, la inferencia puede tomar un tiempo exponencial respecto al número de símbolos, mientras que evaluar un circuito toma un tiempo lineal respecto al tamaño del circuito (o lineal respecto a la *intensidad de integración* del circuito, si éste se construye como un dispositivo físico). Sin embargo, vemos que en la *práctica*, el DPLL realiza las inferencias requeridas bastante rápidamente¹⁴.
- *Compleitud*: anteriormente insinuamos que el agente basado en circuitos podría ser incompleto, debido a la restricción de que sea acíclico. Las causas de la incompletitud son en realidad más básicas. Primero, recordemos que un circuito se ejecuta en tiempo lineal respecto a su tamaño. Esto significa que, para algunos entornos, un circuito que sea completo (a saber, uno que calcula el valor de verdad de cada proposición determinable) debe ser exponencialmente más grande que la *BC* de un agente basado en inferencia. Dicho de otro modo, deberíamos poder resolver el problema de la implicación proposicional en menor tiempo que el tiempo exponencial, lo que parece improbable. Una segunda causa es la naturaleza del

¹⁴ De hecho, ¡todas las inferencias hechas por un circuito se pueden hacer por el DPLL en tiempo lineal! Esto es debido a que la evaluación de un circuito, al igual que el encadenamiento hacia delante, se puede emular mediante el DPLL, utilizando la regla de propagación unitaria.

estado interno del agente. El agente basado en inferencia recuerda cada percepción que ha tenido, y conoce, bien implícita o explícitamente, cada sentencia que se sigue de las percepciones y la BC inicial. Por ejemplo, dado $B_{1,1}$, el agente conoce la disyunción $H_{1,2} \vee H_{2,1}$, de lo cual se sigue $B_{2,2}$. Por el otro lado, el agente basado en circuitos olvida todas sus percepciones anteriores y recuerda tan sólo las proposiciones individuales almacenadas en los registros. De este modo, $H_{1,2}$ y $H_{2,1}$ permanecen desconocidas *cada una de ellas* aún después de la primera percepción, así que no se sacará ninguna conclusión acerca de $B_{2,2}$.

- *Facilidad de construcción:* este es un aspecto muy importante acerca del cual es difícil ser precisos. Desde luego, los autores encontramos mucho más fácil especificar la «física» de forma declarativa, mientras que idear pequeños circuitos acíclicos, y no demasiado incompletos, para la detección directa de hoyos, nos ha parecido bastante dificultoso.

En suma, parece que hay *compensaciones* entre la eficiencia computacional, la concisión, la completitud y la facilidad de construcción. Cuando la conexión entre las percepciones y las acciones es simple (como en la conexión entre *Resplandor* y *Agarrar*) un circuito parece que es óptimo. En un dominio como el ajedrez, por ejemplo, las reglas declarativas son concisas y fácilmente codificables (como mínimo en la lógica de primer orden), y en cambio, utilizar un circuito para calcular los movimientos directamente a partir de los estados del tablero, sería un esfuerzo inimaginablemente enorme.

Podemos observar estos diferentes tipos de compensaciones en el reino animal. Los animales inferiores, con sus sistemas nerviosos muy sencillos, quizás estén basados en circuitos, mientras que los animales superiores, incluyendo a los humanos, parecen realizar inferencia con base en representaciones explícitas. Esta característica les permite ejecutar funciones mucho más complejas de un agente. Los humanos también disponen de circuitos para implementar sus reflejos, y quizás, también para **compilar** sus representaciones declarativas en circuitos, cuando ciertas inferencias pasan a ser una rutina. En este sentido, el diseño de un **agente híbrido** (véase Capítulo 2) podría poseer lo mejor de ambos mundos.

COMPILACIÓN

7.8 Resumen

Hemos introducido los agentes basados en conocimiento y hemos mostrado cómo definir una lógica con la que los agentes pueden razonar acerca del mundo. Los principales puntos son los siguientes:

- Los agentes inteligentes necesitan el conocimiento acerca del mundo para tomar las decisiones acertadas.
- Los agentes contienen el conocimiento en forma de **sentencias** mediante un **lenguaje de representación del conocimiento**, las cuales quedan almacenadas en una **base de conocimiento**.
- Un agente basado en conocimiento se compone de una base de conocimiento y un mecanismo de inferencia. El agente opera almacenando las sentencias acerca del mundo en su base de conocimiento, utilizando el mecanismo de inferencia para

inferir sentencias nuevas, y utilizando estas sentencias nuevas para decidir qué acción debe tomar.

- Un lenguaje de representación del conocimiento se define por su **sintaxis**, que especifica la estructura de las sentencias, y su **semántica**, que define el **valor de verdad** de cada sentencia en cada **mundo posible**, o **modelo**.
- La relación de **implicación** entre las sentencias es crucial para nuestro entendimiento acerca del razonamiento. Una sentencia α implica otra sentencia β si β es verdadera en todos los mundos donde α lo es. Las definiciones familiares a este concepto son: la **valididad** de la sentencia $\alpha \Rightarrow \beta$, y la **insatisfacibilidad** de la sentencia $\alpha \wedge \neg\beta$.
- La inferencia es el proceso que consiste en derivar nuevas sentencias a partir de las ya existentes. Los algoritmos de inferencia **sólidos** sólo derivan aquellas sentencias que son implicadas; los algoritmos **completos** derivan todas las sentencias implicadas.
- La **lógica proposicional** es un lenguaje muy sencillo compuesto por los **símbolos proposicionales** y las **conectivas lógicas**. De esta manera se pueden manejar proposiciones que se sabe son ciertas, falsas, o completamente desconocidas.
- El conjunto de modelos posibles, dado un vocabulario proposicional fijado, es finito, y así se puede comprobar la implicación tan sólo enumerando los modelos. Los algoritmos de inferencia basados en la **comprobación de modelos** más eficientes para la lógica proposicional, entre los que se encuentran los métodos de búsqueda local y *backtracking*, a menudo pueden resolver problemas complejos muy rápidamente.
- Las **reglas de inferencia** son patrones de inferencia sólidos que se pueden utilizar para encontrar demostraciones. De la regla de **resolución** obtenemos un algoritmo de inferencia completo para bases de conocimiento que están expresadas en **forma normal conjuntiva**. El **encadenamiento hacia delante** y el **encadenamiento hacia atrás** son algoritmos de razonamiento muy adecuados para bases de conocimiento expresadas en **cláusulas de Horn**.
- Se pueden diseñar dos tipos de agentes que utilizan la lógica proposicional: los **agentes basados en inferencia** utilizan algoritmos de inferencia para guardar la pista del mundo y deducir propiedades ocultas, mientras que los **agentes basados en círculos** representan proposiciones mediante bits en registros, y los actualizan utilizando la propagación de señal de los circuitos lógicos.
- La lógica proposicional es razonablemente efectiva para ciertas tareas de un agente, pero no se puede escalar para entornos de tamaño ilimitado, a causa de su falta de poder expresivo para manejar el tiempo de forma precisa, el espacio, o patrones genéricos de relaciones entre objetos.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

La ponencia «Programas con Sentido Común» de John McCarthy (McCarthy, 1958, 1968) promulgaba la noción de agentes que utilizan el razonamiento lógico para mediar entre sus percepciones y sus acciones. En él también erigió la bandera del declarativis-

mo, señalando que decirle a un agente lo que necesita saber es una forma muy elegante de construir *software*. El artículo «El Nivel de Conocimiento» de Allen Newell (1982) propone que los agentes racionales se pueden describir y analizar a un nivel abstracto definido a partir del conocimiento que el agente posee más que a partir de los programas que ejecuta. En Boden (1977) se comparan los enfoques declarativo y procedural en la IA. Este debate fue reanimado, entre otros, por Brooks (1991) y Nilsson (1991).

La Lógica tiene sus orígenes en la Filosofía y las Matemáticas de los Griegos antiguos. En los trabajos de Platón se encuentran esparcidos diversos principios lógicos (principios que conectan la estructura sintáctica de las sentencias con su verdad o falsedad, con su significado, o con la validez de los argumentos en los que aparecen). El primer estudio sistemático acerca de la lógica que se conoce lo llevó a cabo Aristóteles, cuyo trabajo fue recopilado por sus estudiantes después de su muerte, en el 322 a.C., en un tratado denominado *Organon*. Los **silogismos** de Aristóteles fueron lo que ahora podríamos llamar reglas de inferencia. Aunque los silogismos tenían elementos, tanto de la lógica proposicional como de la de primer orden, el sistema como un todo era algo endeble según los estándares actuales. No permitió aplicar los patrones de inferencia a sentencias de complejidad diversa, cosa que sí lo permite la lógica proposicional moderna.

Las escuelas íntimamente ligadas de los estoicos y los megarianos (que se originaron en el siglo V d.C. y continuaron durante varios siglos después) introdujeron el estudio sistemático de la implicación y otras estructuras básicas que todavía se utilizan en la lógica proposicional moderna. La utilización de las tablas de verdad para definir las conectivas lógicas se debe a Filo de Megara. Los estoicos tomaron como válidas cinco reglas de inferencia básicas sin demostrarlas, incluyendo la regla que ahora denominamos Modus Ponens. Derivaron un buen número de reglas a partir de estas cinco, utilizando entre otros principios, el teorema de la deducción (página 236) y fueron mucho más claros que Aristóteles acerca del concepto de demostración. Los estoicos afirmaban que su lógica era completa, en el sentido de ser capaz de reproducir todas las inferencias válidas; sin embargo, lo que de ellos queda es un conjunto de explicaciones demasiado fragmentadas. Que se sepa, un buen relato acerca de las lógicas Megariana y Estoica es el texto de Benson Mates (1953).

La idea de reducir la lógica a un proceso puramente mecánico, aplicado a un lenguaje formal es de Leibniz (1646-1716). Sin embargo, su propia lógica matemática era gravemente deficiente, y él es mucho más recordado simplemente por introducir estas ideas como objetivos a alcanzar que por sus intentos de lograrlo.

George Boole (1847) introdujo el primer sistema detallado y factible sobre lógica formal en su libro *El análisis Matemático de la Lógica*. La lógica de Boole estaba totalmente modelada a partir del álgebra de los números reales y utilizó la sustitución de expresiones lógicamente equivalentes como su principal método de inferencia. Aunque el sistema de Boole no abarcaba toda la lógica proposicional, estaba lo bastante cerca como para que otros matemáticos completaran las lagunas. Schröder (1877) describió la forma normal conjuntiva, mientras que las cláusulas de Horn fueron introducidas mucho más tarde por Alfred Horn (1951). La primera explicación completa acerca de la lógica proposicional moderna (y de la lógica de primer orden) se encuentra en el *Begriffschrift* («Escritura de Conceptos» o «Notación conceptual») de Gottlob Frege (1879).

El primer artefacto mecánico para llevar a cabo inferencias lógicas fue construido por el tercer Conde de Stanhope (1753-1816). El demostrador de Stanhope podía manejar silogismos y ciertas inferencias con la teoría de las probabilidades. William Stanley Jevons, uno de esos que hizo mejoras y amplió el trabajo de Boole, construyó su «piano lógico» en 1869, artefacto para realizar inferencias en lógica Booleana. En el texto de Martín Gardner (1968) se encuentra una historia entretenida e instructiva acerca de estos y otros artefactos mecánicos modernos utilizados para el razonamiento. El primer programa de computador para la inferencia lógica publicado fue el Teórico Lógico de Newell, Shaw y Simon (1957). Este programa tenía la intención de modelar los procesos del pensamiento humano. De hecho, Martin Davis (1957) había diseñado un programa similar que había presentado en una demostración en 1954, pero los resultados del Teórico Lógico fueron publicados un poco antes. Tanto el programa de 1954 de Davis como el Teórico Lógico estaban basados en algunos métodos *ad hoc* que no influyeron fuertemente más tarde en la deducción automática.

Las tablas de verdad, como un método para probar la validez o *insatisfacibilidad* de las sentencias en el lenguaje de la lógica proposicional, fueron introducidas por separado, por Ludwig Wittgenstein (1922) y Emil Post (1921). En los años 30 se realizaron una gran cantidad de avances en los métodos de inferencia para la lógica de primer orden. En concreto, Gödel (1930) demostró que se podía obtener un procedimiento completo para la inferencia en lógica de primer orden, mediante su reducción a la lógica proposicional utilizando el teorema de Herbrand (Herbrand, 1930). Retomaremos otra vez esta historia en el Capítulo 9; aquí el punto importante es que el desarrollo de los algoritmos proposicionales eficientes de los años 60 fue causado en gran parte, por el interés de los matemáticos en un demostrador de teoremas efectivo para la lógica de primer orden. El algoritmo de Davis y Putnam (Davis y Putnam, 1960) fue el primer algoritmo efectivo para la resolución proposicional, pero en muchos casos era menos eficiente que el algoritmo DPLL con *backtracking* introducido dos años después (1962). En un trabajo de gran influencia de J. A. Robinson (1965) apareció la regla de resolución general y una demostración de su completitud, en el que también se mostró cómo razonar con lógica de primer orden, sin tener que recurrir a las técnicas proposicionales.

Stephen Cook (1971) demostró que averiguar la *satisfacibilidad* de una sentencia en lógica proposicional es un problema NP-completo. Ya que averiguar la implicación es equivalente a averiguar la *insatisfacibilidad*, éste es un problema co-NP-completo. Se sabe que muchos subconjuntos de la lógica proposicional se pueden resolver en un tiempo polinómico; las cláusulas de Horn son uno de estos subconjuntos. El algoritmo de encadenamiento hacia delante en tiempo lineal para las cláusulas de Horn se debe a Dowling y Gallier (1984), quienes describen su algoritmo como un proceso de flujo de datos parecido a la propagación de señales en un circuito. La *satisfacibilidad* ha sido uno de los ejemplos canónicos para las reducciones NP; por ejemplo, Kaye (2000) demostró que el juego del dragaminas (véase Ejercicio 7.11) es NP-completo.

Los algoritmos de búsqueda local para la *satisfacibilidad* se intentaron por diversos autores en los 80; todos los algoritmos estaban basados en la idea de minimizar las cláusulas insatisfactibles (Hansen y Jaumard, 1990). Un algoritmo particularmente efectivo fue desarrollado por Gu (1989) e independientemente por Selman *et al.* (1992), quien lo lla-

mó GSAT y demostró que era capaz de resolver muy rápidamente un amplio rango de problemas muy duros. El algoritmo SAT-CAMINAR descrito en el capítulo es de Selman *et al.* (1996).

La «transición de fase» en los problemas aleatorios *k*-SAT de *satisfacibilidad* fue identificada por primera vez por Simon y Dubois (1989). Los resultados empíricos de Crawford y Auton (1993) sugieren que se encuentra en el valor del ratio cláusula/variable alrededor de 4,24 para problemas grandes de 3-SAT; este trabajo también describe una implementación muy eficiente del DPLL. (Bayardo y Schrag, 1997) describen otra implementación eficiente del DPLL utilizando las técnicas de satisfacción de restricciones, y (Moskewicz *et al.* 2001) describen el CHAFF, que resuelve problemas de verificación de hardware con millones de variables y que fue el ganador de la Competición SAT 2002. Li y Anbulagan (1997) analizan las heurísticas basadas en la propagación unitaria que permiten que los resolutores sean más rápidos. Cheeseman *et al.* (1991) proporcionan datos acerca de un número de problemas de la misma familia y conjeturan que todos los problemas NP duros tienen una transición de fase. Kirkpatrick y Selman (1994) describen mecanismos mediante los cuales la física estadística podría aclarar de forma precisa las «condiciones» que definen la transición de fase. Los análisis teóricos acerca de su *localización* son todavía muy flojos: todo lo que se puede demostrar es que se encuentra en el rango [3.003, 4.598] para 3-SAT aleatorio. Cook y Mitchell (1997) hacen un excelente repaso de los resultados en este y otros temas relacionados con la *satisfacibilidad*.

Las investigaciones teóricas iniciales demostraron que DPLL tiene una complejidad media polinómica para ciertas distribuciones normales de problemas. Este hecho, potencialmente apasionante, pasó a ser menos apasionante cuando Franco y Paull (1983) demostraron que los mismos problemas se podían resolver en tiempo constante simplemente utilizando asignaciones aleatorias. El método de generación aleatoria descrito en el capítulo tiene problemas más duros. Motivados por el éxito empírico de la búsqueda local en estos problemas, Koutsoupias y Papadimitriou (1992) demostraron que un algoritmo sencillo de ascensión de colina puede resolver muy rápido *casi todas* las instancias del problema de la *satisfacibilidad*; sugiriendo que los problemas duros son poco frecuentes. Más aún, Schöning (1999) presentó una variante aleatoria de GSAT cuyo tiempo de ejecución esperado en el *peor de los casos* era de 1.333" para los problemas 3-SAT (todavía exponencial, pero sustancialmente más rápido que los límites previos para los peores casos). Los algoritmos de *satisfacibilidad* son todavía una área de investigación muy activa; la colección de artículos de Du *et al.* (1999) proporciona un buen punto de arranque.

Los agentes basados en circuitos se remontan al trabajo de gran influencia de McCulloch y Pitts (1943), quienes iniciaron el campo de las redes neuronales. Al contrario del supuesto popular, el trabajo trata de la implementación del diseño de un agente basado en circuitos Booleanos en su cerebro. Sin embargo, los agentes basados en circuitos han recibido muy poca atención en la IA. La excepción más notable es el trabajo de Stan Rosenschein (Rosenschein, 1985; Kaelbling y Rosenschein, 1990), quienes desarrollaron mecanismos para compilar agentes basados en circuitos a partir de las descripciones declarativas del entorno y la tarea. Los circuitos para actualizar las proposiciones almacenadas en registros están íntimamente relacionados con el **axioma del estado sucesor** desarrollado para la lógica de primer orden por Reiter (1991). El trabajo de Rod Brooks

(1986, 1989) demuestra la efectividad de los diseños basados en circuitos para controlar robots (un tema del que nos ocuparemos en el Capítulo 25). Brooks (1991) sostiene que los diseños basados en circuitos son *todo* lo que se necesita en la IA (dicha representación y razonamiento es algo incómodo, costoso, e innecesario). Desde nuestro punto de vista, ningún enfoque es suficiente por sí mismo.

El mundo de *wumpus* fue inventado por Gregory Yob (1975). Irónicamente, Yob lo desarrolló porque estaba aburrido de los juegos basados en una matriz: la topología de su mundo de *wumpus* original era un dodecaedro; nosotros lo hemos retornado a la aburrida matriz. Michael Genesereth fue el primero en sugerir que se utilizara el mundo de *wumpus* para evaluar un agente.



EJERCICIOS

7.1 Describa el mundo de *wumpus* según las características de entorno tarea listadas en el Capítulo 2.

7.2 Suponga que el agente ha avanzado hasta el instante que se muestra en la Figura 7.4(a), sin haber percibido nada en la casilla [1, 1], una brisa en la [2, 1], y un hedor en la [1, 2], y en estos momentos está interesado sobre las casillas [1, 3], [2, 2] y [3, 1]. Cada una de ellas puede tener un hoyo y como mucho en una se encuentra el *wumpus*. Siguiendo el ejemplo de la Figura 7.5, construya el conjunto de los mundos posibles. (Debería encontrar unos 32 mundos.) Marque los mundos en los que la BC es verdadera y aquellos en los que cada una de las siguientes sentencias es verdadera:

$$\begin{aligned}\alpha_2 &= \text{«No hay ningún hoyo en la casilla [2, 2]»} \\ \alpha_3 &= \text{«Hay un } wumpus \text{ en la casilla [1, 3]»}\end{aligned}$$

Por lo tanto, demuestre que $BC \models \alpha_2$ y $BC \models \alpha_3$.

7.3 Considere el problema de decidir si una sentencia en lógica proposicional es verdadera dado un modelo.

- a) Escriba un algoritmo recursivo $\text{VERDADERA-LP}(s, m)$ que devuelva *verdadero* si y sólo si la sentencia s es verdadera en el modelo m (donde el modelo m asigna un valor de verdad a cada símbolo de la sentencia s). El algoritmo debería ejecutarse en tiempo lineal respecto al tamaño de la sentencia. (De forma alternativa, utiliza una versión de esta función del repositorio de código en línea, *online*.)
- b) Dé tres ejemplos de sentencias de las que se pueda determinar si son verdaderas o falsas dado un modelo parcial, que no especifique el valor de verdad de algunos de los símbolos.
- c) Demuestre que el valor de verdad (si lo tiene) de una sentencia en un modelo parcial no se puede determinar, por lo general, eficientemente.
- d) Modifique el algoritmo VERDADERA-LP para que algunas veces pueda juzgar la verdad a partir de modelos parciales, manteniendo su estructura recursiva y tiempo de ejecución lineal. Dé tres ejemplos de sentencias de las cuales *no* se detecte su verdad en un modelo parcial mediante su algoritmo.

- e) Investigue si el algoritmo modificado puede hacer que *IMPLEMENTACIÓN-TV*? sea más eficiente.

7.4 Demuestre cada una de las siguientes aserciones:

- a) α es válido si y sólo si *Verdadero* $\models \alpha$.
- b) Para cualquier α , *Falso* $\models \alpha$.
- c) $\alpha \models \beta$ si y sólo si la sentencia $(\alpha \Rightarrow \beta)$ es válida.
- d) $\alpha \equiv \beta$ si y sólo si la sentencia $(\alpha \Leftrightarrow \beta)$ es válida.
- e) $\alpha \models \beta$ si y sólo si la sentencia $(\alpha \wedge \neg\beta)$ es *insatisfacible*.

7.5 Consideré un vocabulario con sólo cuatro proposiciones, A, B, C, y D. ¿Cuántos modelos hay para las siguientes sentencias?

- a) $(A \wedge B) \vee (B \wedge C)$
- b) $A \vee B$
- c) $A \Leftrightarrow B \Leftrightarrow C$

7.6 Hemos definido cuatro conectivas lógicas binarias.

- a) ¿Hay otras conectivas que podrían ser útiles?
- b) ¿Cuántas conectivas binarias puede haber?
- c) ¿Por qué algunas de ellas no son muy útiles?

7.7 Utilizando un método a tu elección, verifique cada una de las equivalencias de la Figura 7.11.

7.8 Demuestre para cada una de las siguientes sentencias, si es válida, *insatisfacible*, o ninguna de las dos cosas. Verifique su decisión utilizando las tablas de verdad o las equivalencias de la Figura 7.11.

- a) $Humo \Rightarrow Humo$
- b) $Humo \Rightarrow Fuego$
- c) $(Humo \Rightarrow Fuego) \Rightarrow (\neg Humo \Rightarrow \neg Fuego)$
- d) $Humo \vee Fuego \vee \neg Fuego$
- e) $((Humo \wedge Calor) \Rightarrow Fuego) \Leftrightarrow ((Humo \Rightarrow Fuego) \vee (Calor \Rightarrow Fuego))$
- f) $(Humo \Rightarrow Fuego) \Rightarrow ((Humo \wedge Calor) \Rightarrow Fuego)$
- g) $Grande \vee Mudo \vee (Grande \Rightarrow Mudo)$
- h) $(Grande \wedge Mudo) \vee \neg Mudo$

7.9 (Adaptado de Barwise y Etchemendy (1993).) Dado el siguiente párrafo, ¿puede demostrar que el unicornio es un animal mitológico? ¿que es mágico?, ¿que tiene cuernos?

Si el unicornio es un animal mitológico, entonces es inmortal, pero si no es mitológico, entonces es un mamífero mortal. Si el unicornio es inmortal o mamífero, entonces tiene cuernos. El unicornio es mágico si tiene cuernos.

7.10 Cualquier sentencia en lógica proposicional es lógicamente equivalente a la aserción de que no se presenta el caso en que cada mundo posible la haga falsa. Demuestre a partir de esta observación que cualquier sentencia se puede escribir en FNC.

7.11 El muy conocido juego del dragaminas está íntimamente relacionado con el mundo de *wumpus*. Un mundo del dragaminas es una matriz rectangular de N casillas con M minas invisibles esparcidas por la matriz. Cualquier casilla puede ser visitada por el agente; si se encuentra que hay una mina obtiene una muerte instantánea. El dragaminas indica la presencia de minas mostrando en cada casilla visitada el *número* de minas que se encuentran alrededor directa o diagonalmente. El objetivo es conseguir visitar todas las casillas libres de minas.

- a) Dejemos que X_{ij} sea verdadero si y sólo si la casilla $[i, j]$ contiene una mina. Anote en forma de sentencia la aserción de que hay exactamente dos minas adyacentes a la casilla $[1, 1]$ apoyándose en alguna combinación lógica de proposiciones con X_{ij} .
- b) Generalice su aserción de la pregunta (a) explicando cómo construir una sentencia en FNC que aserte que k de n casillas vecinas contienen minas.
- c) Explique de forma detallada, cómo un agente puede utilizar DPLL para demostrar que una casilla contiene (o no) una mina, ignorando la restricción global de que hay exactamente M minas en total.
- d) Suponga que la restricción global se construye mediante su método de la pregunta (b). ¿Cómo depende el número de cláusulas de M y N ? Proponga una variación del DPLL para que la restricción global no se tenga que representar explícitamente.
- e) ¿Algunas conclusiones derivadas con el método de la pregunta (c) son invalidadas cuando se tiene en cuenta la restricción global?
- f) Dé ejemplos de configuraciones de explorar valores que induzcan *dependencias a largo plazo* como la que genera que el contenido de una casilla no explorada daría información acerca del contenido de una casilla bastante distante. [Pista: pruebe primero con un tablero de $N \times 1$.]

7.12 Este ejercicio trata de la relación entre las cláusulas y las sentencias de implicación.

- a) Demuestre que la cláusula $(\neg P_1 \vee \dots \vee \neg P_m \vee Q)$ es lógicamente equivalente a la implicación $(P_1 \wedge \dots \wedge P_m) \Rightarrow Q$.
- b) Demuestre que cada cláusula (sin tener en cuenta el número de literales positivos) se puede escribir en la forma $(P_1 \wedge \dots \wedge P_m) \Rightarrow (Q_1 \vee \dots \vee Q_n)$, donde las P s y las Q s son símbolos proposicionales. Una base de conocimiento compuesta por este tipo de sentencia está en **forma normal implicativa** o **forma de Kowalski**.
- c) Interprete la regla de resolución general para las sentencias en forma normal implicativa.

7.13 En este ejercicio diseñaremos más cosas del agente *wumpus* basado en circuitos.

- a) Escriba una ecuación, similar a la Ecuación (7.4), para la proposición *Flecha*, que debería ser verdadera cuando el agente aún tiene una flecha. Dibuje su circuito correspondiente.
- b) Repita la pregunta (a) para *OrientadoDerecha*, utilizando la Ecuación (7.5) como modelo.

c) Cree versiones de las Ecuaciones 7.7 y 7.8 para encontrar el *wumpus*, y dibuje el circuito.

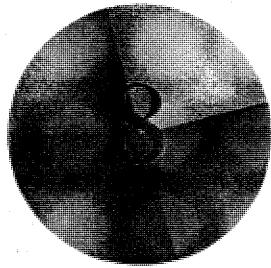


7.14 Discuta lo que quiere significar un comportamiento *óptimo* en el mundo de *wumpus*. Demuestre que nuestra definición de AGENTE-WUMPUS-LP no es óptima, y sugiera mecanismos para mejorarla.

7.15 Amplíe el AGENTE-WUMPUS-LP para que pueda guardar la pista de todos los hechos relevantes que están *dentro* de su base de conocimiento.

7.16 ¿Cuánto se tarda en demostrar que $BC \models \alpha$ utilizando el DPLL cuando α es un literal que ya está en la BC ? Explíquelo.

7.17 Traza el comportamiento del DPLL con la base de conocimiento de la Figura 7.15 cuando intenta demostrar Q , y compare este comportamiento con el del algoritmo del encadenamiento hacia delante.



Lógica de primer orden

Donde nos daremos cuenta de que el mundo está bendecido con muchos objetos, que algunos de los cuales están relacionados con otros objetos, y nos esforzamos en razonar sobre ellos.

LÓGICA DE PRIMER
ORDEN

En el Capítulo 7 demostramos cómo un agente basado en el conocimiento podía representar el mundo en el que actuaba y deducir qué acciones debía llevar a cabo. En el capítulo anterior utilizamos la lógica proposicional como nuestro lenguaje de representación, porque nos bastaba para ilustrar los conceptos fundamentales de la lógica y de los agentes basados en el conocimiento. Desafortunadamente, la lógica proposicional es un lenguaje demasiado endeble para representar de forma precisa el conocimiento de entornos complejos. En este capítulo examinaremos la **lógica de primer orden**¹ que es lo suficientemente expresiva como para representar buena parte de nuestro conocimiento de sentido común. La lógica de predicados también subsume, o forma la base para, muchos otros lenguajes de representación y ha sido estudiada intensamente durante varias décadas. En la Sección 8.1 comenzamos con una breve discusión general acerca de los lenguajes de representación; en la Sección 8.2, se muestra la sintaxis y la semántica de la lógica de primer orden; y en la Sección 8.3 se ilustra el uso de la lógica de primer orden en representaciones sencillas.

8.1 Revisión de la representación

En esta sección, discutiremos acerca de la naturaleza de los lenguajes de representación. Esta discusión nos llevará al desarrollo de la lógica de primer orden, un lenguaje mu-

¹ También denominada **Cálculo de Predicados (de Primer Orden)**, algunas veces se abrevia mediante LP o CP.

cho más expresivo que la lógica proposicional, que introdujimos en el Capítulo 7. Veremos la lógica proposicional y otros tipos de lenguajes para entender lo que funciona y lo que no. Nuestra discusión será rápida, resumiendo siglos del pensamiento humano, de ensayo y error, todo ello en unos pocos párrafos.

Los lenguajes de programación (como C++, o Java, o Lisp) son, de lejos, la clase más amplia de lenguajes formales de uso común. Los programas representan en sí mismos, y de forma directa, sólo procesos computacionales. Las estructuras de datos de los programas pueden representar hechos; por ejemplo, un programa puede utilizar una matriz de 4×4 para representar el contenido del mundo de *wumpus*. De esta manera, una sentencia de un lenguaje de programación como *Mundo*[2, 2] \leftarrow *Hoyo*, es una forma bastante natural de expresar que hay un hoyo en la casilla [2, 2]. (A estas representaciones se les puede considerar *ad hoc*; los sistemas de bases de datos fueron desarrollados para proporcionar una manera más genérica, e independiente del dominio, de almacenar y recuperar hechos.) De lo que carecen los lenguajes de programación, es de algún mecanismo general para derivar hechos de otros hechos; cada actualización de la estructura de datos se realiza mediante un procedimiento específico del dominio, cuyos detalles se derivan del conocimiento acerca del dominio del o de la programadora. Este enfoque **procedural** puede contrastar con la naturaleza declarativa de la lógica proposicional, en la que el conocimiento y la inferencia se encuentran separados, y en la que la inferencia se realiza de forma totalmente independiente del dominio.

Otro inconveniente de las estructuras de datos de los programas (y de las bases de datos, respecto a este tema) es la falta de un mecanismo sencillo para expresar, por ejemplo, «Hay un hoyo en la casilla [2, 2] o en la [3, 1]» o «Si hay un *wumpus* en la casilla [1, 1] entonces no hay ninguno en la [2, 2]». Los programas pueden almacenar un valor único para cada variable, y algunos sistemas permiten el valor «desconocido», pero carecen de la expresividad que se necesita para manejar información incompleta.

COMPOSICIONAL

La lógica proposicional es un lenguaje declarativo porque su semántica se basa en la relación de verdad entre las sentencias y los mundos posibles. Lo que tiene el suficiente poder expresivo para tratar información incompleta, mediante la disyunción y la conjunción. La lógica proposicional presenta una tercera característica que es muy deseable en los lenguajes de representación, a saber, la **composicionalidad**. En un lenguaje composicional, el significado de una sentencia es una función del significado de sus partes. Por ejemplo, « $M_{1,4} \wedge M_{1,2}$ » está relacionada con los significados de « $M_{1,4}$ » y « $M_{1,2}$ ». Sería muy extraño que « $M_{1,4}$ » significara que hay mal hedor en la casilla [1, 4], que « $M_{1,2}$ » significara que hay mal hedor en la casilla [1, 2], y que en cambio, « $M_{1,4} \wedge M_{1,2}$ » significara que Francia y Polonia empataran en el partido de jockey de calificación de la última semana. Está claro que la no composicionalidad repercute en que al sistema de razonamiento le sea mucho más difícil subsistir.

Tal como vimos en el Capítulo 7, la lógica proposicional carece del poder expresivo para describir de forma *precisa* un entorno con muchos objetos. Por ejemplo, estábamos forzados a escribir reglas separadas para cada casilla al hablar acerca de las brisas y de los hoyos, tal como

$$B_{1,1} \Leftrightarrow (H_{1,2} \vee H_{2,1})$$

Por otro lado, en el lenguaje natural parece bastante sencillo decir, de una vez por todas, que «En las casillas adyacentes a hoyos se percibe una pequeña brisa». De alguna manera, la sintaxis y la semántica del lenguaje natural nos hace posible describir el entorno de forma precisa.

De hecho, si lo pensamos por un momento, los lenguajes naturales (como el inglés o el castellano) son muy expresivos. Hemos conseguido escribir casi la totalidad de este libro en lenguaje natural, sólo con lapsos ocasionales en otros lenguajes (incluyendo la lógica, las matemáticas, y los lenguajes de diagramas visuales). En la lingüística y la filosofía del lenguaje hay una larga tradición que ve el lenguaje natural esencialmente como un lenguaje declarativo de representación del conocimiento e intenta definir su semántica formal. Como en un programa de investigación, si tuviera éxito sería de gran valor para la inteligencia artificial, porque esto permitiría utilizar un lenguaje natural (o alguna variación) con los sistemas de representación y razonamiento.

El punto de vista actual sobre el lenguaje natural es que sirve para un propósito algo diferente, a saber, como un medio de **comunicación** más que como una pura representación. Cuando una persona señala y dice, «¡Mira!», el que le oye llega a saber que lo que dice es que Superman finalmente ha aparecido sobre los tejados. Con ello, no queremos decir que la sentencia «¡Mira!» expresa ese hecho. Más bien, que el significado de la sentencia depende tanto de la propia sentencia como del **contexto** al que la sentencia hace referencia. Está claro que uno no podría almacenar una sentencia como «¡Mira!» en una base de conocimiento y esperar recuperar su significado sin haber almacenado también una representación de su contexto, y esto revela la problemática de cómo se puede representar el propio contexto. Los lenguajes naturales también son composicionales (el significado de una sentencia como «Entonces ella lo vio» puede depender de un contexto construido a partir de muchas sentencias precedentes y posteriores a ella). Por último, los lenguajes naturales sufren de la **ambigüedad**, que puede causar ciertos obstáculos en su comprensión. Tal como comenta Pinker (1995): «Cuando la gente piensa acerca de la *primavera*, seguramente no se confunden sobre si piensan acerca de una estación o algo que va *boing?* (y si una palabra se puede corresponder con dos pensamientos, los pensamientos no pueden ser palabras).»

Nuestro enfoque consistirá en adoptar los fundamentos de la lógica proposicional (una semántica composicional declarativa que es independiente del contexto y no ambigua) y construir una lógica más expresiva basada en dichos fundamentos, tomando prestadas de los lenguajes naturales las ideas acerca de la representación, al mismo tiempo que evitando sus inconvenientes. Cuando observamos la sintaxis del lenguaje natural, los elementos más obvios son los nombres y las sentencias nominales que se refieren a los **objetos** (casillas, hoyos, *wumpus*) y los verbos y las sentencias verbales que se refieren a las **relaciones** entre los objetos (en la casilla se percibe una brisa, la casilla es adyacente a, el agente lanza una flecha). Algunas de estas relaciones son **funciones** (relaciones en las que dada una «entrada» se obtiene un solo «valor»). Es fácil empezar a listar ejemplos de objetos, relaciones y funciones:

- Objetos: gente, casas, números, teorías, Ronald McDonald, colores, partidos de béisbol, guerras, siglos...

OBJETOS

RELACIONES

FUNCIONES

PROPIEDADES

- Relaciones: éstas pueden ser relaciones unitarias, o **propiedades**, como ser de color rojo, ser redondo, ser ficticio, ser un número primo, ser multihistoriado..., o relaciones *n*-arias más generales, como ser hermano de, ser más grande que, estar dentro de, formar parte de, tener color, ocurrir después de, ser dueño de uno mismo, o interponerse entre...
- Funciones: el padre de, el mejor amigo de, el tercer turno, uno mayor que, el comienzo de...

Efectivamente, se puede pensar en casi cualquier aserción como una referencia a objetos y propiedades o relaciones. Como los siguientes ejemplos:

- «Uno sumado a dos es igual a tres.»
Objetos: uno, dos, tres, uno sumado a dos; Relaciones: es igual a; Funciones: sumado a. («Uno sumado a dos» es el nombre de un objeto que se obtiene aplicando la función «sumado a» a los objetos «uno» y «dos». Hay otro nombre para este objeto.)
- «Las casillas que rodean al *wumpus* son pestilentes.»
Objetos: *wumpus*, casillas; Propiedad: pestilente; Relación: rodear a.
- «El malvado rey Juan gobernó Inglaterra en 1200.»
Objetos: Juan, Inglaterra, 1200; Relación: gobernar; Propiedades: malvado, rey.

El lenguaje de la **lógica de primer orden**, cuya sintaxis y semántica definiremos en la siguiente sección, está construido sobre objetos y relaciones. Precisamente por este motivo ha sido tan importante para las Matemáticas, la Filosofía y la inteligencia artificial (y en efecto, en el día a día de la existencia humana) porque se puede pensar en ello de forma utilitaria como en el tratamiento con objetos y de las relaciones entre éstos. La lógica de primer orden también puede expresar hechos acerca de *algunos* o *todos* los objetos de un universo de discurso. Esto nos permite representar leyes generales o reglas, tales como el enunciado «Las casillas que rodean al *wumpus* son pestilentes».

La principal diferencia entre la lógica proposicional y la de primer orden se apoya en el **compromiso ontológico** realizado por cada lenguaje (es decir, lo que asume cada uno acerca de la naturaleza de la *realidad*). Por ejemplo, la lógica proposicional asume que hay hechos que suceden o no suceden en el mundo. Cada hecho puede estar en uno de los dos estados: verdadero o falso². La lógica de primer orden asume mucho más, a saber, que el mundo se compone de objetos con ciertas relaciones entre ellos que suceden o no suceden. Las lógicas de propósito específico aún hacen compromisos ontológicos más allá; por ejemplo, la **lógica temporal** asume que los hechos suceden en *tiempos* concretos y que esos tiempos (que pueden ser instantes o intervalos) están ordenados. De esta manera, las lógicas de propósito específico dan a ciertos tipos de objetos (y a los axiomas acerca de ellos) un estatus de «primera clase» dentro de la lógica, más que simplemente definiéndolos en la base de conocimiento. La **lógica de orden superior** ve las relaciones y funciones que se utilizan en la lógi-

COMPROMISO ONTOLÓGICO

LÓGICA TEMPORAL

LÓGICA DE ORDEN SUPERIOR

² A diferencia de los hechos en la **lógica difusa**, que tienen un **grado de verdad** entre 0 y 1. Por ejemplo, la sentencia «Vietnam es una gran ciudad» podría ser verdadera sólo con un grado 0,6 en nuestro mundo.

EL LENGUAJE DEL PENSAMIENTO

Los filósofos y los psicólogos han reflexionado profundamente sobre cómo representan el conocimiento los seres humanos y otros animales. Está claro que la evolución del lenguaje natural ha jugado un papel importante en el desarrollo de esta habilidad en los seres humanos. Por otro lado, muchas evidencias en la Psicología sugieren que los seres humanos no utilizan el lenguaje de forma directa en sus representaciones internas. Por ejemplo ¿cuál de las dos siguientes frases formaba el inicio de la Sección 8.1?

«En esta sección, discutiremos acerca de la naturaleza de los lenguajes de representación...»

«Esta sección cubre el tema de los lenguajes de representación del conocimiento...»

Wanner (1974) encontró que los sujetos hacían la elección acertada en los tests a un nivel casual (cerca del 50 por ciento de las veces) pero que recordaban el contenido de lo que habían leído con un 90 por ciento de precisión. Esto sugiere que la gente procesa las palabras para formar algún tipo de representación no verbal, lo que llamamos **memoria**.

El mecanismo concreto mediante el cual el lenguaje permite la representación y modela las ideas en los seres humanos sigue siendo una incógnita fascinante. La famosa hipótesis de **Sapir-Whorf** sostiene que el lenguaje que hablamos influye profundamente en la manera en que pensamos y tomamos decisiones, en concreto, estableciendo las estructuras de categorías con las que separamos el mundo en diferentes agrupaciones de objetos. Whorf (1956) sostuvo que los esquimales tenían muchas palabras para la nieve, así que tenían una experiencia de la nieve diferente de las personas que hablaban otros idiomas. Algunos lingüistas no están de acuerdo con el fundamento factual de esta afirmación (Pullum (1991) argumenta que el Inuit, el Yupik, y otros lenguajes similares parecen tener un número parecido de palabras que el inglés para los conceptos relacionados con la nieve) mientras que otros apoyan dicha afirmación (Fortescue, 1984). Parece perfectamente comprensible que las poblaciones que tienen una familiaridad mayor con algunos aspectos del mundo desarrollan un vocabulario mucho más detallado en dichos temas, por ejemplo, los entomólogos dividen lo que muchos de nosotros llamamos simplemente *escarabajos* en cientos de miles de especies y además están personalmente familiarizados con muchas de ellas. (El biólogo evolucionista J. B. S. Haldane una vez se quejó de «Una afición desmesurada a los escarabajos» por parte del Creador.) Más aún, los esquiadores expertos tienen muchos términos para la nieve (en polvo, sopa de pescado, puré de patatas, cruda, maíz, cemento, pasta, azúcar, asfalto, pana, pelusa, etcétera) que representan diferencias que a los profanos no nos son familiares. Lo que no está claro es la dirección de la causalidad (¿los esquiadores se dan cuenta de las diferencias sólo por aprender las palabras, o las diferencias surgen de la experiencia individual y llegan a emparejarse con las etiquetas que se están utilizando en el grupo?) Esta cuestión es especialmente importante en el estudio del desarrollo infantil. Hasta ahora disponemos de poco entendimiento acerca del grado en el cual el aprendizaje del lenguaje y el razonamiento están entrelazados. Por ejemplo, ¿el conocimiento del nombre de un concepto, como *licenciado*, hace que nos sea más fácil construir y razonar acerca de conceptos más complejos que engloban a dicho nombre, como *licenciado idóneo*?

ca de primer orden como objetos en sí mismos. Esto nos permite hacer aseraciones acerca de *todas* las relaciones, por ejemplo, uno podría desear definir lo que significa que una relación sea transitiva. A diferencia de las lógicas de propósito específico, la lógica de orden superior es estrictamente más expresiva que la lógica de primer orden, en el sentido de que algunas sentencias de la lógica de orden superior no se pueden expresar mediante un número finito de sentencias de la lógica de primer orden.

COMPROMISOS EPISTEMOLÓGICOS

Una lógica también se puede caracterizar por sus **compromisos epistemológicos** (los posibles estados del conocimiento respecto a cada hecho que la lógica permite). Tanto en la lógica proposicional como en la de primer orden, una sentencia representa un hecho y el agente o bien cree que la sentencia es verdadera, o cree que la sentencia es falsa, o no tiene ninguna opinión. Por lo tanto, estas lógicas tienen tres estados posibles de conocimiento al considerar cualquier sentencia. Por otro lado, los sistemas que utilizan la **teoría de las probabilidades** pueden tener un *grado de creencia*, que va de cero (no se cree en absoluto) a uno (se tiene creencia total)³. Por ejemplo, un agente del mundo de *wumpus* que utilice las probabilidades podría creer que el *wumpus* se encuentra en la casilla [1, 3] con una probabilidad de 0,75. En la Figura 8.1 se resumen los compromisos ontológicos y epistemológicos de cinco lógicas distintas.

En la siguiente sección nos meteremos en los detalles de la lógica de primer orden. Al igual que un estudiante de Física necesita familiarizarse con las Matemáticas, un estudiante de IA debe desarrollar sus capacidades para trabajar con la notación lógica. Por otro lado, *no* es tan importante conseguir una preocupación desmesurada sobre las *especificaciones* de la notación lógica (al fin y al cabo, hay docenas de versiones distintas). Los conceptos principales que hay que tener en cuenta son cómo el lenguaje nos facilita una representación precisa y cómo su semántica nos permite realizar procedimientos sólidos de razonamiento.

Lenguaje	Compromiso ontológico (lo que sucede en el mundo)	Compromiso epistemológico (lo que el agente cree acerca de los hechos)
Lógica proposicional	Hechos	Verdadero/falso/desconocido
Lógica de primer orden	Hechos, objetos, relaciones	Verdadero/falso/desconocido
Lógica temporal	Hechos, objetos, relaciones, tiempos	Verdadero/falso/desconocido
Teoría de las probabilidades	Hechos	Grado de creencia $\in [0, 1]$
Lógica difusa	Hechos con un grado de verdad $\in [0, 1]$	Valor del intervalo conocido

Figura 8.1 Lenguajes formales y sus compromisos ontológicos y epistemológicos.

³ Es importante no confundir el grado de creencia de la teoría de probabilidades con el grado de verdad de la lógica difusa. Realmente, algunos sistemas difusos permiten una incertidumbre (un grado de creencia) acerca de los grados de verdad.

8.2 Sintaxis y semántica de la lógica de primer orden

Comenzamos esta sección especificando de forma más precisa la forma en la que los mundos posibles en la lógica de primer orden reflejan el compromiso ontológico respecto a los objetos y las relaciones. Entonces introducimos los diferentes elementos del lenguaje, explicando su semántica a medida que avanzamos.

Modelos en lógica de primer orden

Recuerde del Capítulo 7 que los modelos en un lenguaje lógico son las estructuras formales que establecen los mundos posibles que se tienen en cuenta. Los modelos de la lógica proposicional son sólo conjuntos de valores de verdad para los símbolos proposicionales. Los modelos de la lógica de primer orden son más interesantes. Primero, ¡estos contienen a los objetos! El **dominio** de un modelo es el conjunto de objetos que contiene; a estos objetos a veces se les denomina **elementos del dominio**. La Figura 8.2 muestra un modelo con cinco objetos: Ricardo Corazón de León, Rey de Inglaterra de 1189 a 1199; su hermano más joven, el malvado Rey Juan, quien reinó de 1199 a 1215; las piernas izquierda de Ricardo y Juan; y una corona.

Los objetos en el modelo pueden estar relacionados de diversas formas. En la figura, Ricardo y Juan son hermanos. Hablando formalmente, una relación es sólo un con-

DOMINIO
ELEMENTOS DEL DOMINIO

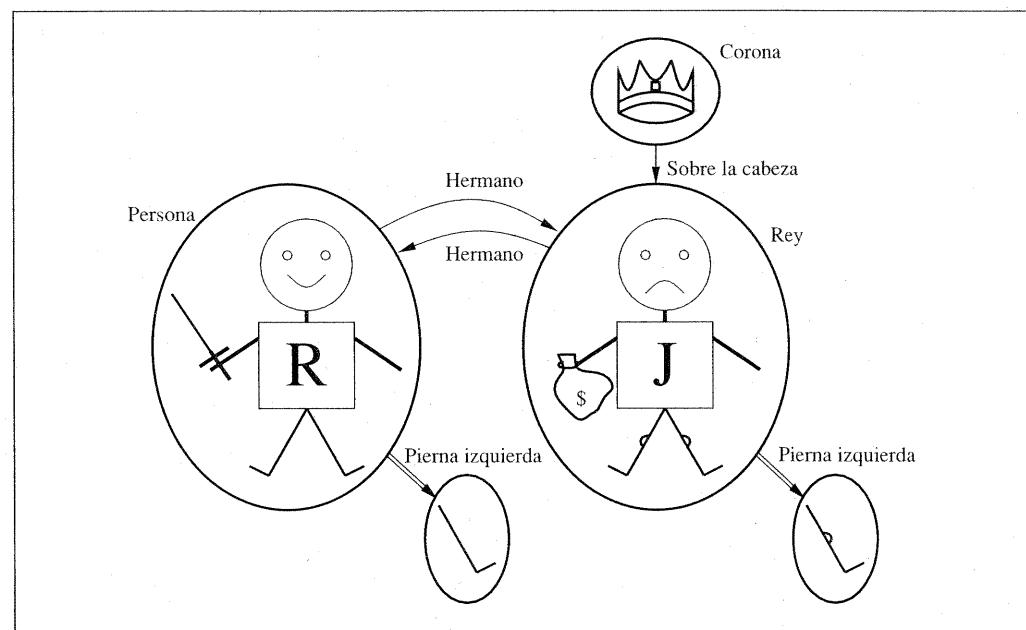


Figura 8.2 Un modelo que contiene cinco objetos, dos relaciones binarias, tres relaciones unitarias (indicadas mediante etiquetas sobre los objetos), y una función unitaria: pierna izquierda.

junto de **tuplas** de objetos que están relacionados. (Una tupla es una colección de objetos colocados en un orden fijo que se escriben entre paréntesis angulares.) De esta manera, la relación de hermandad en este modelo es el conjunto

$$\{\langle \text{Ricardo Corazón de León}, \text{Rey Juan} \rangle, \langle \text{Rey Juan}, \text{Ricardo Corazón de León} \rangle\} \quad (8.1)$$

(Aquí hemos nombrado los objetos en español, pero se puede, si uno lo desea, sustituir mentalmente los nombres por las imágenes.) La corona está colocada sobre la cabeza del Rey Juan, así que la relación «sobre la cabeza» contiene sólo una tupla, $\langle \text{Corona}, \text{Rey Juan} \rangle$. Las relaciones «hermano» y «sobre la cabeza» son relaciones binarias, es decir, relacionan parejas de objetos. El modelo también contiene relaciones unitarias, o propiedades: la propiedad «ser persona» es verdadera tanto para Ricardo como para Juan; la propiedad «ser rey» es verdadera sólo para Juan (presumiblemente porque Ricardo está muerto en este instante); y la propiedad «ser una corona» sólo es verdadera para la corona.

Hay ciertos tipos de relaciones que es mejor que se consideren como funciones; en estas relaciones un objeto dado debe relacionarse exactamente con otro objeto. Por ejemplo, cada persona tiene una pierna izquierda, entonces el modelo tiene la función unitaria «pierna izquierda» con las siguientes aplicaciones:

$$\begin{aligned} \langle \text{Ricardo Corazón de León} \rangle &\rightarrow \text{pierna izquierda de Ricardo} \\ \langle \text{Rey Juan} \rangle &\rightarrow \text{pierna izquierda de Juan} \end{aligned} \quad (8.2)$$

Hablando de forma estricta, los modelos en la lógica de primer orden requieren **funciones totales**, es decir, debe haber un valor para cada tupla de entrada. Así, la corona debe tener una pierna izquierda y por lo tanto, también cada una de las piernas izquierdas. Hay una solución técnica para este problema inoportuno, incluyendo un objeto «invisible» adicional, que es la pierna izquierda de cada cosa que no tiene pierna izquierda, inclusive ella misma. Afortunadamente, con tal de que uno no haga aserciones acerca de piernas izquierdas de cosas que no tienen piernas izquierdas, estos tecnicismos dejan de tener importancia.

Símbolos e interpretaciones

Ahora volvemos a la sintaxis del lenguaje. El lector impaciente puede obtener una descripción completa de la gramática formal de la lógica de primer orden en la Figura 8.3.

Los elementos sintácticos básicos de la lógica de primer orden son los símbolos que representan los objetos, las relaciones y las funciones. Por consiguiente, los símbolos se agrupan en tres tipos: **símbolos de constante**, que representan objetos; **símbolos de predicado**, que representan relaciones; y **símbolos de función**, que representan funciones. Adoptamos la convención de que estos símbolos comiencen en letra mayúscula. Por ejemplo, podríamos utilizar los símbolos de constante *Ricardo* y *Juan*; los símbolos de predicado *Hermano*, *SobreCabeza*, *Persona*, *Rey* y *Corona*; y el símbolo de función *PiernaIzquierda*. Al igual que con los símbolos proposicionales, la selección de los nombres depende enteramente del usuario. Cada símbolo de predicado y de función tiene una **aridad** que establece su número de argumentos.

<i>Sentencia</i>	\rightarrow	<i>SentenciaAtómica</i>
		(<i>Sentencia Conectiva Sentencia</i>)
		<i>Cuantificador Variable... Sentencia</i>
		\neg <i>Sentencia</i>
<i>SentenciaAtómica</i>	\rightarrow	<i>Predicado(Término...) Término = Término</i>
<i>Término</i>	\rightarrow	<i>Función(Término...) Constante Variable</i>
<i>Conectiva</i>	\rightarrow	$\Rightarrow \wedge \vee \Leftrightarrow$
<i>Cuantificador</i>	\rightarrow	$\forall \exists$
<i>Constante</i>	\rightarrow	$A X_1 Juan \dots$
<i>Variable</i>	\rightarrow	$a x s \dots$
<i>Predicado</i>	\rightarrow	<i>AntesDe TieneColor EstáLLloviendo ...</i>
<i>Función</i>	\rightarrow	<i>Madre PiernaIzquierda ...</i>

Figura 8.3 La sintaxis de la lógica de primer orden con igualdad, especificada en BNF. (Mire la página 984 si no estás familiarizado con esta notación.) La sintaxis es estricta con el tema de los paréntesis; los comentarios acerca de los paréntesis y la precedencia de los operadores de la página 230 se aplica de la misma forma a la lógica de primer orden.

INTERPRETACIÓN**INTERPRETACIÓN
DESEADA**

La semántica debe relacionar las sentencias con los modelos para determinar su valor de verdad. Para que esto ocurra, necesitamos de una **interpretación** que especifique exactamente qué objetos, relaciones y funciones son referenciados mediante símbolos de constante, de predicados y de función, respectivamente. Una interpretación posible para nuestro ejemplo (a la que llamaremos **interpretación deseada**) podría ser la siguiente:

- *Ricardo* se refiere a Ricardo Corazón de León y *Juan* se refiere al malvado Rey Juan.
- *Hermano* se refiere a la relación de hermandad, es decir, al conjunto de tuplas de objetos que se muestran en la Ecuación (8.1); *SobreCabeza* se refiere a la relación «sobre la cabeza» que sucede entre la corona y el Rey Juan; *Persona*, *Rey* y *Corona* se refieren a los conjuntos de objetos que son personas, reyes y coronas.
- *PiernaIzquierda* se refiere a la función «pierna izquierda», es decir, la aplicación que se muestra en la Ecuación (8.2).

Hay muchas otras interpretaciones posibles que se relacionan con estos símbolos para este modelo en concreto. Por ejemplo, una interpretación podría relacionar *Ricardo* con la corona y *Juan* con la pierna izquierda del Rey Juan. Hay cinco objetos, por lo tanto hay 25 interpretaciones posibles sólo para los símbolos de constante *Ricardo* y *Juan*. Fíjese en que no todos los objetos necesitan un nombre (por ejemplo, la interpretación deseada no nombra la corona o las piernas). También es posible que un objeto tenga varios

nombres; hay una interpretación en la que tanto *Ricardo* como *Juan* se refieren a la corona. Si encuentra que le confunde esta posibilidad recuerde que en la lógica proposicional es totalmente posible tener un modelo en el que *Nublado* y *Soleado* sean ambos verdaderos; la tarea de la base de conocimiento consiste justamente en excluir lo que es inconsistente con nuestro conocimiento.

El valor de verdad de cualquier sentencia se determina por un modelo y por una interpretación de los símbolos de la sentencia. Por lo tanto, la implicación, la validez, etcétera, se determinan con base en *todos los modelos posibles* y *todas las interpretaciones posibles*. Es importante fijarse en que el número de elementos del dominio en cada modelo puede ser infinito, por ejemplo, los elementos del dominio pueden ser números enteros o reales. Por eso, el número de modelos posibles es infinito, igual que el número de interpretaciones. La comprobación de la implicación mediante la enumeración de todos los modelos posibles, que funcionaba en la lógica proposicional, no es una opción acertada para la lógica de primer orden. Aunque el número de objetos esté restringido, el número de combinaciones puede ser enorme. Con los símbolos de nuestro ejemplo, hay aproximadamente 10^{25} combinaciones para un dominio de cinco objetos. (Véase Ejercicio 8.5.)

Términos

TERMINO

Un **término** es una expresión lógica que se refiere a un objeto. Por lo tanto, los símbolos de constante son términos, pero no siempre es conveniente tener un símbolo distinto para cada objeto. Por ejemplo, en español podríamos utilizar la expresión «la pierna izquierda del Rey Juan», y sería mucho mejor que darle un nombre a su pierna. Para esto sirven los símbolos de función: en vez de utilizar un símbolo de constante utilizamos *PiernaIzquierda(Juan)*. En el caso general, un término complejo está formado por un símbolo de función seguido de una lista de términos entre paréntesis que son los argumentos del símbolo de función. Es importante recordar que un término complejo tan sólo es un tipo de nombre algo complicado. No es una «llamada a una subrutina» que «devuelva un valor». No hay una subrutina *PiernaIzquierda* que tome una persona como entrada y devuelva una pierna. Podemos razonar acerca de piernas izquierdas (por ejemplo haciendo constar que cada uno tiene una pierna y entonces deducir que Juan debe tener una) sin tener que proporcionar una definición de *PiernaIzquierda*. Esto es algo que no se puede hacer mediante subrutinas en los lenguajes de programación⁴.

La semántica formal de los términos es sencilla. Considera un término $f(t_1, \dots, t_n)$. El símbolo de función f se refiere a alguna función del modelo (llamémosla F); los términos argumento se refieren a objetos del dominio (llamémoslos d_1, \dots, d_n); y el término en su globalidad se refiere al objeto que es el valor obtenido de aplicar la función F a los

⁴ Las **expresiones-λ** proporcionan una notación útil mediante la cual se construyen nuevos símbolos de función «al vuelo». Por ejemplo, la función que eleva al cuadrado su argumento se puede escribir como $(\lambda x x \times x)$ y se puede aplicar a argumentos del mismo modo que cualquier otro símbolo de función. Una expresión-λ también se puede definir y utilizar como un símbolo de predicado. (Véase Capítulo 22.) El operador lambda del Lisp juega exactamente el mismo papel. Fíjese en que el uso de λ de este modo *no* aumenta el poder expresivo de la lógica de primer orden; porque cualquier sentencia que tenga una expresión-λ se puede rescribir «enchufando» sus argumentos para obtener una sentencia equivalente.

objetos d_1, \dots, d_n . Por ejemplo, supongamos que el símbolo de función *PiernaIzquierda* se refiere a la función que se muestra en la Ecuación (8.2) y que Juan se refiere al Rey Juan, entonces, *PiernaIzquierda(Juan)* se refiere a la pierna izquierda del Rey Juan. De esta manera, la interpretación establece el referente de cada término.

Sentencias atómicas

Ahora que ya tenemos tanto los términos para referirnos a los objetos, como los símbolos de predicado para referirnos a las relaciones, podemos juntarlos para construir **sentencias atómicas** que representan hechos. Una sentencia atómica está compuesta por un símbolo de predicado seguido de una lista de términos entre paréntesis:

$$\text{Hermano}(\text{Ricardo}, \text{Juan})$$

Esto representa, bajo la interpretación deseada que hemos dado antes, que Ricardo Corazón de León es el hermano del Rey Juan⁵. Las sentencias atómicas pueden tener términos complejos. De este modo,

$$\text{CasadoCon}(\text{Padre}(\text{Ricardo}), \text{Madre}(\text{Juan}))$$

representa que el padre de Ricardo Corazón de León está casado con la madre del Rey Juan (otra vez, bajo la adecuada interpretación).

Una sentencia atómica es verdadera en un modelo dado, y bajo una interpretación dada, si la relación referenciada por el símbolo de predicado sucede entre los objetos referenciados por los argumentos.



Sentencias compuestas

Podemos utilizar las **conectivas lógicas** para construir sentencias más complejas, igual que en la lógica proposicional. La semántica de las sentencias formadas con las conectivas lógicas es idéntica a la de la lógica proposicional. Aquí hay cuatro sentencias que son verdaderas en el modelo de la Figura 8.2, bajo la interpretación deseada:

- ¬Hermano(*PiernaIzquierda(Ricardo)*, Juan)
- Hermano(Ricardo, Juan) \wedge Hermano(Juan, Ricardo)
- Rey(Ricardo) \vee Rey(Juan)
- ¬Rey(Ricardo) \Rightarrow Rey(Juan)

Cuantificadores

Una vez tenemos una lógica que nos permite representar objetos, es muy natural querer expresar las propiedades de colecciones enteras de objetos en vez de enumerar los objetos por su nombre. Los **cuantificadores** nos permiten hacer esto. La lógica de primer orden contiene dos cuantificadores estándar, denominados *universal* y *existencial*.

CUANTIFICADORES

⁵ Por lo general utilizaremos la convención de ordenación de los argumentos $P(x, y)$, que se interpreta como « x es P de y ».

Cuantificador universal (\forall)

Retomemos la dificultad que teníamos en el Capítulo 7 con la expresión de las reglas generales en la lógica proposicional. Las reglas como «Las casillas vecinas al *wumpus* son apestosas» y «Todos los reyes son personas» son el pan de cada día de la lógica de primer orden. En la Sección 8.3 trataremos con la primera de éstas. Respecto a la segunda regla, «Todos los reyes son personas», se escribe en la lógica de primer orden

$$\forall x \text{ } Rey(x) \Rightarrow Persona(x)$$

VARIABLE

TÉRMINO BASE

INTERPRETACIÓN AMPLIADA

generalmente \forall se pronuncia «Para todo...». (Recuerda que la A boca abajo representa «todo».) Así, la sentencia dice, «Para todo x , si x es un rey, entonces x es una persona». Al símbolo x se le llama **variable**. Por convenio, las variables se escriben en minúsculas. Una variable es un término en sí mismo, y como tal, también puede utilizarse como el argumento de una función, por ejemplo, *PiernaIzquierda*(x). Un término que no tiene variables se denomina **término base**.

De forma intuitiva, la sentencia $\forall x P$, donde P es una expresión lógica, dice que P es verdadera para cada objeto x . Siendo más precisos, $\forall x P$ es verdadera en un modelo dado bajo una interpretación dada, si P es verdadera para todas las **interpretaciones ampliadas**, donde cada interpretación ampliada especifica un elemento del dominio al que se refiere x .

Esto suena algo complicado, pero tan sólo es una manera cautelosa de definir el sentido intuitivo de la cuantificación universal. Considere el modelo que se muestra en la Figura 8.2 y la interpretación deseada que va con él. Podemos ampliar la interpretación de cinco maneras:

- $x \rightarrow$ Ricardo Corazón de León,
- $x \rightarrow$ Rey Juan,
- $x \rightarrow$ pierna izquierda de Ricardo,
- $x \rightarrow$ pierna izquierda de Juan,
- $x \rightarrow$ la corona.

La sentencia cuantificada universalmente $\forall x \text{ } Rey(x) \Rightarrow Persona(x)$ es verdadera bajo la interpretación inicial si la sentencia $Rey(x) \Rightarrow Persona(x)$ es verdadera en cada una de las interpretaciones ampliadas. Es decir, la sentencia cuantificada universalmente es equivalente a afirmar las cinco sentencias siguientes:

Ricardo Corazón de León es un rey \Rightarrow Ricardo Corazón de León es una persona.

Rey Juan es un rey \Rightarrow Rey Juan es una persona.

La pierna izquierda de Ricardo es un rey \Rightarrow La pierna izquierda de Ricardo es una persona.

La pierna izquierda de Juan es un rey \Rightarrow La pierna izquierda de Juan es una persona.

La corona es un rey \Rightarrow La corona es una persona.

Vamos a observar cuidadosamente este conjunto de aserciones. Ya que en nuestro modelo el Rey Juan es el único rey, la segunda sentencia aserta que él es una persona, tal como esperamos. Pero, ¿qué ocurre con las otras cuatro sentencias, donde parece que incluso se reivindica acerca de piernas y coronas? Eso forma parte del sentido que tie-

ne «Todos los reyes son personas»? De hecho, las otras cuatro aserciones son verdaderas en el modelo, pero no hacen en absoluto ninguna reivindicación acerca de la naturaleza de persona de las piernas, coronas, o en efecto de Ricardo. Esto es porque ninguno de estos objetos es un rey. Mirando la tabla de verdad de la conectiva \Rightarrow (Figura 7.8) vemos que la implicación es verdadera siempre que su premisa sea falsa (*independientemente* del valor de verdad de la conclusión). Así que, al afirmar una sentencia cuantificada universalmente, que es equivalente a afirmar la lista total de implicaciones individuales, acabamos afirmando la conclusión de la regla sólo para aquellos objetos para los que la premisa es verdadera y no decimos nada acerca de aquellos individuos para los que la premisa es falsa. De este modo, las entradas para la tabla de verdad de la conectiva \Rightarrow son perfectas para escribir reglas generales mediante cuantificadores universales.

Un error común, hecho frecuentemente aún por los lectores más diligentes que han leído este párrafo varias veces, es utilizar la conjunción en vez de la implicación. La sentencia

$$\forall x \text{ } Rey(x) \wedge Persona(x)$$

sería equivalente a afirmar

Ricardo Corazón de León es un rey \wedge Ricardo Corazón de León es una persona
 Rey Juan es un rey \wedge Rey Juan es una persona
 La pierna izquierda de Ricardo es un rey \wedge La pierna izquierda de Ricardo es una persona
 etcétera. Obviamente, esto no plasma lo que queremos expresar.

Cuantificación existencial (\exists)

La cuantificación universal construye enunciados acerca de todos los objetos. De forma similar, utilizando un cuantificador existencial, podemos construir enunciados acerca de *algún* objeto del universo de discurso sin nombrarlo. Para decir, por ejemplo, que el Rey Juan tiene una corona sobre su cabeza, escribimos

$$\exists x \text{ } Corona(x) \wedge SobreCabeza(x, Juan)$$

$\exists x$ se pronuncia «Existe un x tal que...» o «Para algún x ...».

De forma intuitiva, la sentencia $\exists x P$ dice que P es verdadera al menos para un objeto x . Siendo más precisos, $\exists x P$ es verdadera en un modelo dado bajo una interpretación dada si P es verdadera *al menos en una* interpretación ampliada que asigna a x un elemento del dominio. Para nuestro ejemplo, esto significa que al menos una de las sentencias siguientes debe ser verdadera:

Ricardo Corazón de León es una corona \wedge Ricardo Corazón de León está sobre la cabeza de Juan;

Rey Juan es una corona \wedge Rey Juan está sobre la cabeza de Juan;

La pierna izquierda de Ricardo es una corona \wedge La pierna izquierda de Ricardo está sobre la cabeza de Juan;

La pierna izquierda de Juan es una corona \wedge La pierna izquierda de Juan está sobre la cabeza de Juan;

La corona es una corona \wedge La corona está sobre la cabeza de Juan.

La quinta aserción es verdadera en nuestro modelo, por lo que la sentencia original cuantificada existencialmente es verdadera en el modelo. Fíjese en que, según nuestra definición, la sentencia también sería verdadera en un modelo en el que el Rey Juan llevara dos coronas. Esto es totalmente consistente con la sentencia inicial «El Rey Juan tiene una corona sobre su cabeza»⁶.

Igual que el utilizar con el cuantificador \forall la conectiva \Rightarrow parece ser lo natural, \wedge es la conectiva natural para ser utilizada con el cuantificador \exists . Utilizar \wedge como la conectiva principal con \forall nos llevó a un enunciado demasiado fuerte en el ejemplo de la sección anterior; y en efecto, utilizar \Rightarrow con \exists nos lleva a un enunciado demasiado débil. Considere la siguiente sentencia:

$$\exists x \text{ Corona}(x) \Rightarrow \text{SobreCabeza}(x, \text{Juan})$$

Superficialmente, esto podría parecer una interpretación razonable de nuestra sentencia. Al aplicar la semántica vemos que la sentencia dice que al menos una de las aserciones siguientes es verdadera:

Ricardo Corazón de León es una corona \Rightarrow Ricardo Corazón de León está sobre la cabeza de Juan;

Rey Juan es una corona \Rightarrow Rey Juan está sobre la cabeza de Juan;

La pierna izquierda de Ricardo es una corona \Rightarrow La pierna izquierda de Ricardo está sobre la cabeza de Juan;

etcétera. Ahora una implicación es verdadera si son verdaderas la premisa y la conclusión, o si su premisa es falsa. Entonces, si Ricardo Corazón de León no es una corona, entonces la primera aserción es verdadera y se satisface el existencial. Así que, una implicación cuantificada existencialmente es verdadera en cualquier modelo que contenga un objeto para el que la premisa de la implicación sea falsa; de aquí que este tipo de sentencias al fin y al cabo no digan mucho.

Cuantificadores anidados

A menudo queremos expresar sentencias más complejas utilizando múltiples cuantificadores. El caso más sencillo es donde los cuantificadores son del mismo tipo. Por ejemplo, «Los camaradas son hermanos» se puede escribir como

$$\forall x \forall y \text{ Hermano}(x, y) \Rightarrow \text{Camarada}(x, y)$$

⁶ Hay una variante del cuantificador existencial, escrito por lo general \exists^1 o $\exists!$, que significa «Existe exactamente uno.» El mismo significado se puede expresar utilizando sentencias de igualdad, tal como mostraremos en esta misma sección.

Los cuantificadores consecutivos del mismo tipo se pueden escribir como un solo cuantificador con sendas variables. Por ejemplo, para decir que la relación de hermandad es una relación simétrica podemos escribir

$$\forall x, y \text{ } Camarada(x, y) \Rightarrow Camarada(y, x)$$

En otros casos tenemos combinaciones. «Todo el mundo ama a alguien» significa que para todas las personas, hay alguien que esa persona ama:

$$\forall x \exists y Ama(x, y)$$

Por otro lado, para decir «Hay alguien que es amado por todos», escribimos

$$\exists y \forall x Ama(x, y)$$

Por lo tanto, el orden de los cuantificadores es muy importante. Está más claro si introducimos paréntesis. $\forall x (\exists y Ama(x, y))$ dice que *todo el mundo* tiene una propiedad en particular, en concreto, la propiedad de amar a alguien. Por otro lado, $\exists y (\forall x Ama(x, y))$ dice que *alguien* en el mundo tiene una propiedad particular, en concreto, la propiedad de ser amado por todos.

Puede aparecer alguna confusión cuando dos cuantificadores se utilizan con el mismo identificador de variable. Considere la sentencia

$$\forall x [Corona(x) \vee (\exists x Hermano(Ricardo, x))]$$

Aquí la x de *Hermano(Ricardo, x)* está cuantificada existencialmente. La regla es que la variable pertenece al cuantificador más anidado que la mencione; entonces no será el sujeto de cualquier otro cuantificador⁷. Otra forma de pensar en esto es: $\exists x Hermano(Ricardo, x)$ es una sentencia acerca de Ricardo (que él tiene un hermano), no acerca de x ; así que poner $\forall x$ fuera no tiene ningún efecto. Se podría perfectamente haber escrito $\exists z Hermano(Ricardo, z)$. Y como esto puede ser una fuente de confusión, siempre utilizaremos variables diferentes.

Conexiones entre \forall y \exists

Los dos cuantificadores realmente están íntimamente conectados el uno al otro, mediante la negación. Afirmar que a todo el mundo no le gustan las pastinacas es lo mismo que afirmar que no existe alguien a quien le gusten, y viceversa:

$$\forall x \neg Gusta(x, Pastinacas) \text{ es equivalente a } \neg \exists x Gusta(x, Pastinacas).$$

⁷ Es el potencial para la inferencia entre cuantificadores que utilizan el mismo identificador de variable lo que motiva el mecanismo barroco de las interpretaciones ampliadas en la semántica de las sentencias cuantificadas. El enfoque intuitivo más obvio de sustituir los objetos de cada ocurrencia de x falla en nuestro ejemplo porque la x de *Hermano(Ricardo, x)* sería «capturada» por la sustitución. Las interpretaciones ampliadas manejan este tema de forma correcta porque la asignación para x del cuantificador más interiorizado estropea a los cuantificadores externos.

Podemos dar un paso más allá: «A todo el mundo le gusta el helado» significa que no hay nadie a quien no le guste el helado:

$$\forall x \ Gusta(x, \text{Helado}) \text{ es equivalente a } \neg \exists x \ \neg Gusta(x, \text{Helado}).$$

Como \forall realmente es una conjunción sobre el universo de objetos y \exists es una disyunción, no sería sorprendente que obedezcan a las leyes de Morgan. Las leyes de Morgan para las sentencias cuantificadas y no cuantificadas son las siguientes:

$$\begin{array}{ll} \forall x \ \neg P \equiv \neg \exists x \ P & \neg P \wedge \neg Q \equiv \neg(P \vee Q) \\ \neg \forall x \ P \equiv \exists x \ \neg P & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\ \forall x \ P \equiv \neg \exists x \ \neg P & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\ \exists x \ P \equiv \neg \forall x \ \neg P & P \vee Q \equiv \neg(\neg P \wedge \neg Q) \end{array}$$

De este modo, realmente no necesitamos \forall y \exists al mismo tiempo, igual que no necesitamos \wedge y \vee al mismo tiempo. Todavía es más importante la legibilidad que la parquedad, así que seguiremos utilizando ambos cuantificadores.

Igualdad

SÍMBOLO DE IGUALDAD

La lógica de primer orden incluye un mecanismo extra para construir sentencias atómicas, uno que no utiliza un predicado y unos términos como hemos descrito antes. En lugar de ello, podemos utilizar el **símbolo de igualdad** para construir enunciados describiendo que dos términos se refieren al mismo objeto. Por ejemplo,

$$Padre(Juan) = Enrique$$

dice que el objeto referenciado por *Padre(Juan)* y el objeto referenciado por *Enrique* son el mismo. Como una interpretación especifica el referente para cualquier término, determinar el valor de verdad de una sentencia de igualdad consiste simplemente en ver que los referentes de los dos términos son el mismo objeto.

El símbolo de igualdad se puede utilizar para representar hechos acerca de una función dada, tal como hicimos con el símbolo *Padre*, también se puede utilizar con la negación para insistir en que dos términos no son el mismo objeto. Para decir que Ricardo tiene al menos dos hermanos escribiríamos

$$\exists x, y \ Hermano(x, \text{Ricardo}) \wedge Hermano(y, \text{Ricardo}) \wedge \neg(x = y)$$

La sentencia

$$\exists x, y \ Hermano(x, \text{Ricardo}) \wedge Hermano(y, \text{Ricardo})$$

no tiene el significado deseado. En concreto, es verdadero en el modelo de la Figura 8.2, donde Ricardo tiene sólo un hermano. Para verlo, considere las interpretaciones ampliadas en las que x e y son asignadas al Rey Juan. La adición de $\neg(x = y)$ excluye dichos modelos. La notación $x \neq y$ se utiliza a veces como abreviación de $\neg(x = y)$.

8.3 Utilizar la lógica de primer orden

Ahora que hemos definido un lenguaje lógico expresivo, es hora de aprender a utilizarlo. La mejor forma de hacerlo es a través de ejemplos. Hemos visto algunas sentencias sencillas para mostrar los diversos aspectos de la sintaxis lógica; en esta sección proporcionaremos unas representaciones más sistemáticas de algunos **dominios** sencillos. En la representación del conocimiento un dominio es sólo algún ámbito del mundo acerca del cual deseamos expresar algún conocimiento.

Comenzaremos con una breve descripción de la interfaz DECIR/PREGUNTAR para las bases de conocimiento en primer orden. Entonces veremos los dominios de las relaciones de parentesco, de los números, de los conjuntos, de las listas y del mundo de *wumpus*. La siguiente sección contiene un ejemplo mucho más sustancial (sobre circuitos electrónicos) y en el Capítulo 10 cubriremos cada aspecto del universo de discurso.

Aserciones y peticiones en lógica de primer orden

Las sentencias se van añadiendo a la base de conocimiento mediante DECIR, igual que en la lógica proposicional. Este tipo de sentencias se denominan **aserciones**. Por ejemplo, podemos afirmar que Juan es un rey y que los reyes son personas mediante las siguientes sentencias:

$$\begin{aligned} \text{DECIR}(BC, \text{Rey}(Juan)) \\ \text{DECIR}(BC, \forall x \text{ Rey}(x) \Rightarrow \text{Persona}(x)) \end{aligned}$$

Podemos hacer preguntas a la base de conocimiento mediante PREGUNTAR. Por ejemplo,

$$\text{PREGUNTAR}(BC, \text{Rey}(Juan))$$

que devuelve *verdadero*. Las preguntas realizadas con PREGUNTAR se denominan **peticiones u objetivos** (no deben confundirse con los objetivos que se utilizan para describir los estados deseados por el agente). En general, cualquier petición que se implica lógicamente de la base de conocimiento sería respondida afirmativamente. Por ejemplo, dadas las dos aserciones en el párrafo precedente, la petición

$$\text{PREGUNTAR}(BC, \text{Persona}(Juan))$$

también devolvería *verdadero*. También podemos realizar peticiones cuantificadas, tales como

$$\text{PREGUNTAR}(BC, \exists x \text{ Persona}(x)).$$

La respuesta a esta petición podría ser *verdadero*, pero esto no es de ayuda ni es ameno. (Es como responder a «¿Me puedes decir qué hora es?» con un «Sí».) Una petición con variables existenciales es como preguntar «Hay algún x tal que...» y lo resolvemos proporcionando dicha x . La forma estándar para una respuesta de este tipo es una **sustitución o lista de ligaduras**, que es un conjunto de parejas de variable/término. En este

DOMINIOS

AFIRMACIONES

PETICIONES

OBJETIVOS

SUSTITUCIÓN

LISTA DE LIGADURAS

caso en particular, dadas las dos aserciones, la respuesta sería $\{x/Juan\}$. Si hay más de una respuesta posible se puede devolver una lista de sustituciones.

El dominio del parentesco

El primer ejemplo que vamos a tratar es el dominio de las relaciones familiares, o de parentesco. Este dominio incluye hechos como «Isabel es la madre de Carlos» y «Carlos es el padre de Guillermo», y reglas como «La abuela de uno es la madre de su padre».

Está claro que los objetos de nuestro dominio son personas. Tendremos dos predicados unitarios: *Masculino* y *Femenino*. Las relaciones de parentesco (de paternidad, de hermandad, de matrimonio, etcétera) se representarán mediante los predicados binarios: *Padre*, *Hermano Político*, *Hermano*, *Hermana*, *Niño*, *Hija*, *Hijo*, *Esposo*, *Mujer*, *Marido*, *Abuelo*, *Nieto*, *Primo*, *Tía*, y *Tío*. Utilizaremos funciones para *Madre* y *Padre*, porque todas las personas tienen exactamente uno de cada uno (al menos de acuerdo con las reglas de la naturaleza).

Podemos pasar por cada función y predicado, apuntando lo que sabemos en términos de los otros símbolos. Por ejemplo, la madre de uno es uno de los padres y es femenino:

$$\forall x, y \text{ } Madre(y) = x \Leftrightarrow Femenino(x) \wedge Padre(x, y).$$

El marido de uno es un esposo masculino:

$$\forall x, y \text{ } Marido(y, x) \Leftrightarrow Masculino(y) \wedge Esposo(y, x).$$

Masculino y *Femenino* son categorías disjuntas:

$$\forall x \text{ } Masculino(x) \Leftrightarrow \neg Femenino(x).$$

Padre e hijo son relaciones inversas:

$$\forall x, y \text{ } Padre(x, y) \Leftrightarrow Hijo(y, x).$$

Un abuelo es el parente del parente de uno:

$$\forall x, y \text{ } Abuelo(x, y) \Leftrightarrow \exists z \text{ } Padre(x, z) \wedge Padre(z, y).$$

Un hermano es otro parente del parente de uno:

$$\forall x, y \text{ } Hermano(x, y) \Leftrightarrow x \neq y \wedge \exists z \text{ } Padre(z, x) \wedge Padre(z, y).$$

Podríamos seguir con más páginas como esta, y el Ejercicio 8.11 pide que haga justamente eso.

Cada una de estas sentencias se puede ver como un **axioma** del dominio del parentesco. Los axiomas se asocian por lo general con dominios puramente matemáticos (veremos algunos axiomas sobre números en breve) pero la verdad es que se necesitan en todos los dominios. Los axiomas proporcionan la información factual esencial de la cual se pueden derivar conclusiones útiles. Nuestros axiomas de parentesco también son

DEFINICIONES

definiciones; tienen la forma $\forall x, y P(x, y) \Leftrightarrow \dots$. Los axiomas definen la función *Madre* y los predicados *Marido*, *Masculino*, *Padre*, *Abuelo* y *Hermano* en términos de otros predicados. Nuestras definiciones «tocan el fondo» de un conjunto básico de predicados (*Hijo*, *Esposo*, y *Femenino*) sobre los cuales se definen los demás. Esta es una forma muy natural de desarrollar la representación de un dominio, y es análogo a la forma en que los paquetes de *software* se desarrollan a partir de definiciones sucesivas de subrutinas, partiendo de una biblioteca de funciones primitivas. Fíjese en que no hay necesariamente un único conjunto de predicados primitivos; podríamos perfectamente haber utilizado *Padre*, *Esposo* y *Masculino*. En algunos dominios, tal como veremos, no hay un conjunto básico claramente identificable.

TEOREMAS

No todas las sentencias lógicas acerca de un dominio son axiomas. Algunas son **teoremas**, es decir, son deducidas a partir de los axiomas. Por ejemplo, considere la aserción acerca de que la relación de hermandad es simétrica:

$$\forall x, y \text{ Hermano}(x, y) \Leftrightarrow \text{Hermano}(y, x).$$

¿Es un axioma o un teorema? De hecho, es un teorema que lógicamente se sigue de los axiomas definidos para la relación de hermandad. Si PREGUNTAMOS a la base de conocimiento sobre esta sentencia, la base devolvería *verdadero*.

Desde un punto de vista puramente lógico, una base de conocimiento sólo necesita contener axiomas y no necesita contener teoremas, porque los teoremas no aumentan el conjunto de conclusiones que se siguen de la base de conocimiento. Desde un punto de vista práctico, los teoremas son esenciales para reducir el coste computacional para derivar sentencias nuevas. Sin ellos, un sistema de razonamiento tiene que empezar desde el principio cada vez, como si un físico tuviera que volver a deducir las reglas del cálculo con cada problema nuevo.

No todos los axiomas son definiciones. Algunos proporcionan información más general acerca de ciertos predicados sin tener que constituir una definición. Por el contrario, algunos predicados no tienen una definición completa porque no sabemos lo suficiente para caracterizarlos totalmente. Por ejemplo, no hay una manera obvia para completar la sentencia:

$$\forall x \text{ Persona}(x) \Leftrightarrow \dots$$

Afortunadamente, la lógica de primer orden nos permite hacer uso del predicado *Persona* sin definirlo completamente. En lugar de ello, podemos escribir especificaciones parciales de las propiedades que cada persona tiene y de las propiedades que hacen que algo sea una persona:

$$\begin{aligned} \forall x \text{ Persona}(x) &\Leftrightarrow \dots \\ \forall x \dots &\Leftrightarrow \text{Persona}(x) \end{aligned}$$

Los axiomas también pueden ser «tan sólo puros hechos», tal como *Masculino(Jaime)* y *Esposo(Jaime, Laura)*. Este tipo de hechos forman las descripciones de las instancias de los problemas concretos, permitiendo así que se responda a preguntas concretas. Entonces, las respuestas a estas preguntas serán los teoremas que se siguen de los axiomas. A menudo, nos encontramos con que las respuestas esperadas no están disponibles, por ejemplo, de *Masculino(Jorge)* y *Esposo(Jorge, Laura)* esperamos ser capaces de in-

ferir *Femenino(Laura)*; pero esta sentencia no se sigue de los axiomas dados anteriormente. Y esto es una señal de que nos hemos olvidado de algún axioma.

NÚMEROS NATURALES

AXIOMAS DE PEANO

Números, conjuntos y listas

Los números son quizás el ejemplo más gráfico de cómo se puede construir una gran teoría a partir de un núcleo de axiomas diminuto. Aquí describiremos la teoría de los **números naturales**, o la de los enteros no negativos. Necesitamos un predicado *NumNat* que será verdadero para los números naturales; necesitamos un símbolo de constante, 0; y necesitamos un símbolo de función, *S* (sucesor). Los **axiomas de Peano** definen los números naturales y la suma⁸. Los números naturales se definen recursivamente:

$$\begin{aligned} &\text{NumNat}(0) \\ &\forall n \text{ NumNat}(n) \Rightarrow \text{NumNat}(S(n)) \end{aligned}$$

Es decir, 0 es un número natural, y para cada objeto *n*, si *n* es un número natural entonces el *S(n)* es un número natural. Así, los números naturales son el 0, el *S(0)*, el *S(S(0))*, etcétera. También necesitamos un axioma para restringir la función sucesor:

$$\begin{aligned} &\forall n 0 \neq S(n) \\ &\forall m, n m \neq n \Rightarrow S(m) \neq S(n) \end{aligned}$$

Ahora podemos definir la adición (suma) en términos de la función sucesor:

$$\begin{aligned} &\forall m \text{ NumNat}(m) \Rightarrow +(m, 0) = m \\ &\forall m, n \text{ NumNat}(m) \wedge \text{NumNat}(n) \Rightarrow +(S(m), n) = S(+m, n) \end{aligned}$$

INFIXA

PREFIJA

SINTAXIS EDULCORADA

El primero de estos axiomas dice que sumar 0 a cualquier número natural *m* da el mismo *m*. Fíjese en el uso de la función binaria «+» en el término *+(m, 0)*; en las matemáticas habituales el término estaría escrito *m + 0*, utilizando la notación **infija**. (La notación que hemos utilizado para la lógica de primer orden se denomina **prefija**.) Para hacer que nuestras sentencias acerca de los números sean más fáciles de leer permitiremos el uso de la notación infija. También podemos escribir *S(n)* como *n + 1*, entonces el segundo axioma se convierte en

$$\forall m, n \text{ NumNat}(m) \wedge \text{NumNat}(n) \Rightarrow (m + 1) + n = (m + n) + 1$$

Este axioma reduce la suma a la aplicación repetida de la función sucesor.

El uso de la notación infija es un ejemplo de **sintaxis edulcorada**, es decir, una ampliación o abreviación de una sintaxis estándar que no cambia su semántica. Cualquier sentencia que está edulcorada puede «des-edulcorarse» para producir una sentencia equivalente en la habitual lógica de primer orden.

Una vez tenemos la suma, es fácil definir la multiplicación como una suma repetida, la exponentiación como una multiplicación repetida, la división entera y el resto, los

⁸ Los axiomas de Peano también incluyen el principio de inducción, que es una sentencia de lógica segundo orden más que de lógica de primer orden. La importancia de esta diferencia se explica en el Capítulo 9.

números primos, etcétera. De este modo, la totalidad de la teoría de los números (incluyendo la criptografía) se puede desarrollar a partir de una constante, una función, un predicado y cuatro axiomas.

CONJUNTOS

El dominio de los **conjuntos** es tan fundamental para las matemáticas como para el razonamiento del sentido común. (De hecho, es posible desarrollar la teoría de los números con base en la teoría de los conjuntos.) Queremos ser capaces de representar conjuntos individuales, incluyendo el conjunto vacío. Necesitamos un mecanismo para construir conjuntos añadiendo un elemento a un conjunto o tomando la unión o la intersección de dos conjuntos. Querremos saber si un elemento es un miembro de un conjunto, y ser capaces de distinguir conjuntos de objetos que no son conjuntos.

Utilizaremos el vocabulario habitual de la teoría de conjuntos como sintaxis edulcorada. El conjunto vacío es una constante escrita como $\{\}$. Hay un predicado unitario, *Conjunto*, que es verdadero para los conjuntos. Los predicados binarios son $x \in s$ (x es un miembro del conjunto s) y $s_1 \subseteq s_2$ (el conjunto s_1 es un subconjunto, no necesariamente propio, del conjunto s_2). Las funciones binarias son $s_1 \cap s_2$ (la intersección de dos conjuntos), $s_1 \cup s_2$ (la unión de dos conjuntos), y $\{x|s\}$ (el conjunto resultante de añadir el elemento x al conjunto s). Un conjunto posible de axiomas es el siguiente:

1. Los únicos conjuntos son el conjunto vacío y aquellos construidos añadiendo algo a un conjunto:

$$\forall s \text{ } Conjunto(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \text{ } Conjunto(s_2) \wedge s = \{x|s_2\})$$

2. El conjunto vacío no tiene elementos añadidos a él, en otras palabras, no hay forma de descomponer un *ConjuntoVacío* en un conjunto más pequeño y un elemento:

$$\neg \exists x, s \text{ } \{x|s\} = \{\}$$

3. Añadir un elemento que ya pertenece a un conjunto no tiene ningún efecto:

$$\forall x, s \text{ } x \in s \Leftrightarrow s = \{x|s\}$$

4. Los únicos elementos de un conjunto son los que fueron añadidos a él. Expresamos esto recursivamente, diciendo que x es un miembro de s si y sólo si s es igual a algún conjunto s_2 al que se le ha añadido un elemento y , y que y era el mismo elemento que x o que x es un miembro de s_2 :

$$\forall x, s \text{ } x \in s \Leftrightarrow [\exists y, s_2 \text{ } (s = \{y|s_2\} \wedge (x = y \vee x \in s_2))]$$

5. Un conjunto es un subconjunto de otro conjunto si y sólo si todos los miembros del primer conjunto son miembros del segundo conjunto:

$$\forall s_1, s_2 \text{ } s_1 \subseteq s_2 \Leftrightarrow (\forall x \text{ } x \in s_1 \Rightarrow x \in s_2)$$

6. Dos conjuntos son iguales si y sólo si cada uno es subconjunto del otro:

$$\forall s_1, s_2 \text{ } (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1)$$

7. Un objeto pertenece a la intersección de dos conjuntos si y sólo si es miembro de ambos conjuntos:

$$\forall x, s_1, s_2 \text{ } x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2)$$

8. Un objeto pertenece a la unión de dos conjuntos si y sólo si es miembro de alguno de los dos:

$$\forall x, s_1, s_2 \quad x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2)$$

LISTAS

Las **listas** son muy parecidas a los conjuntos. Las diferencias son que las listas están ordenadas y que el mismo elemento puede aparecer más de una vez en una lista. Podemos utilizar el vocabulario del Lisp para las listas: *Nil* es la constante para las listas sin elementos; *Cons*, *Unir*, *Primero* y *Resto* son funciones; y *Encontrar* es el predicado que hace en listas lo que *Miembro* hace en conjuntos. *¿Lista?* es un predicado que es verdadero sólo para las listas. Como en los conjuntos, es común el uso de sintaxis edulcoradas en las sentencias lógicas que tratan sobre listas. La lista vacía es `[]`. El término *Cons(x, y)*, donde *y* es un conjunto no vacío, se escribe `[x|y]`. El término *Cons(x, Nil)*, (por ejemplo, la lista conteniendo el elemento *x*), se escribe `[x]`. Una lista con varios elementos, como `[A, B, C]`, se corresponde al término anidado *Cons(A, Cons(B, Cons(C, Nil)))*. El Ejercicio 8.14 pide que escriba los axiomas para las listas.

El mundo de *wumpus*

En el Capítulo 7 se dieron algunos axiomas para el mundo de *wumpus* en lógica proposicional. Los axiomas en lógica de primer orden de esta sección son mucho más precisos, capturando de forma natural exactamente lo que queremos expresar.

Recuerde que el agente *wumpus* recibe un vector de percepciones con cinco elementos. La sentencia en primer orden correspondiente almacenada en la base de conocimiento debe incluir tanto la percepción como el instante de tiempo en el que ocurrió ésta, de otra manera el agente se confundiría acerca de cuándo vio qué cosa. Utilizaremos enteros para los instantes de tiempo. Una típica sentencia de percepción sería

$$\text{Percepción}([\text{MalHedor}, \text{Brisa}, \text{Resplandor}, \text{Nada}, \text{Nada}], 5)$$

Aquí, *Percepción* es un predicado binario y *MalHedor* y otros son constantes colocadas en la lista. Las acciones en el mundo de *wumpus* se pueden representar mediante términos lógicos:

Girar(Derecha), *Girar(Izquierda)*, *Avanzar*, *Disparar*, *Agarrar*, *Libertar*, *Escalar*

Para hallar qué acción es la mejor, el programa del agente construye una petición como esta

$$\exists a \text{ MejorAcción}(a, 5)$$

PREGUNTAR resolvería esta petición y devolvería una lista de ligaduras como `{a/Agarrar}`. El programa del agente entonces puede devolver *Agarrar* como la acción que debe llevar a cabo, pero primero debe DECIR a la propia base de conocimiento que está ejecutando la acción *Agarrar*.

Los datos acerca de las crudas percepciones implican ciertos hechos acerca del estado actual. Por ejemplo:

$$\forall t, s, g, m, c \text{ Percepción}([s, \text{Brisa}, g, m, c], t) \Rightarrow \text{Brisa}(t),$$

$$\forall t, s, b, m, c \text{ Percepción}([s, b, \text{Resplandor}, m, c], t) \Rightarrow \text{Resplandor}(t),$$

etcétera. Estas reglas muestran una forma trivial del proceso de razonamiento denominado **percepción**, que estudiaremos en profundidad en el Capítulo 24. Fíjese en la cuantificación sobre t . En la lógica proposicional, habríamos necesitado copias de cada sentencia para cada instante de tiempo.

El comportamiento simple de tipo «reflexivo» también puede ser implementado mediante sentencias de implicación cuantificada. Por ejemplo, tenemos

$$\forall t \text{Resplandor}(t) \Rightarrow \text{MejorAcción(Agarrar, } t)$$

Dadas las percepciones y las reglas de los párrafos precedentes, esto nos daría la conclusión deseada $\text{MejorAcción(Agarrar, 5)}$ (es decir, Agarrar es lo más correcto a hacer). Fíjese en la correspondencia entre esta regla y la conexión directa percepción-acción de los agentes basados en circuitos de la Figura 7.20; la conexión en el circuito se cuantifica *implícitamente* sobre el tiempo.

Hasta ahora en esta sección, las sentencias que tratan con el tiempo han sido sentencias **sincrónicas** («al mismo tiempo»), es decir, las sentencias relacionan propiedades del estado del mundo con otras propiedades del mismo estado del mundo. Las sentencias que permiten razonar «a través del tiempo» se denominan **diacrónicas**; por ejemplo, el agente necesita saber combinar la información acerca de sus localizaciones anteriores con la información acerca de la acción que acaba de realizar, para establecer su localización actual. Aplazaremos la discusión acerca de las sentencias diacrónicas hasta el Capítulo 10; por ahora, sólo asuma que las inferencias necesarias se han realizado para los predicados de localización, y otros, dependientes del tiempo.

Hemos representado las percepciones y las acciones; ahora es el momento de representar el propio entorno. Vamos a empezar con los objetos. Los candidatos obvios son las casillas, los hoyos y el *wumpus*. Podríamos nombrar cada casilla ($\text{Casilla}_{1,2}$, etcétera) pero entonces el hecho de que la $\text{Casilla}_{1,2}$ y la $\text{Casilla}_{1,3}$ estén adyacentes tendría que ser un hecho «extra», y necesitaríamos un hecho de este tipo para cada par de casillas. Es mejor utilizar un término complejo en el que la fila y la columna aparezcan como enteros; por ejemplo, simplemente podemos usar la lista de términos $[1, 2]$. La adyacencia entre dos casillas se puede definir mediante

$$\begin{gathered} \forall x, y, a, b, \text{Adyacente}([x, y], [a, b]) \Leftrightarrow \\ [a, b] \in \{[x + 1, y], [x - 1, y], [x, y + 1], [x, y - 1]\} \end{gathered}$$

También podríamos nombrar cada hoyo, pero sería inapropiado por otro motivo: no hay ninguna razón para distinguir a unos hoyos de otros⁹. Es mucho más sencillo utilizar un predicado unario *Hoyo* que es verdadero en las casillas que contengan hoyos. Por último, como sólo hay exactamente un *wumpus*, una constante *Wumpus* es tan buena como un predicado unitario (y quizás más digno para el punto de vista del *wumpus*). El *wumpus* vive exactamente en una casilla, por tanto es una buena idea utilizar una función como *Casa(Wumpus)* para nombrar la casilla. Esto evita por completo el enorme conjunto de

SINCRÓNICA

DIACRÓNICA

⁹ De forma similar, muchos de nosotros no nombramos cada pájaro que vuela sobre nuestras cabezas en sus migraciones a regiones más cálidas en invierno. Un ornitólogo que desea estudiar los patrones de migración, los ratios de supervivencia, etcétera, nombraría cada pájaro por medio de una alarma en su pata, porque cada pájaro debe ser observado.

sentencias que se necesitaban en la lógica proposicional para decir que una casilla en concreto contenía al *wumpus*. (Aún sería mucho peor para la lógica proposicional si hubieran dos *wumpus*.)

La localización del agente cambia con el tiempo, entonces escribiremos $En(Agente, s, t)$ para indicar que el agente se encuentra en la casilla s en el instante t . Dada su localización actual, el agente puede inferir las propiedades de la casilla a partir de las propiedades de su percepción actual. Por ejemplo, si el agente se encuentra en una casilla y percibe una brisa, entonces la casilla tiene una corriente de aire:

$$\forall s, t \ En(Agente, s, t) \wedge Brisa(t) \Rightarrow CorrienteAire(s)$$

Es útil saber si una *casilla* tiene una corriente de aire porque sabemos que los hoyos no pueden desplazarse. Fíjese en que *CorrienteAire* no tiene el argumento del tiempo.

Habiendo descubierto qué casillas tienen brisa (o son apestosas) y, muy importante, las que *no* tienen brisa (o *no* son apestosas), el agente puede deducir dónde están los hoyos (y dónde está el *wumpus*). Hay dos tipos de reglas sincrónicas que podrían permitir sacar este tipo de deducciones:

• Reglas de diagnóstico:

Las reglas de diagnóstico nos llevan de los efectos observados a sus causas ocultas. Para encontrar hoyos, las reglas obvias de diagnóstico dicen que si una casilla tiene una brisa, alguna casilla adyacente debe contener un hoyo, o

$$\forall s \ CorrienteAire(s) \Rightarrow \exists r \ Adyacente(r, s) \wedge Hoyo(r)$$

y si una casilla no tiene una brisa, ninguna casilla adyacente contiene un hoyo¹⁰:

$$\forall s \neg CorrienteAire(s) \Rightarrow \neg \exists r \ Adyacente(r, s) \wedge Hoyo(r)$$

Combinando estas dos reglas, obtenemos la siguiente sentencia bicondicional

$$\forall s \ CorrienteAire(s) \Leftrightarrow \exists r \ Adyacente(r, s) \wedge Hoyo(r) \quad (8.3)$$

REGLAS DE
DIAGNÓSTICO

REGLAS CAUSALES

• Reglas causales:

Las reglas causales reflejan la dirección que se asume de causalidad en el mundo: algunas propiedades ocultas del mundo causan que se generen ciertas percepciones. Por ejemplo, un hoyo causa que todas sus casillas adyacentes tengan una brisa:

$$\forall r \ Hoyo(r) \Rightarrow [\forall s \ Adyacente(s, r) \Rightarrow CorrienteAire(s)]$$

y si todas las casillas adyacentes a una casilla dada no tienen hoyos, la casilla no tiene brisa:

$$\forall s [\forall r \ Adyacente(r, s) \Rightarrow \neg Hoyo(r)] \Rightarrow \neg CorrienteAire(s)$$

¹⁰ Hay una tendencia humana natural en olvidar anotar la información negativa de este tipo. En una conversación esta tendencia es totalmente normal (sería muy extraño decir «Hay dos copas en la mesa y *no hay tres o más*», aunque pensar que «Hay dos copas en la mesa» sigue siendo verdadero, estrictamente hablando, cuando hay tres o más). Retomaremos este tema en el Capítulo 10.

Con algo de esfuerzo, es posible demostrar que estas dos sentencias juntas son equivalentes lógicamente a la sentencia bicondicional de la Ecuación (8.3). También se puede pensar en la propia bicondicional como en una regla causal, porque describe cómo se genera el valor de verdad de *CorrienteAire* a partir del estado del mundo.

RAZONAMIENTO
BASADO EN MODELOS

Los sistemas que razonan con reglas causales se denominan sistemas de **razonamiento basado en modelos**, porque las reglas causales forman un modelo de cómo se comporta el entorno. La diferencia entre el razonamiento basado en modelos y el de diagnóstico es muy importante en muchas áreas de la IA. El diagnóstico médico en concreto ha sido un área de investigación muy activa, en la que los enfoques basados en asociaciones directas entre los síntomas y las enfermedades (un enfoque de diagnóstico) han sido reemplazados gradualmente por enfoques que utilizan un modelo explícito del proceso de la enfermedad y de cómo se manifiesta en los síntomas. Estos temas se presentarán también en el Capítulo 13.



Cualquier tipo de representación que el agente utilice, *si los axiomas describen correctamente y completamente la forma en que el mundo se comporta y la forma en que las percepciones se producen, entonces cualquier procedimiento de inferencia lógica completo inferirá la posible descripción del estado del mundo más robusta, dadas las percepciones disponibles*. Así que el diseñador del agente puede concentrarse en obtener el conocimiento acorde, sin preocuparse demasiado acerca del proceso de deducción. Además, hemos visto que la lógica de primer orden puede representar el mundo de *wumpus*, y no de forma menos precisa que la descripción en lenguaje natural dada en el Capítulo 7.

8.4 Ingeniería del conocimiento con lógica de primer orden

INGENIERÍA DEL CONOCIMIENTO

La sección anterior ilustraba el uso de la lógica de primer orden para representar el conocimiento de tres dominios sencillos. Esta sección describe el proceso general de construcción de una base de conocimiento (un proceso denominado **ingeniería del conocimiento**). Un ingeniero del conocimiento es alguien que investiga un dominio concreto, aprende qué conceptos son los importantes en ese dominio, y crea una representación formal de los objetos y relaciones del dominio. Ilustraremos el proceso de ingeniería del conocimiento en un dominio de circuitos electrónicos, que imaginamos ya es bastante familiar, para que nos podamos concentrar en los temas representacionales involucrados. El enfoque que tomaremos es adecuado para desarrollar bases de conocimiento de *propósito-específico* cuyo dominio se circunscribe cuidadosamente y cuyo rango de peticiones se conoce de antemano. Las bases de conocimiento de *propósito-general*, cuya intención es que apoyen las peticiones de todo el abanico del conocimiento humano, se discutirán en el Capítulo 10.

El proceso de ingeniería del conocimiento

Los proyectos de ingeniería del conocimiento varían ampliamente en su contenido, alcance, y dificultad, pero todos estos proyectos incluyen los siguientes pasos:

1. *Identificar la tarea.* El ingeniero del conocimiento debe delinear el rango de las preguntas que la base de conocimiento debe soportar y los tipos de hechos que estarán disponibles para cada instancia de problema en particular. Por ejemplo, ¿la base de conocimiento *wumpus* necesita ser capaz de escoger las acciones o se requiere que sólo responda a las preguntas acerca del contenido del entorno? ¿Los hechos sensoriales incluirán la localización actual? La tarea determinará qué conocimiento debe ser representado para conectar las instancias de los problemas a las respuestas. Este paso es análogo al proceso REAS del diseño de agentes del Capítulo 2.
2. *Recopilar el conocimiento relevante.* El ingeniero del conocimiento debería ser ya un experto en el dominio, o debería necesitar trabajar con expertos reales para extraer el conocimiento que ellos poseen (en un proceso denominado **adquisición del conocimiento**). En esta fase el conocimiento no se representa formalmente. La idea es entender el alcance de la base de conocimiento, tal como se determinó en la tarea, y entender cómo trabaja realmente el dominio.

Para el ejemplo del mundo de *wumpus*, que está definido por un conjunto artificial de reglas, el conocimiento relevante es fácil de identificar. (Sin embargo, fíjese que la definición de adyacencia no estaba suministrada explícitamente por las reglas del mundo de *wumpus*.) Para los dominios reales, el tema de la relevancia puede ser bastante difícil, por ejemplo, un sistema de simulación de diseños de VLSI podría, o no, necesitar tener en cuenta las pérdidas de capacitación o los efectos de revestimiento.

3. *Decidir el vocabulario de los predicados, funciones y constantes.* Es decir, traducir los conceptos importantes del nivel del dominio a nombres del nivel lógico. Esto involucra a muchas cuestiones de *estilo* de la ingeniería del conocimiento. Al igual que el estilo de la programación, esto puede tener un impacto significativo en el éxito final del proyecto. Por ejemplo, ¿los hoyos estarían representados por objetos o por un predicado unitario aplicado a las casillas? ¿La orientación del agente estaría representada por una función o un predicado? ¿La localización del *wumpus* dependería del tiempo? Una vez se han realizado las elecciones, el resultado es un vocabulario que se conoce por la **ontología** del dominio. La palabra *ontología* indica una teoría concreta sobre la naturaleza del ser o de la existencia de algo. La ontología establece qué tipo de cosas existen, pero no determina sus propiedades e interrelaciones específicas.
4. *Codificar el conocimiento general acerca del dominio.* El ingeniero del conocimiento anota los axiomas para todos los términos del vocabulario. Esto hace que se defina (en todo lo posible) el significado de los términos, permitiendo al experto comprobar el contenido. A menudo, esta fase revela ideas equivocadas o lagunas en el vocabulario, que deberán fijarse volviendo al paso 3 y repitiendo los pasos del proceso.

5. *Codificar una descripción de la instancia de un problema específico.* Si la ontología está bien pensada este paso será fácil. Consistirá en escribir sentencias atómicas sencillas acerca de instancias de conceptos que ya son parte de la ontología. Para un agente lógico, las instancias del problema se obtienen de los sensores, mientras que para una base de conocimiento «no corpórea» se obtienen de sentencias adicionales de la misma manera que un programa tradicional las obtiene de los datos de entrada.
6. *Plantear peticiones al procedimiento de inferencia y obtener respuestas.* Esta es la fase en donde se obtiene la recompensa: podemos dejar al procedimiento de inferencia trabajar sobre los axiomas y los hechos del problema concreto para derivar los hechos que estamos interesados en conocer.
7. *Depurar la base de conocimiento.* Rara vez, las respuestas a las peticiones son correctas en un primer intento. Más concretamente, las respuestas serán correctas *para la base de conocimiento como si fueran escritas*, asumiendo que el procedimiento de inferencia sea sólido, pero no serán las que el usuario estaba esperando. Por ejemplo, si falta un axioma, algunas peticiones no serán respondidas por la base de conocimiento. Esto podría resultar un proceso de depuración considerable. Axiomas ausentes o axiomas que son demasiado débiles se pueden identificar fácilmente fijándonos en los sitios donde la cadena del razonamiento para inesperadamente. Por ejemplo, si la base de conocimiento incluye uno de los axiomas de diagnóstico para hoyos,

$$\forall s \text{ CorrienteAire}(s) \Rightarrow \exists r \text{ Adyacente}(r, s) \wedge \text{Hoyo}(r)$$

pero no el otro axioma, entonces el agente nunca será capaz de demostrar la *ausencia* de hoyos. Los axiomas incorrectos se pueden identificar porque son enunciados falsos acerca del mundo. Por ejemplo, la sentencia

$$\forall x \text{ NumPatas}(x, 4) \Rightarrow \text{Mamífero}(x)$$

es falsa para los reptiles, anfibios, y mucho más importante, para las mesas. *La falsedad de esta sentencia se puede determinar independientemente del resto de la base de conocimiento.* En contraste, un error típico en un programa se parece a éste:

$$\text{offset}^{11} = \text{posición} + 1$$

Es imposible decir si este enunciado es correcto sin mirar el resto del programa para ver si, por ejemplo, *offset* se utiliza para referirse a la posición actual, o a una más allá de la posición actual, o si el valor de posición es cambiado por otro enunciado y así *offset* sería otra vez cambiado.

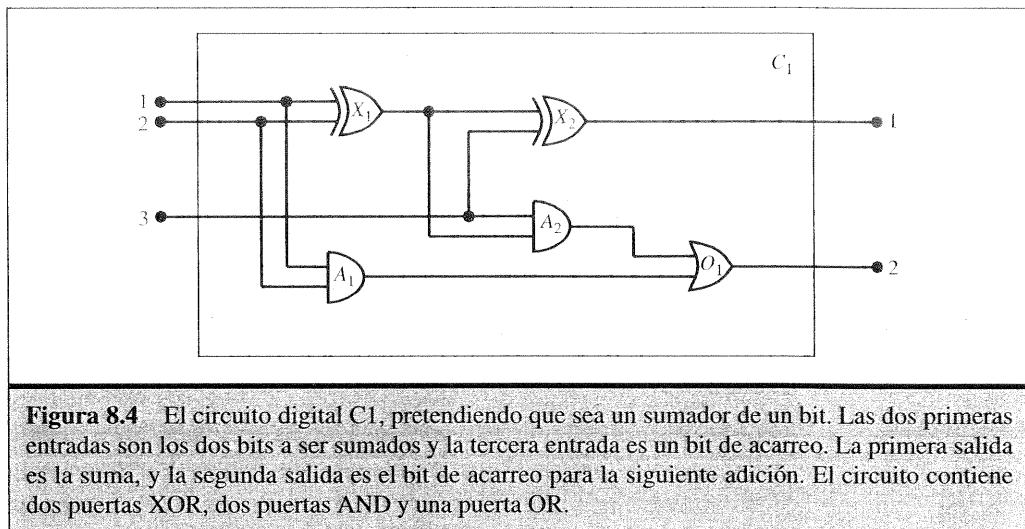
Para entender mejor este proceso de siete pasos vamos a aplicarlo a un ejemplo ampliado: el dominio de los circuitos electrónicos.

El dominio de los circuitos electrónicos

Desarrollaremos una ontología y una base de conocimiento que nos permitirá razonar acerca de los circuitos digitales del tipo como el que se muestra en la Figura 8.4. Seguiremos el proceso de los siete pasos de la ingeniería del conocimiento.

¹¹ En español *compensación*, aunque en tipografía se utiliza el término *offset*. (N del RT.)





Identificar la tarea

Hay muchas tareas de razonamiento relacionadas con los circuitos digitales. En el nivel más alto, uno analiza la funcionalidad del circuito. Por ejemplo, ¿el circuito de la Figura 8.4 realmente suma correctamente? Si todas las entradas son buenas, ¿cuál es la salida de la puerta A2? También son interesantes las preguntas acerca de la estructura del circuito. Por ejemplo, ¿están todas las puertas conectadas a la primera terminal de entrada? ¿El circuito contiene bucles de retroalimentación? Estas serán nuestras tareas en esta sección. Hay más niveles de análisis detallados, incluyendo aquellos relacionados con retardos temporales, superficie del circuito, consumo de corriente, coste de producción, etcétera. Cada uno de estos niveles requeriría de un conocimiento adicional.

Recopilar el conocimiento relevante

¿Qué sabemos acerca de los circuitos digitales? Para nuestros propósitos, los circuitos digitales están compuestos por cables y puertas. La señal circula por los cables a las terminales entrada de las puertas, y cada puerta produce una señal en la terminal salida que circula a través de otro cable. Para determinar cómo serán estas señales, necesitamos saber cómo transforman las puertas su señal de entrada. Hay cuatro tipos de puertas: las puertas AND, OR y XOR tienen dos terminales de entrada, las puertas NOT tienen una. Todas las puertas tienen una terminal de salida. Los circuitos, al igual que las puertas, tienen terminales de entrada y de salida.

Para razonar acerca de la funcionalidad y la conectividad no necesitamos hablar acerca de los propios cables, los recorridos que los cables realizan, o los empalmes en los que dos cables se juntan. Todo lo que nos importa son las conexiones entre los terminales (podemos decir que un terminal de salida está conectado con otro terminal de entrada sin tener que mencionar el cable que realmente los conecta). Hay muchos factores

del dominio que son irrelevantes para nuestro análisis, tales como el tamaño, la forma, el color, o el coste de los diferentes componentes del circuito.

Si nuestro propósito fuera algo distinto a la verificación del diseño al nivel de puertas, la ontología sería diferente. Por ejemplo, si estuviéramos interesados en depurar circuitos defectuosos, entonces probablemente sería una buena idea incluir los cables en la ontología, porque un cable defectuoso puede corromper el flujo de señal que pasa por él. Para resolver defectos de tiempo, necesitaríamos incluir puertas de retardo. Si estuviéramos interesados en diseñar un producto que fuera rentable, entonces serían importantes el coste del circuito y su velocidad relativa a otros productos del mercado.

Decidir el vocabulario

Ahora sabemos que queremos hablar acerca de circuitos, terminales, señales y puertas. El siguiente paso es elegir las funciones, predicados y constantes para representarlos. Comenzaremos a partir de las puertas individuales y ascenderemos a los circuitos.

Lo primero, necesitamos ser capaces de distinguir una puerta de las otras. Esto se controla nombrando las puertas con constantes: X_1, X_2 , etcétera. Aunque cada puerta está conectada en el circuito en su manera particular, su *comportamiento* (la forma en que transforma las señales de entrada en la señal de salida) sólo depende de su *tipo*. Podemos utilizar una función para referirnos al tipo de puerta¹². Por ejemplo, podemos escribir $Tipo(X_1) = XOR$. Esto introduce la constante *XOR* para un tipo concreto de puerta; las otras constantes se llamarán *OR*, *AND* y *NOT*, la función *Tipo* no es la única forma de codificar la distinción ontológica. Podríamos haber utilizado un predicado binario, $Tipo(X_1, XOR)$, o diferentes predicados individuales, como $XOR(X_1)$. Cualquiera de estas elecciones trabajaría correctamente, pero al elegir la función *Tipo* evitamos la necesidad de un axioma que diga que cada puerta individual sólo puede ser de un tipo. La semántica de la función ya lo garantiza.

Ahora consideraremos los terminales. Una puerta o un circuito puede tener uno o más terminales de entrada y uno o más terminales de salida. Simplemente podríamos nombrar cada uno con una constante, igual que hicimos con las puertas. De esta manera, la puerta X_1 podría tener los terminales cuyos nombres serían $X_1Entrada_1, X_1Entrada_2$, y $X_1Salida_1$. Sin embargo, la tendencia a generar nombres compuestos largos se debería evitar. Llamar a algo $X_1Entrada_1$ no hace que sea la primera entrada de X_1 ; necesitaríamos decir esto utilizando una aserción explícita. Probablemente es mejor nombrar la puerta utilizando una función, tal como nombramos la pierna izquierda del Rey Juan con *PiernaIzquierda(Juan)*. Entonces dejemos que *Entrada(1, X₁)* denote el terminal de la primera entrada de la puerta X_1 . Utilizaríamos una función similar *Salida* para los terminales de salida.

La conectividad entre las puertas se puede representar por el predicado *Conectado*, que toma dos terminales como argumentos, como en *Conectado(Salida(1, X₁), Entrada(1, X₂))*.

¹² Fíjese en que hemos utilizado nombres con las letras adecuadas (A_1, X_1 , etcétera) simplemente para hacer más fácil la lectura del ejemplo. La base de conocimiento también debe contener información sobre los tipos de puertas.

Por último, necesitamos saber si una señal está *on* u *off*. Una posibilidad es utilizar un predicado unitario, *On*, que sea verdadero cuando la señal de un terminal es *on*. Sin embargo, esto hace un poco difícil plantear preguntas tales como «¿Cuáles son todos los posibles valores de las señales de los terminales de salida del circuito C_1 ?» Por lo tanto, introduciremos los dos «valores de la señal» como las constantes 1 y 0, y una función *Señal* que tome un terminal como argumento y denote el valor de señal para ese terminal.

Codificar el conocimiento general del dominio

Un síntoma de que poseemos una buena ontología es que haya pocas reglas generales que se necesiten especificar. Un síntoma de que tengamos un buen vocabulario es que cada regla se pueda representar de forma clara y precisa. En nuestro ejemplo, sólo necesitamos siete reglas sencillas para describir cada cosa que necesitamos saber acerca de los circuitos:

1. Si dos terminales están conectados entonces tienen la misma señal:

$$\forall t_1, t_2 \text{ Conectado}(t_1, t_2) \Rightarrow \text{Señal}(t_1) = \text{Señal}(t_2)$$

2. La señal de cada terminal es o bien 1 o bien 0 (pero no ambos):

$$\begin{aligned} \forall t \text{ Señal}(t) = 1 \vee \text{Señal}(t) = 0 \\ 1 \neq 0 \end{aligned}$$

3. El predicado Conectado es conmutativo:

$$\forall t_1, t_2 \text{ Conectado}(t_1, t_2) \Leftrightarrow \text{Conectado}(t_2, t_1)$$

4. La salida de una puerta OR es 1 si y sólo si alguna de sus entradas son 1:

$$\forall g \text{ Tipo}(g) = OR \Rightarrow \text{Señal}(\text{Salida}(1, g)) = 1 \Leftrightarrow \exists n \text{ Señal}(\text{Entrada}(n, g)) = 1$$

5. La salida de una puerta AND es 0 si y sólo si cualquiera de sus entradas es 0:

$$\forall g \text{ Tipo}(g) = AND \Rightarrow \text{Señal}(\text{Salida}(1, g)) = 0 \Leftrightarrow \exists n \text{ Señal}(\text{Entrada}(n, g)) = 0$$

6. La salida de una puerta XOR es 1 si y sólo si sus entradas son diferentes:

$$\begin{aligned} \forall g \text{ Tipo}(g) = XOR \Rightarrow \\ \text{Señal}(\text{Salida}(1, g)) = 1 \Leftrightarrow \text{Señal}(\text{Entrada}(1, g)) \neq \text{Señal}(\text{Entrada}(2, g)) \end{aligned}$$

7. La salida de una puerta NOT es la opuesta de su entrada:

$$\forall g (\text{Tipo}(g) = NOT) \Rightarrow \text{Señal}(\text{Salida}(1, g)) \neq \text{Señal}(\text{Entrada}(1, g))$$

Codificar la instancia del problema específico

El circuito que se muestra en la Figura 8.4 está codificado como el circuito C_1 , con la siguiente descripción. Lo primero que hacemos es categorizar las puertas:

$$\text{Tipo}(X_1) = XOR \quad \text{Tipo}(X_2) = XOR$$

$$\text{Tipo}(A_1) = AND \quad \text{Tipo}(A_2) = AND$$

$$\text{Tipo}(O_1) = OR$$

Entonces, mostramos las conexiones entre ellas:

<i>Conecado(Salida(1, X₁), Entrada(1, X₂))</i>	<i>Conecado(Entrada(1, C₁), Entrada(1, X₁))</i>
<i>Conecado(Salida(1, X₁), Entrada(2, A₂))</i>	<i>Conecado(Entrada(1, C₁), Entrada(1, A₁))</i>
<i>Conecado(Salida(1, A₂), Entrada(1, O₁))</i>	<i>Conecado(Entrada(2, C₁), Entrada(2, X₁))</i>
<i>Conecado(Salida(1, A₁), Entrada(2, O₁))</i>	<i>Conecado(Entrada(2, C₁), Entrada(2, A₁))</i>
<i>Conecado(Salida(1, X₂), Salida(1, C₁))</i>	<i>Conecado(Entrada(3, C₁), Entrada(2, X₂))</i>
<i>Conecado(Salida(1, O₁), Salida(2, C₁))</i>	<i>Conecado(Entrada(3, C₁), Entrada(1, A₂))</i>

Plantear peticiones al procedimiento de inferencia

¿Qué combinación de entradas causaría que la primera salida de C_1 (la suma de bits) sea 0 y que la segunda salida de C_1 (el bit de acarreo) sea 1?

$$\exists i_1, i_2, i_3 \text{ Señal}(\text{Entrada}(1, C_1)) = i_1 \wedge \text{Señal}(\text{Entrada}(2, C_1)) = i_2 \\ \wedge \text{Señal}(\text{Entrada}(3, C_1)) = i_3 \wedge \text{Señal}(\text{Salida}(1, C_1)) = 0 \wedge \text{Señal}(\text{Salida}(2, C_1)) = 1$$

Las respuestas serían las sustituciones de las variables i_1 , i_2 , e i_3 de tal modo que la sentencia resultante esté implicada de la base de conocimiento. Hay tres posibles sustituciones:

$$\{i_1/1, i_2/1, i_3/0\} \quad \{i_1/1, i_2/0, i_3/1\} \quad \{i_1/0, i_2/1, i_3/1\}$$

¿Cuáles son los posibles conjuntos de valores de los terminales para el circuito de suma?

$$\exists i_1, i_2, i_3, o_1, o_2 \text{ Señal}(\text{Entrada}(1, C_1)) = i_1 \wedge \text{Señal}(\text{Entrada}(2, C_1)) = i_2 \\ \wedge \text{Señal}(\text{Entrada}(3, C_1)) = i_3 \wedge \text{Señal}(\text{Salida}(1, C_1)) = o_1 \wedge \text{Señal}(\text{Salida}(2, C_1)) = o_2$$

Esta última petición devuelve una tabla completa de entradas-salidas para el dispositivo, que se puede utilizar para comprobar si en efecto realiza correctamente la suma. Este es un ejemplo sencillo de la **verificación de circuitos**. También podemos utilizar la definición del circuito para construir sistemas digitales más grandes, sobre los cuales, se puede aplicar el mismo tipo de procedimiento de verificación. (Véase Ejercicio 8.17.) Muchos dominios son tratables con el mismo método de desarrollo de una base de conocimiento estructurada, en los que los conceptos más complejos se definen con base en los más sencillos.

VERIFICACIÓN DE
CIRCUITOS

Depurar la base de conocimiento

Podemos perturbar la base de conocimiento de muchas maneras para averiguar qué tipos de comportamientos erróneos emergen. Por ejemplo, suponga que omitimos la asercción acerca de que $1 \neq 0^{13}$. De repente, el sistema será incapaz de demostrar las salidas del circuito, excepto en los casos en los que las entradas sean 000 y 110. Podemos lo-

¹³ Este tipo de omisión es bastante común porque los humanos típicamente asumen que nombres diferentes se refieren a cosas diferentes. Los sistemas de programación lógica, que se describen en el Capítulo 9, también realizan esta asunción.

calizar el problema preguntando por las salidas de cada puerta. Por ejemplo, podemos preguntar

$$\exists i_1, i_2, \text{o } Señal(Entrada(1, C_1)) = i_1 \wedge Señal(Entrada(2, C_1)) = i_2 \wedge Señal(Salida(1, X_1))$$

que revela que no se conocen las salidas de la puerta X_1 para los casos en los que las entradas son 10 y 01. Entonces, observamos el axioma acerca de las puertas XOR, aplicada a la puerta X_1 :

$$Señal(Salida(1, X_1)) = 1 \Leftrightarrow Señal(Entrada(1, X_1)) \neq Señal(Entrada(2, X_1))$$

Si se sabe que las entradas deben ser 1 y 0, entonces esto se reduce a

$$Señal(Salida(1, X_1)) = 1 \Leftrightarrow 1 \neq 0$$

Ahora el problema es aparente: el sistema es incapaz de inferir que $Señal(Salida(1, X_1)) = 1$, por lo tanto, necesitamos decirle que $1 \neq 0$.

8.5 Resumen

En este capítulo hemos introducido la **lógica de primer orden**, un lenguaje de representación que es mucho más potente que la lógica proposicional. Los puntos importantes son los siguientes:

- Los lenguajes de representación del conocimiento deberían ser declarativos, proposicionales, expresivos, independientes del contexto, y no ambiguos.
- Las lógicas difieren en sus **compromisos ontológicos** y **compromisos epistemológicos**. Mientras que la lógica proposicional se compromete sólo con la existencia de hechos, la lógica de primer orden se compromete con la existencia de objetos y sus relaciones, y por ello gana poder expresivo.
- Un **mundo posible**, o **modelo**, se define para la lógica de primer orden como un conjunto de objetos, las relaciones entre ellos y las funciones que se les puede aplicar.
- Los **símbolos de constante** identifican los objetos, los **símbolos de predicado** identifican las relaciones, y los **símbolos de función** identifican las funciones. Una interpretación especifica una aplicación de los símbolos al modelo. Los **términos complejos** aplican símbolos de función a los términos para identificar un objeto. Dados un modelo y una interpretación, se determina el valor de verdad de la sentencia.
- Una **sentencia atómica** consiste en un predicado aplicado a uno o más términos; el predicado es verdadero cuando la relación identificada por el predicado sucede entre los objetos identificados por los términos. Las **sentencias compuestas** utilizan las conectivas como lo hace la lógica proposicional, y las **sentencias cuantificadas** permiten expresar reglas generales.
- Desarrollar una base de conocimiento en lógica de primer orden requiere un proceso cuidadoso para analizar el dominio, escoger el vocabulario y codificar los axiomas que se necesitan para soportar las inferencias deseadas.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

Aunque incluso la lógica de Aristóteles trata con la generalización sobre los objetos, los inicios reales de la lógica de primer orden, con la introducción de los cuantificadores, se dan en el *Begriffschrift* («Escritura de Conceptos» o «Notación Conceptual») de Gottlob Frege (1879). La habilidad de Frege para anidar los cuantificadores fue un gran paso adelante, sin embargo, él utilizaba una notación algo poco elegante (un ejemplo aparece en la portada de este libro). La notación actual de la lógica de primer orden es sustancialmente de Giuseppe Peano (1889), pero la semántica es virtualmente idéntica a la de Frege. Aunque de manera extraña, los axiomas de Peano se debieron en gran medida a Grassmann (1861) y Dedekind (1888).

Una de las barreras más grandes en el desarrollo de la lógica de primer orden ha sido la concentración de esfuerzo en la lógica monádica respecto a la poliádica. Esta fijación sobre los predicados monádicos había sido casi universal en los sistemas lógicos desde Aristóteles hasta Boole. El primer tratamiento sistemático acerca de las relaciones lo realizó Augustus de Morgan (1864), quien citaba el siguiente ejemplo para mostrar los tipos de inferencia que la lógica de Aristóteles no podía manejar: «Todos los caballos son animales; por tanto, la cabeza de un caballo es la cabeza de un animal.» Esta inferencia es inaccesible a Aristóteles porque cualquier regla válida que puede apoyar esta inferencia primero debe analizar la sentencia utilizando el predicado binario « x es la cabeza de y ». La lógica de las relaciones fue estudiada en profundidad por Charles Sanders Peirce (1870), quien también desarrolló la lógica de primer orden independientemente de Frege, aunque un poco más tarde (Peirce, 1883).

Leopold Löwenheim (1915) realizó un tratamiento sistemático de la teoría de modelos para la lógica de primer orden en 1915. Este texto trataba el símbolo de igualdad como una parte integral de la lógica. Los resultados de Löwenheim se desarrollaron bastante por Thoralf Skolem (1920). Alfred Tarski (1935, 1956) dio una definición explícita del valor de verdad y de la satisfacción en la teoría de modelos para la lógica de primer orden, utilizando la teoría de conjuntos.

McCarthy (1958) fue el principal responsable de la introducción de la lógica de primer orden como una herramienta para construir sistemas de IA. El porvenir de la IA basada en la lógica fue avanzando de forma significativa a partir del desarrollo de la resolución de Robinson (1965), un procedimiento completo de inferencia para la lógica de primer orden, que se describe en el Capítulo 9. El enfoque lógico asentó sus raíces en el Stanford. Cordell Green (1969a, 1969b) desarrolló un sistema de razonamiento en lógica de primer orden, QA3, llevando a los primeros intentos de construir un robot lógico en el SRI (Fikes y Nilsson, 1971). La lógica de primer orden se aplicó por Zohar Manna y Richard Waldinger (1971) para razonar acerca de los programas y más tarde por Michael Genesereth (1984) para razonar acerca de los circuitos. En Europa, la programación lógica (una forma restringida del razonamiento en lógica de primer orden) se desarrolló para el análisis lingüístico (Colmerauer *et al.*, 1973) y para sistemas declarativos en general (Kowalski, 1974). La lógica computacional también estuvo bien consolidada en Edinburgh a través del proyecto LCF (Lógica para Funciones Computables) de (Gordon *et al.*, 1979). En los Capítulos 9 y 10 se hace una crónica de estos desarrollos.

Hay un buen número de textos introductorios a la lógica de primer orden. Quine (1982) es uno de los más legibles. Enderton (1972) da una perspectiva orientada más a las matemáticas. Un tratamiento muy formal de la lógica de primer orden, a través de muchos y muy avanzados temas de la lógica, es el proporcionado por Bell y Machover (1977). Manna y Waldinger (1985) dan una introducción asequible a la lógica desde la perspectiva de las ciencias de la computación. Gallier (1986) proporciona una exposición matemática extremadamente rigurosa de la lógica de primer orden, a través del tratamiento de un enorme material sobre su uso en el razonamiento automático. *Logical Foundations of Artificial Intelligence* (Genesereth y Nilsson, 1987) proporciona tanto una introducción sólida a la lógica como el primer tratamiento sistemático de los agentes lógicos, con percepciones y acciones.

EJERCICIOS



8.1 Una base de conocimiento representa el mundo mediante un conjunto de sentencias con una estructura no explícita. Por otro lado, una representación **analógica** tiene una estructura física que se corresponde directamente con la estructura de lo que se representa. Considere un mapa de carreteras de su país como una representación analógica de hechos de ese país. La estructura bidimensional del mapa se corresponde con la estructura bidimensional de la superficie que se analiza.

- a) Dé cinco ejemplos de **símbolos** para el lenguaje del mapa.
- b) Una sentencia *explícita* es una sentencia que el creador de la representación realmente escribe. Una sentencia *implícita* es una sentencia que se obtiene de las sentencias explícitas a partir de las propiedades de la representación analógica. Dé tres ejemplos de sentencias *implícitas* y *explícitas* del lenguaje del mapa.
- c) Dé tres ejemplos de hechos acerca de la estructura física de su país que no se pueden representar en el lenguaje del mapa.
- d) Dé dos ejemplos de hechos que sean más fáciles de expresar en el lenguaje del mapa que en lógica de primer orden.
- e) Dé dos ejemplos de representaciones analógicas útiles. ¿Cuáles son las ventajas y desventajas de cada uno de estos lenguajes?

8.2 Considere una base de conocimiento con tan sólo estas dos sentencias: $P(a)$ y $P(b)$. ¿Esta base de conocimiento implica $\forall x P(x)$? Explique su respuesta en términos de modelos.

8.3 ¿Es válida la sentencia $\exists x, y \ x = y$? Explíquelo.

8.4 Escriba una sentencia lógica tal que en cada mundo en que sea verdadera contenga exactamente un objeto.

8.5 Considere un vocabulario de símbolos que contenga símbolos de constante c , símbolos de predicado p_k para cada aridad k , y símbolos de función f_k para cada aridad k , donde $1 \leq k \leq A$. Permita que el tamaño del dominio esté fijado a D . Para una combinación dada de modelo-interpretación, cada símbolo de predicado o de función

es una aplicación a una relación o función, respectivamente, de la misma aridad. Puede asumir que las funciones en el modelo permiten algunas tuplas de entrada para las cuales la función no da ningún valor (por ejemplo, el valor es un objeto invisible). Derive una fórmula para el número de combinaciones modelo-interpretación para un dominio con D elementos. No se preocupe si debe eliminar combinaciones redundantes.

8.6 Represente las siguientes sentencias en lógica de primer orden, utilizando un vocabulario consistente (que usted debe definir):

- a) Algunos estudiantes estudian francés en la primavera de 2001.
- b) Cada estudiante que estudia francés lo aprueba.
- c) Sólo un estudiante estudia griego en la primavera de 2001.
- d) La mejor puntuación en griego siempre es mayor que la mejor puntuación en francés.
- e) Todas las personas que compran una póliza son inteligentes.
- f) Nadie compra una póliza cara.
- g) Hay un agente que vende pólizas sólo a la gente que no está asegurada.
- h) Hay un barbero que afeita a todos los hombres de la ciudad que no se afeitan ellos mismos.
- i) Una persona nacida en Reino Unido, cuyos padres sean ciudadanos de Reino Unido o residentes en Reino Unido, es un ciudadano de Reino Unido.
- j) Una persona nacida fuera de Reino Unido, que tenga uno de los padres ciudadano de Reino Unido o residente en Reino Unido, es ciudadano de Reino Unido por ascendencia.
- k) Los políticos pueden mentir a algunos todo el tiempo, y pueden mentir a todos algún tiempo, pero no pueden mentir a todos todo el tiempo.

8.7 Represente la sentencia «Todos los alemanes hablan los mismos idiomas» en cálculo de predicados. Utilice $Habla(x, l)$ para indicar que una persona x habla el idioma l .

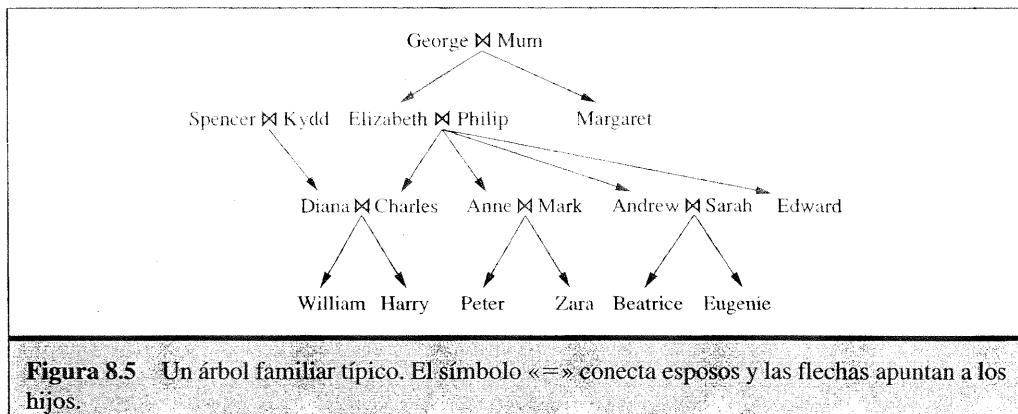
8.8 ¿Qué axioma se necesita para inferir el hecho $Femenino(Laura)$ dados los hechos $Masculino(Jim)$ y $Esposo(Jim, Laura)$?

8.9 Escriba un conjunto general de hechos y axiomas para representar la aserción «Wellington oyó que Napoleón había muerto» y responda correctamente a la pregunta ¿Napoleón oyó que Wellington había muerto?

8.10 Transforme los hechos del mundo de *wumpus* en lógica proposicional de la Sección 7.5 a lógica de primer orden. ¿Cuánto más compacta es esta versión?

8.11 Escriba axiomas describiendo los predicados *Nieto*, *Bisabuelo*, *Hermano*, *Hermana*, *Hija*, *Hijo*, *Tía*, *Tío*, *HermanoPolítico*, *HermanaPolítica* y *PrimoHermano*. Averigüe la definición adecuada del primo $n.^o m, n$ veces extraído, y escriba la definición en lógica de primer orden.

Ahora anote los hechos básicos que están representados en el árbol familiar de la Figura 8.5. Mediante un sistema de razonamiento lógico apropiado, DILE todas las sentencias que ha anotado y PREGÚNTALE quién es el nieto de Elizabeth, los hermanos legales de Diana, y los bisabuelos de Zara.



8.12 Anote una sentencia que aserte que la $+$ es una función conmutativa. ¿Su sentencia se sigue de los axiomas de Peano? Si es así, explique por qué; y si no, dé un modelo en el que los axiomas sean verdaderos y su sentencia falsa.

8.13 Explique qué está incorrecto en la siguiente definición propuesta acerca del predicado de membresía a un conjunto, \in :

$$\begin{aligned} \forall x, s \quad x \in \{x|s\} \\ \forall x, s \quad x \in s \Rightarrow \forall y \quad x \in \{y|s\} \end{aligned}$$

8.14 Utilizando el conjunto de axiomas como ejemplo, escriba axiomas del dominio de las listas, incluyendo todas las constantes, funciones y predicados que se mencionan en el capítulo.

8.15 Explique qué está equivocado en la siguiente definición propuesta acerca de la adyacencia de casillas en el mundo de *wumpus*:

$$\forall x, y \quad \text{Adyacente}([x, y], [x + 1, y]) \wedge \text{Adyacente}([x, y], [x, y + 1])$$

8.16 Escriba los axiomas que se necesitan para razonar acerca de la localización del *wumpus*, utilizando el símbolo de constante *wumpus* y un predicado binario *En(wumpus, Localización)*. Recuerde que tan sólo hay un *wumpus*.



8.17 Amplíe el vocabulario de la Sección 8.4 para definir la adición de números binarios de n -bits. Entonces codifique la descripción del sumador de cuatro bits de la Figura 8.6, y plantee las peticiones que se necesitan para verificar que en efecto se comporta correctamente.

8.18 La representación del circuito en el capítulo es más detallado de lo que se necesita si sólo nos preocupa la funcionalidad del circuito. Una formulación más sencilla describe cualquier puerta, o circuito, de m -entradas y n -salidas, utilizando un predicado con $m + n$ argumentos, de tal manera que el predicado es verdadero cuando las entradas y las salidas son consistentes. Por ejemplo, las puertas NOT se describen mediante un predicado binario *NOT(i, o)* para el cual *NOT(0, 1)* y *NOT(1, 0)* se conocen. Las composiciones de puertas se definen mediante conjunciones de predicados de puertas en las que

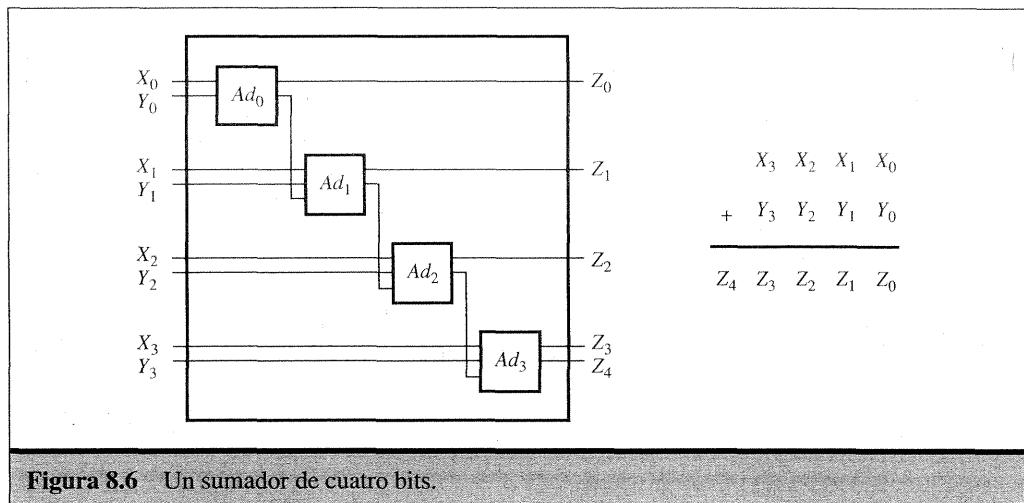


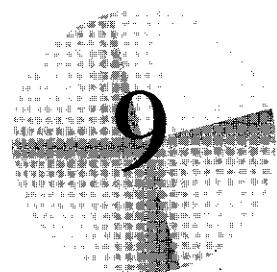
Figura 8.6 Un sumador de cuatro bits.

las variables compartidas indican conexiones directas. Por ejemplo, un circuito NAND puede estar compuesto por puertas AND y puertas NOT:

$$\forall i_1, i_2, o_a, o \quad NAND(i_1, i_2, o) \Leftrightarrow AND(i_1, i_2, o_a) \wedge NOT(o_a, o)$$

Utilizando esta representación, defina el sumador de un bit de la Figura 8.4 y el sumador de cuatro bits de la Figura 8.6, y explique qué peticiones utilizaría para verificar los diseños. ¿Qué tipos de preguntas *no* se soportan mediante esta representación que *sí* se soportan mediante la representación de la Sección 8.4?

8.19 Obtenga una solicitud de pasaporte de su país, identifique las reglas que determinan la aprobación del pasaporte, y tradúzcalas a la lógica de primer orden, siguiendo los pasos perfilados en la Sección 8.4.



Inferencia en lógica de primer orden

Donde definiremos procedimientos eficientes para responder a preguntas planteadas en lógica de primer orden.

En el Capítulo 7 se definió el concepto de inferencia y se demostró cómo se puede llevar a cabo inferencia completa y sólida en lógica proposicional. En este capítulo ampliamos estos temas para conseguir algoritmos que pueden responder a preguntas expresadas en lógica de primer orden. Esto es muy significativo, porque si se trabaja duro, más o menos cualquier cosa se puede representar en lógica de primer orden.

En el Apartado 9.1 introducimos las reglas de inferencia para los cuantificadores y mostramos cómo reducir la inferencia de primer orden a la inferencia proposicional, aunque con un gran coste. En el Apartado 9.2 describimos el concepto de **unificación**, mostrando cómo se puede utilizar para construir reglas de inferencia que trabajen directamente en lógica de primer orden. Entonces discutimos sobre las tres grandes familias de algoritmos de inferencia de primer orden: en el Apartado 9.3 se tratan el **encadenamiento hacia delante** y sus aplicaciones en las **bases de datos deductivas** y en los **sistemas de producción**; en el Apartado 9.4 se desarrollan el **encadenamiento hacia atrás** y los sistemas de **programación lógica**; y en el Apartado 9.5 se describen los sistemas de **demonstración de teoremas** basados en la resolución. Por lo general, uno intenta utilizar el método más eficiente que se pueda acomodar a los hechos y axiomas que se necesitan expresar. El razonamiento con sentencias totalmente generales en lógica de primer orden mediante la resolución suele ser menos eficiente que el razonamiento con cláusulas positivas mediante el encadenamiento hacia delante o hacia atrás.

9.1 Lógica proposicional vs. Lógica de primer orden

Esta sección y la siguiente introducen las ideas sobre las que se basan los sistemas actuales de inferencia lógicos. Comenzamos con algunas reglas de inferencia sencillas que se pueden aplicar a las sentencias con cuantificadores para obtener sus sentencias equivalentes, sin cuantificar. Estas reglas nos conducen de forma natural a la idea de que la inferencia *de primer orden* se puede hacer convirtiendo la base de conocimiento a lógica *proposicional* y utilizando la inferencia *proposicional*. La siguiente sección nos muestra un atajo que es obvio, ir hacia los métodos de inferencia que manipulen directamente las sentencias en lógica de primer orden.

Reglas de inferencia para cuantificadores

Vamos a comenzar con los cuantificadores universales. Suponga que nuestra base de conocimiento contiene el axioma popular que afirma que los reyes que son codiciosos también son malvados.

$$\forall x \text{ Rey}(x) \wedge \text{Codicioso}(x) \Rightarrow \text{Malvado}(x).$$

Entonces parece bastante permisible inferir cualquiera de las siguientes sentencias:

$$\text{Rey}(\text{Juan}) \wedge \text{Codicioso}(\text{Juan}) \Rightarrow \text{Malvado}(\text{Juan}).$$

$$\text{Rey}(\text{Ricardo}) \wedge \text{Codicioso}(\text{Ricardo}) \Rightarrow \text{Malvado}(\text{Ricardo}).$$

$$\text{Rey}(\text{Padre}(\text{Juan})) \wedge \text{Codicioso}(\text{Padre}(\text{Juan})) \Rightarrow \text{Malvado}(\text{Padre}(\text{Juan})).$$

⋮

ESPECIFICACIÓN UNIVERSAL

La regla de la **Especificación Universal** (EU para abreviar) dice que podemos inferir cualquier sentencia obtenida por sustitución de la variable por un **término base** (un término sin variables)¹. Para anotar la regla de inferencia formalmente utilizamos el concepto de **sustitución** que se introdujo en el Apartado 8.3. Vamos a denotar el resultado de aplicar una sustitución θ a una sentencia α mediante $\text{SUST}(\theta, \alpha)$. Entonces la regla se escribe

$$\frac{\forall v \ \alpha}{\text{SUST}(\{v/g\}, \alpha)}$$

para cualquier variable v y término base g . Por ejemplo, las tres sentencias mostradas anteriormente se obtienen con las sustituciones $\{x/\text{Juan}\}$, $\{x/\text{Ricardo}\}$, y $\{x/\text{Padre}(\text{Juan})\}$.

ESPECIFICACIÓN EXISTENCIAL

La correspondiente regla de **Especificación Existencial** para el cuantificador existencial es ligeramente más complicada. Para cualquier sentencia α , variable v , y símbolo de constante k que no aparezca en ninguna otra parte de la base de conocimiento,

$$\frac{\exists v \ \alpha}{\text{SUST}(\{v/k\}, \alpha)}$$

¹ No confunda estas sustituciones con las interpretaciones ampliadas utilizadas para definir la semántica de los cuantificadores. La sustitución reemplaza una variable por un término (una pieza de la sintaxis) para producir una nueva sentencia, mientras que una interpretación es una aplicación de una variable a un objeto del dominio de discurso.

Por ejemplo, la sentencia

$$\exists x \text{ Corona}(x) \wedge \text{SobreCabeza}(x, Juan)$$

podemos inferir la sentencia

$$\text{Corona}(C_1) \wedge \text{SobreCabeza}(C_1, Juan)$$

mientras que C_1 no aparezca en ningún otro sitio de la base de conocimiento. Básicamente, la sentencia existencial nos dice que hay algún objeto que satisface una condición, y el proceso de especificación tan sólo le da un nombre a dicho objeto. Naturalmente, ese nombre no puede pertenecer a otro objeto previamente. Las matemáticas nos proporcionan un delicioso ejemplo: suponga que descubrimos que hay un número que es un poco mayor que 2,71828 y que satisface la ecuación $d(x^y)/dy = x^y$ para x . Le podemos dar a dicho número un nombre, como e , pero sería un error darle el nombre de un objeto ya existente, como π . En lógica, a este nuevo nombre se le denomina **constante de Skolem**. La Especificación Existencial es un caso especial del proceso más general llamado **skolemización**, que trataremos en el Apartado 9.5.

CONSTANTE DE SKOLEM

EQUIVALENCIA INFERENCIAL

Así como es más complicada que la Especificación Universal, la Especificación Existencial representa un papel algo diferente en la inferencia. Mientras que la Especificación Universal se puede aplicar muchas veces para producir muchas consecuencias diferentes, la Especificación Existencial sólo se puede aplicar una vez, y entonces se puede descartar la sentencia cuantificada existencialmente. Por ejemplo, una vez que hemos añadido la sentencia $Mata(\text{Asesino}, \text{Víctima})$, ya no necesitamos más la sentencia $\exists x \text{ Mata}(x, \text{Víctima})$. Hablando de forma estricta, la nueva base de conocimiento no es equivalente lógicamente a la antigua, pero se puede demostrar que es **equivalente inferencialmente** en el sentido que es *satisfacible* justamente cuando lo es la base de conocimiento original.

Reducción a la inferencia proposicional

Una vez que tenemos las reglas para inferir sentencias no cuantificadas a partir de sentencias cuantificadas, nos es posible reducir la inferencia de primer orden a la inferencia proposicional. En esta sección daremos las principales ideas; los detalles se verán en el Apartado 9.5.

La primera idea consiste en que como una sentencia cuantificada existencialmente se puede sustituir por una especificación, una sentencia cuantificada existencialmente se puede sustituir por el conjunto de *todas las especificaciones posibles*. Por ejemplo, suponga que nuestra base de conocimiento contiene tan sólo las sentencias

$$\begin{aligned} &\forall x \text{ Rey}(x) \wedge \text{Codicioso}(x) \Rightarrow \text{Malvado}(x) \\ &\text{Rey}(Juan) \\ &\text{Codicioso}(Juan) \\ &\text{Hermano}(Ricardo, Juan) \end{aligned} \tag{9.1}$$

Entonces aplicamos la EU a la primera sentencia, utilizando todas las sustituciones de términos base posibles, tomadas del vocabulario de la base de conocimiento, en este caso $\{x/Juan\}$ y $\{x/Ricardo\}$. Obtenemos

$$\begin{aligned} &\text{Rey}(Juan) \wedge \text{Codicioso}(Juan) \Rightarrow \text{Malvado}(Juan), \\ &\text{Rey}(Ricardo) \wedge \text{Codicioso}(Ricardo) \Rightarrow \text{Malvado}(Ricardo), \end{aligned}$$

y descartamos la sentencia cuantificada universalmente. Ahora, la base de conocimiento es esencialmente proposicional, si vemos las sentencias atómicas base (*Rey(Juan)*, *Codicioso(Juan)*, etc.) como símbolos proposicionales. Por tanto, podemos aplicar cualquiera de los algoritmos proposicionales completos del Capítulo 7, para obtener conclusiones como *Malvado(Juan)*.

PROPOSICIONALIZACIÓN

Esta técnica de **proposicionalización** se puede hacer que sea completamente general, tal como mostraremos en el Apartado 9.5; es decir, toda base de conocimiento, y petición, en lógica de primer orden se puede transformar a forma proposicional de tal manera que se mantenga la relación de implicación. De este modo, tenemos un procedimiento de decisión completo para la implicación... o quizás no. Hay un problema: ¡cuando la base de conocimiento incluye un símbolo de función, el conjunto de sustituciones de los términos base es infinito! Por ejemplo, si la base de conocimiento tiene el símbolo *Padre*, entonces se pueden construir términos anidados infinitamente, como *Padre(Padre(Padre(Juan)))*). Nuestros algoritmos proposicionales tendrían serias dificultades con tal conjunto de sentencias infinitamente grande.

Afortunadamente, hay un famoso teorema de Jacques Herbrand (1930) que dice que si una sentencia se implica de una base de conocimiento de primer orden, entonces hay una demostración que involucra tan sólo a un subconjunto finito de la base de conocimiento transformada a proposicional. Ya que cada subconjunto de este tipo tiene un máximo de profundidad en la anidación de sus términos base, podemos encontrar el subconjunto generando primero todas las especificaciones al nivel de los símbolos de constante (*Ricardo* y *Juan*), luego con el nivel siguiente de profundidad, o profundidad 1, (*Padre(Ricardo)* y *Padre(Juan)*), luego con los de nivel de profundidad 2, etc., hasta que seamos capaces de construir una demostración proposicional de la sentencia implicada.

Hemos esbozado un enfoque de inferencia en lógica de primer orden mediante la proposicionalización que es **completo**, es decir, cualquier sentencia implicada se puede demostrar. Esto es un importante logro, dado que el espacio de los modelos posibles es infinito. Por otro lado, ¡no podemos saber que la sentencia se implica hasta que la demostración se ha realizado! ¿Qué ocurre cuando la sentencia no se implica? ¿Podemos decir algo? Bueno, en lógica de primer orden, resulta que no podemos. Nuestro procedimiento de demostración podría continuar, generando más y más términos profundamente anidados, pero no sabremos si se ha atascado en un bucle inútil o si la demostración está a punto de acabar. Algo que se parece mucho al problema de la parada en las máquinas de Turing. Alan Turing (1936) y Alonzo Church (1936) demostraron, mediante caminos más bien diferentes, lo inevitable de este tipo de cosas. *El problema de la implicación en lógica de primer orden es semidecidible*, es decir, existen algoritmos que responden afirmativamente para cada sentencia implicada, pero no existe ningún algoritmo que también responda ante una sentencia no implicada.



9.2 Unificación y sustitución

La sección anterior describía cómo la comprensión de la inferencia en lógica de primer orden parte de los inicios de los 60. El lector observador (y seguramente los lógicos com-

putacionales de los inicios de los 60) se habrá dado cuenta de que el enfoque de la proposicionalización es más bien ineficiente. Por ejemplo, dada la petición $Malvado(x)$ y la base de conocimiento de la Ecuación (9.1) parece algo obstinado generar sentencias como $Rey(Juan) \wedge Codicioso(Juan) \Rightarrow Malvado(Juan)$. Sin embargo, la inferencia de $Malvado(Juan)$ a partir de las sentencias

$$\begin{aligned} & \forall x \ Rey(x) \wedge Codicioso(x) \Rightarrow Malvado(x) \\ & Rey(Juan) \\ & Codicioso(Juan) \end{aligned}$$

parece completamente obvio para un ser humano. Ahora mostraremos cómo hacerlo completamente obvio para un computador.

Una regla de inferencia de primer orden

La inferencia de que Juan es malvado se obtiene de la siguiente manera: encontrar un x tal que x sea rey y x sea codicioso, y entonces inferir que ese x es malvado. De forma más general, si hay alguna sustitución θ que haga que las premisas de la implicación sean idénticas a algunas de las sentencias que ya están en la base de conocimiento, entonces podemos asertar la conclusión de la implicación, después de aplicar θ . En este caso, la sustitución $\{x/Juan\}$ logra ese objetivo.

En realidad podemos hacer que el paso de inferencia aún trabaje más. Suponga que en vez de conocer $Codicioso(Juan)$, sabemos que *todo el mundo* es codicioso:

$$\forall y \ Codicioso(y) \quad (9.2)$$

entonces también seríamos capaces de concluir que $Malvado(Juan)$, porque sabemos que Juan es un rey (dado) y que Juan es codicioso (porque todo el mundo lo es). Lo que necesitamos para este trabajo es encontrar una sustitución para las variables en la sentencia de implicación y las variables de las sentencias que se deben emparejar. En este caso, aplicando la sustitución $\{x/Juan, y/Juan\}$ a las premisas de la implicación $Rey(x)$ y $Codicioso(x)$ y a las sentencias de la base de conocimiento $Rey(Juan)$ y $Codicioso(y)$ las hará idénticas. De este modo podemos inferir la conclusión de la implicación.

Este proceso de inferencia se puede plasmar mediante una única regla de inferencia a la que llamamos **Modus Ponens Generalizado**: para las sentencias atómicas p_i, p'_i , y q , donde hay una sustitución θ tal que $SUST(\theta, p'_i) = SUST(\theta, p_i)$, para todo i ,

$$\frac{p'_1, p'_2, \dots, p'_n, (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{SUST(\theta, q)}$$

Hay $n + 1$ premisas para esta regla: las n p'_i sentencias atómicas y la implicación. La conclusión es el resultado de aplicar la sustitución θ al consecuente q . En nuestro ejemplo:

$$\begin{array}{ll} p'_1 \text{ es } Rey(Juan) & p_1 \text{ es } Rey(x) \\ p'_2 \text{ es } Codicioso(y) & p_2 \text{ es } Codicioso(x) \\ \theta \text{ es } \{x/Juan, y/Juan\} & q \text{ es } Malvado(x) \\ SUST(\theta, q) \text{ es } Malvado(Juan) & \end{array}$$

Es fácil demostrar que el Modus Ponens Generalizado es una regla de inferencia sólida. Primero, observamos que cualquier sentencia p (cuyas variables asumimos que están cuantificadas universalmente) y para cualquier sustitución θ ,

$$p \models \text{SUST}(\theta, p)$$

Esto se sostiene sobre los mismos fundamentos de la Especificación Universal. Y se sostiene en concreto para una sustitución θ que satisface las condiciones de la regla del Modus Ponens Generalizado. De este modo, de p_1', \dots, p_n' podemos inferir

$$\text{SUST}(\theta, p_1') \wedge \dots \wedge \text{SUST}(\theta, p_n')$$

y de la implicación $p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q$ podemos inferir

$$\text{SUST}(\theta, p_1) \wedge \dots \wedge \text{SUST}(\theta, p_n) \Rightarrow \text{SUST}(\theta, q)$$

Ahora bien, θ se define en el Modus Ponens Generalizado como $\text{SUST}(\theta, p_i') = \text{SUST}(\theta, p_i)$, para todo i ; por lo tanto la primera sentencia se empareja exactamente con la premissa de la segunda. De aquí, que $\text{SUST}(\theta, q)$ se siga del Modus Ponens.

ELEVACIÓN

El Modus Ponens Generalizado es una versión **elevada** del Modus Ponens: erige el Modus Ponens proposicional a la lógica de primer orden. En lo que queda del capítulo veremos que podemos desarrollar versiones elevadas del encadenamiento hacia delante, del encadenamiento hacia atrás, y de los algoritmos de resolución que se introdujeron en el Capítulo 7. La ventaja clave de las reglas de inferencia elevadas sobre la proposicionalización es que sólo realizan aquellas sustituciones que se necesitan para permitir avanzar a las inferencias. Un tema potencialmente confuso es que uno percibe que el Modus Ponens Generalizado es menos general que el Modus Ponens (Apartado 7.5): el Modus Ponens reconoce cualquier α atómica que esté en el lado izquierdo de la implicación, mientras que el Modus Ponens Generalizado necesita un formato especial para esta sentencia. La regla es generalizada en el sentido de que permite trabajar con cualquier número de P_i' .

Unificación

UNIFICACIÓN

UNIFICADOR

Las reglas de inferencia elevadas necesitan encontrar las sustituciones que hacen que expresiones lógicas diferentes se hagan idénticas. Este proceso se denomina **unificación** y es el componente clave de todos los algoritmos de inferencia en lógica de primer orden. El algoritmo UNIFICA toma dos sentencias y devuelve un **unificador** para ellas, si éste existe:

$$\text{UNIFICA}(p, q) = \theta \text{ donde } \text{SUST}(\theta, p) = \text{SUST}(\theta, q).$$

Vamos a ver algunos ejemplos de cómo se comportaría UNIFICA. Suponga que tenemos una petición *Conoce(Juan, x)*: ¿a quién conoce Juan? Algunas respuestas a esta pregunta se pueden hallar encontrando todas las sentencias de la base de conocimiento que se unifiquen con *Conoce(Juan, x)*. Aquí tenemos los resultados de la unificación con cuatro sentencias distintas que podrían estar en la base de conocimiento.

$\text{UNIFICA}(\text{Conoce}(Juan, x), \text{Conoce}(Juan, Juana)) = \{x/Juana\}$
 $\text{UNIFICA}(\text{Conoce}(Juan, x), \text{Conoce}(y, Guillermo)) = \{Guillermo, y/Juan\}$
 $\text{UNIFICA}(\text{Conoce}(Juan, x), \text{Conoce}(y, \text{Madre}(y))) = \{y/Juan, x/\text{Madre}(Juan)\}$
 $\text{UNIFICA}(\text{Conoce}(Juan, x), \text{Conoce}(x, Elisabet)) = \text{fallo}.$

La última unificación falla porque x no puede tomar los valores *Juan* y *Elisabet* al mismo tiempo. Ahora, recuerda que $\text{Conoce}(x, Elisabet)$ significa «Todo el mundo conoce a *Elisabet*», por tanto, seríamos capaces de inferir que *Juan* conoce a *Elisabet*. El problema se presenta sólo cuando las dos sentencias tienen que utilizar el mismo nombre de variable, x . Este problema se puede evitar utilizando la **estandarización de las variables** de las sentencias que van a ser unificadas, que consiste en renombrar sus variables para evitar conflictos de nombre. Por ejemplo, podemos renombrar la x de $\text{Conoce}(x, Elisabet)$ por z_{17} (un nuevo nombre de variable) sin tener que cambiar su significado. Ahora la unificación tendrá éxito:

$\text{UNIFICA}(\text{Conoce}(Juan, x), \text{Conoce}(z_{17}, Elisabet)) = \{x/Elisabet, z_{17}/Juan\}$

El Ejercicio 9.7 ahonda más en la necesidad de la estandarización de variables.

Hay una complicación más: decimos que UNIFICA debería devolver una sustitución que hace que dos argumentos sean idénticos. Pero podría haber más de un unificador. Por ejemplo, $\text{UNIFICA}(\text{Conoce}(Juan, x), \text{Conoce}(y, z))$ podría devolver $\{y/Juan, x/z\}$ o $\{y/Juan, x/Juan, z/Juan\}$. El primer unificador nos da $\text{Conoce}(Juan, z)$ como resultado de la unificación, mientras que el segundo nos da $\text{Conoce}(Juan, Juan)$; decimos que el primer unificador es más *general* que el segundo, porque coloca menos restricciones sobre los valores de las variables. Resulta que, para cada par de expresiones unificable hay un **unificador más general** (o UMG) que es único respecto al renombramiento de las variables. En este caso $\{y/Juan, x/z\}$.

En la Figura 9.1 se muestra un algoritmo para obtener el unificador más general. El proceso es muy sencillo: el algoritmo explora recursivamente las dos expresiones «de lado a lado», acumulando un unificador durante el proceso, pero falla si dos posiciones de las dos estructuras que se corresponden no emparejan. Hay un paso en el proceso que es costoso: cuando al intentar emparejar una variable con un término complejo, se debe comprobar si la propia variable ya pertenece al término; y si es así, el emparejamiento falla porque no se puede generar un unificador consistente. Este proceso, denominado **comprobación de ocurrencias**, hace que la complejidad del algoritmo sea cuadrática respecto al tamaño de las expresiones que se van a unificar. Algunos sistemas, incluidos todos los sistemas de programación lógica, simplemente omiten este proceso por lo que algunas veces obtienen como resultado inferencias inconsistentes; otros sistemas utilizan algoritmos más complejos que presentan una complejidad temporal de tipo lineal.

ESTANDARIZACIÓN DE VARIABLES

UNIFICADOR MÁS GENERAL

COMPROBACIÓN DE OCURRENCIAS

Almacenamiento y recuperación

Por debajo de las funciones DECIR y PREGUNTAR, utilizadas para informar o interrogar a la base de conocimiento, están las funciones primitivas ALMACENAR y BUSCAR. ALMACENAR (s) guarda una sentencia s en la base de conocimiento, y BUSCAR(q) devuelve todos los unificadores que unifican la petición q con alguna sentencia de la base

función UNIFICA(x, y, θ) devuelve una sustitución que hace x e y idénticas
entradas: x , una variable, constante, lista, o expresión compuesta
 y , una variable, constante, lista, o expresión compuesta
 θ , la sustitución construida hasta ahora (opcional, por defecto está vacía)

```

si  $\theta = \text{fallo}$  entonces devolver fallo
sino si  $x = y$  entonces devolver  $\theta$ 
sino si  $\text{¿VARIABLE?}(x)$  entonces devolver UNIFICA-VAR( $x, y, \theta$ )
sino si  $\text{¿VARIABLE?}(y)$  entonces devolver UNIFICA-VAR( $y, x, \theta$ )
sino si  $\text{¿EXP-COMPUESTA?}(x) \text{ y } \text{¿EXP-COMPUESTA?}(y)$  entonces
    devolver UNIFICA(ARGS[ $x$ ], ARGS[ $y$ ], UNIFICA(OP[ $x$ ], OP[ $y$ ],  $\theta$ ))
sino si  $\text{¿LISTA?}(x) \text{ y } \text{¿LISTA?}(y)$  entonces
    devolver UNIFICA(REST[ $x$ ], REST[ $y$ ], UNIFICA(PRIMERO[ $x$ ], PRIMERO[ $y$ ],  $\theta$ ))
sino devolver fallo

```

Función UNIFICA-VAR(var, x, θ) devuelve una sustitución

Entradas: var , una variable
 x , una expresión
 θ , la sustitución construida hasta ahora

```

si  $\{var/val\} \in \theta$  entonces devolver UNIFICA( $val, x, \theta$ )
sino si  $\{x/val\} \in \theta$  entonces devolver UNIFICA( $var, val, \theta$ )
sino si  $\text{¿COMPRUEBA-OC?}(var, x)$  entonces devolver fallo
sino devolver añadir  $\{var/x\}$  a  $\theta$ 

```

Figura 9.1 El algoritmo de unificación. El algoritmo trabaja mediante la comparación, elemento a elemento, de las expresiones de entrada. La sustitución θ , que es el argumento de UNIFICA, se va construyendo a lo largo de todo el proceso y se utiliza para asegurar que las comparaciones posteriores sean consistentes con las ligaduras que previamente se han establecido. En una expresión compuesta, como $F(A, B)$, la función OP selecciona el símbolo de función F , y la función ARGS selecciona la lista de argumentos (A, B).

de conocimiento. El problema que hemos utilizado para ilustrar la unificación (encontrando todos los hechos que se unifican con *Conoce(Juan, x)*) es una instancia de BUSCANDO.

La forma más sencilla de implementar ALMACENAR y BUSCAR consiste en mantener todos los hechos en la base de conocimiento mediante una lista larga; entonces, dada una petición q , se llama a UNIFICA(q, s) con cada sentencia s de la lista. Este proceso es ineficiente, pero trabaja bien, y es todo lo que necesita entender para el resto del capítulo. Lo que queda de esta sección da una idea general de los mecanismos para realizar la recuperación de forma más eficiente, y por lo tanto, en una primera lectura se lo puede saltar.

Podemos BUSCAR de forma más eficiente asegurándonos de que las unificaciones sólo se intenten con las sentencias que tienen *alguna* oportunidad de unificarse. Por ejemplo, no hay ninguna posición que pueda unificar *Conoce(Juan, x)* y *Hermano(Ricardo, Juan)*. Podemos evitar este tipo de unificaciones **indexando** los hechos en la base de conocimiento. Un sencillo programa, denominado **indexación de predicados**, coloca to-

dos los hechos *Conoce* en un cajón y todos los hechos *Hermano* en otro. Los cajones se pueden almacenar en una tabla *hash*² para obtener un acceso más eficiente.

La indexación de predicados es útil cuando hay muchos símbolos de predicado pero sólo unas pocas cláusulas por cada símbolo. En algunas aplicaciones, hay muchas cláusulas para un símbolo de predicado dado. Por ejemplo, suponga que las entidades de recaudación de impuestos quieren guardar la pista de quién contrata a quién, utilizando el predicado *Contrata(x, y)*. Esto generaría un cajón muy grande con quizás millones de contratantes y decenas de millones de contratados. Y por tanto, responder a una petición como *Contrata(x, Ricardo)* mediante la indexación de predicados requeriría recorrer el cajón entero.

Para esta petición en concreto, sería de ayuda que los hechos estuvieran indexados tanto por el predicado como por su segundo argumento, quizás utilizando una tabla *hash* de claves combinadas. Entonces podríamos simplemente construir la clave para la petición y recuperar exactamente aquellos hechos que se unifican con la petición. Para otras peticiones, tales como *Contrata(AIMA.org, y)*, necesitaríamos haber indexado los hechos combinando el predicado con el primer argumento. Por lo tanto, los hechos se pueden almacenar bajo múltiples claves, haciéndolos accesibles instantáneamente a las diferentes peticiones con las que se podrían unificar.

Dada una sentencia a almacenar, es posible construir los índices para *todas las posibles* peticiones que se unifican con ella. Para el hecho *Contrata(AIMA.org, Ricardo)*, las peticiones son

<i>Contrata(AIMA.org, Ricardo)</i>	¿Contrata AIMA a Ricardo?
<i>Contrata(x, Ricardo)</i>	¿Quién contrata a Ricardo?
<i>Contrata(AIMA.org, y)</i>	¿A quién contrata AIMA.org?
<i>Contrata(x, y)</i>	¿Quién contrata a quién?

RETÍCULO DE SUBSUNCIÓN

Estas peticiones forman un **retículo de subsunción**, tal como se muestra en la Figura 9.2(a). El retículo tiene algunas propiedades interesantes. Por ejemplo, el hijo de cualquier nodo del retículo se obtiene de su padre mediante una subsunción única; y el descendiente común «más alto» de dos nodos cualesquiera es el resultado de aplicar su unificador más general. La porción del retículo por encima de cualquier hecho base se puede construir sistemáticamente (Ejercicio 9.5). Una sentencia con constantes repetidas tiene un retículo algo diferente, tal como se muestra en la Figura 9.2(b). Los símbolos de función y las variables en las sentencias a que pueden ser almacenadas aún generan estructuras de retículos más interesantes.

El programa que hemos descrito trabaja muy bien siempre que el retículo contenga un número pequeño de nodos. Para un predicado con n argumentos, el retículo contiene $O(2^n)$ nodos. Si se permiten los símbolos de función el número de nodos entonces es exponencial respecto al número de términos de la sentencia a almacenar. Esto nos puede conducir a un número inmenso de índices. En algún punto, los beneficios de la indexación son superados por los costes de almacenar y mantener todos los índices. Podemos responder adoptando una política fija, tal como mantener sólo los índices de las claves compuestas por un predicado

² Una tabla *hash* es una estructura de datos para almacenar y recuperar información indexada mediante claves fijas. Por motivos prácticos, se puede considerar que una tabla *hash* tiene tiempos constantes de almacenamiento y recuperación, aun cuando la tabla contenga un número enorme de elementos.

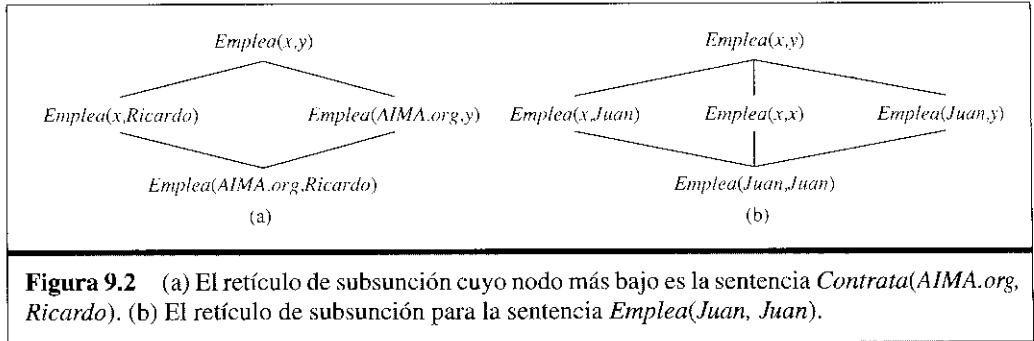


Figura 9.2 (a) El retículo de subsunción cuyo nodo más bajo es la sentencia *Contrata(AIMA.org, Ricardo)*. (b) El retículo de subsunción para la sentencia *Emplea(Juan, Juan)*.

y cada uno de sus argumentos, o utilizando una política adaptativa que genere índices que respondan a las demandas de los tipos de peticiones que se desean realizar. En muchos sistemas de IA, el número de los hechos que se almacenan es lo suficientemente pequeño de manera que la indexación eficiente se considera un problema resuelto. En las bases de datos industriales y comerciales, el problema ha recibido un desarrollo tecnológico sustancial.

9.3 Encadenamiento hacia delante

En el Apartado 7.5 se dio un algoritmo de encadenamiento hacia delante para cláusulas positivas proposicionales. La idea es simple: comenzar a partir de las sentencias atómicas de la base de conocimiento y aplicar el Modus Ponens hacia delante, añadiendo las sentencias atómicas nuevas hasta que no se puedan realizar más inferencias. Aquí explicamos cómo se puede aplicar el algoritmo a las cláusulas positivas de primer orden, y cómo se puede implementar eficientemente. Las cláusulas positivas como *Situación \Rightarrow Respuesta* son especialmente útiles para los sistemas que realizan inferencias en respuesta a información nueva que se ha recibido. Muchos sistemas se pueden definir de esta manera, y el razonamiento hacia delante puede ser mucho más eficiente que la resolución aplicada a la demostración de problemas. Por lo tanto, a menudo vale la pena intentar construir una base de conocimiento tan sólo con cláusulas positivas de tal manera que se evite el coste de aplicar la resolución.

Cláusulas positivas de primer orden

Las cláusulas positivas de primer orden se parecen bastante a las cláusulas positivas proposicionales (Apartado 7.5): éstas son disyunciones de literales de los cuales *sólo uno es positivo*. Una cláusula positiva es atómica o es una implicación cuyo antecedente es una conjunción de literales positivos y cuyo consecuente es un único literal positivo. Las siguientes son cláusulas positivas de primer orden:

$$\begin{aligned} & Rey(x) \wedge Codicioso(x) \Rightarrow Malvado(x) \\ & Rey(Juan) \\ & Codicioso(y) \end{aligned}$$

Diferentes a los literales proposicionales, los literales de primer orden pueden contener variables, en cuyo caso se asume que dichas variables están cuantificadas universalmente. (Lo típico es que omitamos los cuantificadores universales cuando escribamos cláusulas positivas.) Las cláusulas positivas son una forma normal muy adecuada para utilizarla con el Modus Ponens Generalizado.

No se pueden convertir todas las bases de conocimiento a un conjunto de cláusulas positivas, por la restricción de un único literal positivo, pero muchas sí se pueden transformar. Considere el siguiente problema:

La ley dice que es un crimen para un americano vender armas a países hostiles. El país de Nono, un enemigo de América, tiene algunos misiles, y todos sus misiles fueron vendidos por el Coronel West, que es americano.

Demostraremos que West es un criminal. Primero, representaremos estos hechos mediante cláusulas positivas de primer orden. La siguiente sección mostrará cómo el encadenamiento hacia delante resuelve este problema.

«... es un crimen para un americano vender armas a países hostiles»:

$$\text{Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Hostil}(z) \Rightarrow \text{Criminal}(x) \quad (9.3)$$

«Nono ...tiene algunos misiles.» La sentencia $\exists x \text{Tiene}(\text{Nono}, x) \wedge \text{Misil}(x)$ se transforma en dos cláusulas positivas mediante la Eliminación del Existencial, introduciendo la nueva constante M_1 :

$$\text{Tiene}(\text{Nono}, M_1) \quad (9.4)$$

$$\text{Misil}(M_1) \quad (9.5)$$

«Todos los misiles le (a Nono) fueron vendidos por el coronel West»:

$$\text{Misil}(x) \wedge \text{Tiene}(\text{Nono}, x) \Rightarrow \text{Vende}(\text{West}, x, \text{Nono}) \quad (9.6)$$

También necesitamos saber que los misiles son armas:

$$\text{Misil}(x) \Rightarrow \text{Arma}(x) \quad (9.7)$$

Y necesitamos saber que un enemigo de América es un país «hostil»:

$$\text{Enemigo}(x, \text{América}) \Rightarrow \text{Hostil}(x) \quad (9.8)$$

«West, que es Americano...»:

$$\text{Americano}(\text{West}) \quad (9.9)$$

«El país Nono, un enemigo de América...»:

$$\text{Enemigo}(\text{Nono}, \text{América}) \quad (9.10)$$

Esta base de conocimiento no contiene símbolos de función y es por lo tanto una instancia de la clase de bases de conocimiento de **Datalog**, es decir, conjuntos de cláusulas positivas de primer orden sin símbolos de función. Veremos que la ausencia de los símbolos de función hace mucho más fácil la inferencia.

Un algoritmo sencillo de encadenamiento hacia delante

El primer algoritmo de encadenamiento hacia delante que consideraremos es uno muy sencillo, tal como se muestra en la Figura 9.3. Comenzando con los hechos conocidos, el proceso dispara todas las reglas cuyas premisas se satisfacen, añadiendo sus conclusiones al conjunto de hechos conocidos. El proceso se va repitiendo hasta que la petición es respondida (asumiendo que sólo se requiere una respuesta) o no se pueden añadir más hechos. Fíjese en que un hecho no es «nuevo», sólo es el **renombramiento** de un hecho conocido. Una sentencia es el renombramiento de otra si son idénticas excepto en los nombres de las variables. Por ejemplo, *Gusta(x, Helado)* y *Gusta(y, Helado)* son renombramientos cada una de la otra porque sólo se diferencian en la elección de *x* o de *y*; sus significados son el mismo: a todo el mundo le gusta el helado.

Utilizaremos nuestro problema sobre el crimen para ilustrar cómo funciona PREGUNTA-EHD-LPO. Las sentencias de implicación son (9.3), (9.6), (9.7) y (9.8). Se necesitan dos iteraciones:

- En la primera iteración, la regla (9.3) no tiene las premisas satisfechas. La regla (9.6) se satisface con $\{x/M_1\}$, y se añade *Vende(West, M₁, Nono)*. La regla (9.7) se satisface con $\{x/M_1\}$, y se añade *Arma(M₁)*. La regla (9.8) se satisface con $\{x/Nono\}$, y se añade *Hostil(Nono)*.
- En la segunda iteración, la regla (9.3) se satisface con $\{x/West, y/M_1, z/Nono\}$, y se añade *Criminal(West)*.

función PREGUNTA-EHD-LPO(*BC, α*) **devuelve** una sustitución o *falso*

entradas: *BC*, la base de conocimiento, un conjunto de cláusulas positivas de primer orden *α*, la petición, una sentencia atómica

variables locales: *nuevas*, las nuevas sentencias inferidas en cada iteración

repetir hasta *nuevo* está vacío

nuevo $\leftarrow \{\}$

para cada sentencia *r* **en** *BC* **hacer**

$(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{ESTANDARIZAR-VAR}(r)$

para cada *θ* tal que *SUST*(*θ, p₁ ∧ … ∧ p_n*) = *SUST*(*p'₁ ∧ … ∧ p'_n*)

 para algún *p'₁ … p'_n* en *BC*

$q' \leftarrow \text{SUST}(\theta, q)$

si *q'* no es el renombramiento de una sentencia de *BC* o *nuevo* **entonces hacer**

 añadir *q'* a *nuevo*

$\phi \leftarrow \text{UNIFICA}(q', \alpha)$

si *ϕ* no es *falso* **entonces devolver** *ϕ*

 añadir *nuevo* a *BC*

devolver *falso*

Figura 9.3 Un algoritmo de encadenamiento hacia delante, conceptualmente sencillo, pero muy inefficiente. En cada iteración añade a la *BC* todas las sentencias atómicas que se pueden inferir en un paso, a partir de las sentencias de implicación y las sentencias atómicas ya presentes en la *BC*.

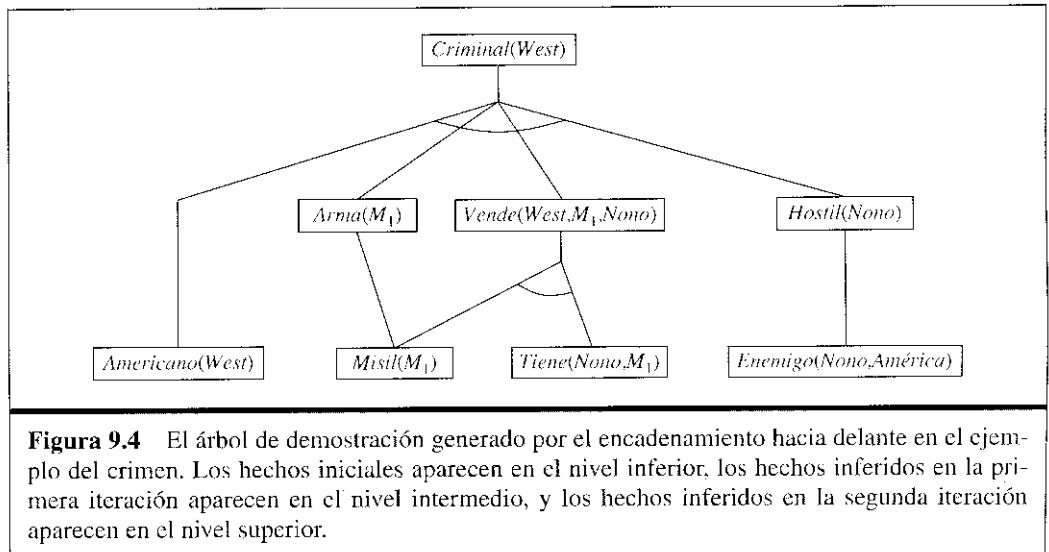


Figura 9.4 El árbol de demostración generado por el encadenamiento hacia delante en el ejemplo del crimen. Los hechos iniciales aparecen en el nivel inferior, los hechos inferidos en la primera iteración aparecen en el nivel intermedio, y los hechos inferidos en la segunda iteración aparecen en el nivel superior.

La Figura 9.4 muestra el árbol de demostración que se ha generado. Fíjese en que las nuevas inferencias son posibles en este punto porque cada sentencia que podía concluirse mediante el encadenamiento hacia delante ya está contenida explícitamente en la BC. Este tipo de base de conocimiento se dice que tiene un proceso de inferencia de **punto fijo**. Los puntos fijos hallados con las cláusulas positivas de primer orden son similares a aquellas que se hallan en el encadenamiento hacia delante proposicional (Apartado 7.5); la principal diferencia es que el punto fijo de primer orden puede incluir sentencias atómicas cuantificadas universalmente.

Es fácil analizar el algoritmo PREGUNTA-EHD-LPO. Primero, es un algoritmo **sólido**, porque cada inferencia es justo una aplicación del Modus Ponens Generalizado, que es sólida. Segundo, es **completo** para las bases de conocimiento con cláusulas positivas; es decir, responde cada petición cuyas respuestas se implican de cada base de conocimiento con cláusulas positivas. En las bases de conocimiento Datalog, que no contienen símbolos de función, la demostración de su completitud es bastante fácil. Comenzamos contando el número de los posibles hechos que pueden ser añadidos, lo que determina el número máximo de iteraciones. Siendo k la **aridad** máxima (número de argumentos) de cada predicado, p el número de predicados, y n el número de símbolos de constante. Está claro que no puede haber más de pn^k hechos base, así que después de estas muchas iteraciones el algoritmo debe encontrar un punto fijo. Entonces podemos construir un argumento muy similar a la demostración de la completitud en el encadenamiento hacia delante proposicional (Apartado 7.5). Los detalles acerca de la transición de la completitud proposicional a la de primer orden se dan en el algoritmo de resolución del Apartado 9.5.

Con las cláusulas positivas que contienen símbolos de función, PREGUNTA-EHD-LPO puede generar muchos hechos nuevos infinitamente, por lo que necesitamos tener mucho cuidado. Para el caso en el que una respuesta se implica para la sentencia de petición q , debemos recurrir al teorema de Herbrand para establecer que el algoritmo

encontrará una demostración. (Véase el Apartado 9.5 para el caso de la resolución.) Si la petición no tiene respuesta, el algoritmo podría fallar y terminar en algunos casos. Por ejemplo, si la base de conocimiento incluye los axiomas de Peano

$$\begin{aligned} & \text{NumNat}(0) \\ & \forall n \text{ NumNat}(n) \Rightarrow \text{NumNat}(S(n)) \end{aligned}$$

entonces el encadenamiento hacia delante añade $\text{NumNat}(S(0))$, $\text{NumNat}(S(S(0)))$, $\text{NumNat}(S(S(S(0))))$, etc. Este problema en general es inevitable. Igual que con la lógica de primer orden general, la implicación con cláusulas positivas es semidecidible.

Encadenamiento hacia delante eficiente

El algoritmo de encadenamiento hacia delante de la Figura 9.3 está diseñado más bien para facilitar su comprensión que para que sea eficiente en su ejecución. Hay tres fuentes de complejidad posibles. Primero, el «bucle interno» del algoritmo requiere que se encuentren todos los unificadores posibles de manera que la premisa se unifique con un conjunto adecuado de hechos de la base de conocimiento. A este proceso, a menudo se le denomina **emparejamiento de patrones** y puede ser muy costoso. Segundo, el algoritmo vuelve a comprobar cada regla en cada iteración para ver si sus premisas se satisfacen, incluso cuando se han realizado muy pocas adiciones a la base de conocimiento en cada iteración. Por último, el algoritmo podría generar muchos hechos que son irrelevantes para el objetivo. Vamos a abordar cada uno de estos problemas por separado.

EMPAРЕJAMIENTO DE PATRONES

Emparejar reglas con los hechos conocidos

El problema de emparejar la premisa de una regla con los hechos de la base de conocimiento podría parecer algo bastante sencillo. Por ejemplo, suponga que queremos aplicar la regla

$$\text{Misil}(x) \Rightarrow \text{Arma}(x)$$

Entonces necesitamos encontrar todos los hechos que se unifican con $\text{Misil}(x)$; en una base de conocimiento indexada de forma adecuada, esto se puede realizar en tiempo constante por cada hecho. Ahora considere una regla como

$$\text{Misil}(x) \wedge \text{Tiene}(Nono, x) \Rightarrow \text{Vende}(West, x, Nono).$$

Otra vez, podemos encontrar todos los objetos que tiene Nono en un tiempo constante por objeto; entonces, por cada objeto, podríamos comprobar si es un misil. Sin embargo, si la base de conocimiento tiene muchos objetos en posesión de Nono y unos pocos misiles, sería mejor encontrar todos los misiles primero y entonces comprobar si son de Nono. Este es el problema de la **ordenación de los conjuntos**: encontrar una ordenación para resolver los conjuntos de las premisas de una regla de tal manera que el coste se minimice. Resulta que encontrar la ordenación óptima es un problema NP-duro, pero hay buenas heurísticas disponibles. Por ejemplo, la heurística de la **variable más**

ORDENACIÓN DE CONJUNTOS

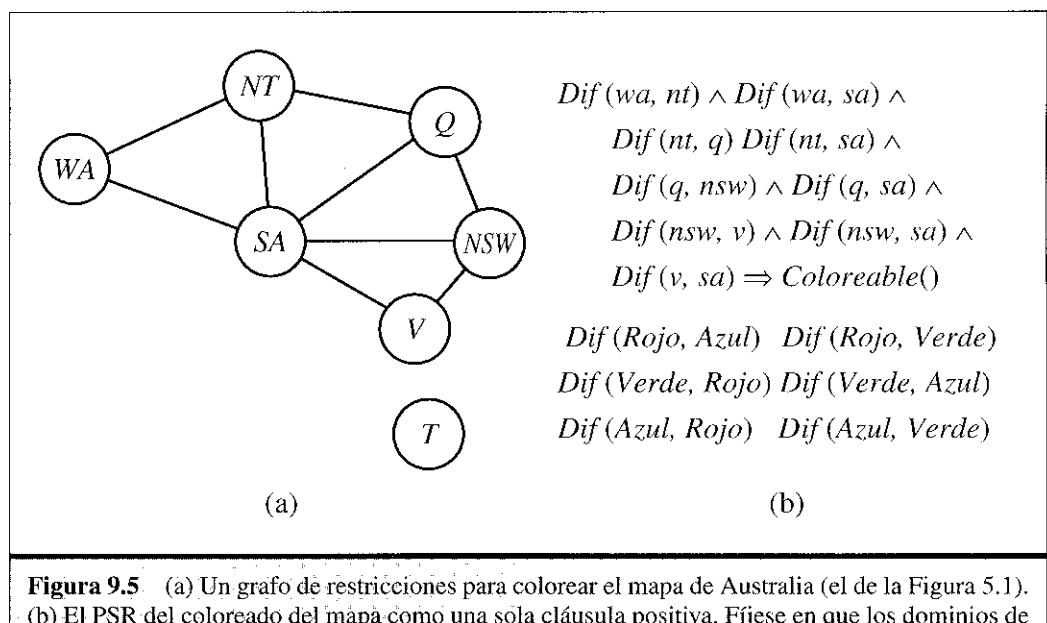
restringida que se utiliza en los PSR del Capítulo 5 sugeriría ordenar los conjuntos para ver los misiles antes si hay menos misiles que objetos que pertenezcan a Nono.

La conexión entre el emparejamiento de patrones y la satisfacción de restricciones realmente es muy estrecha. Podemos ver cada conjuntor como una restricción sobre las variables que contiene, por ejemplo, $Misil(x)$ es una restricción unaria sobre la variable x . Ampliando esta idea, *podemos expresar cada PSR de dominio finito como una única cláusula positiva junto a algunos hechos base asociados a ella*. Considere el problema del coloreado de un mapa de la Figura 5.1 que se muestra en la Figura 9.5(a). Una formulación equivalente mediante una cláusula positiva se muestra en la Figura 9.5(b). Está claro que la conclusión $Coloreable()$ se puede inferir sólo si el PSR tiene una solución. Ya que los PSR en general incluyen a los problemas 3SAT como un caso especial, podemos concluir que *emparejar una cláusula positiva con un conjunto de hechos es un problema NP-duro*.

Podría parecer algo depresivo que el encadenamiento hacia delante tenga un problema de emparejamiento NP-duro en su bucle interno. Sin embargo, hay tres motivos por los cuales animarse:

- Podemos recordarnos a nosotros mismos que muchas reglas en bases de conocimiento del mundo real son más bien pequeñas y sencillas (como las reglas de nuestro problema del crimen) que grandes y complejas (como las de la formulación del PSR de la Figura 9.5). Es algo común asumir, en las bases de datos sobre el mundo, que tanto el tamaño de las reglas como las aridades de los predicados están limitados por una constante y preocuparnos sólo por la **complejidad de los datos**, es decir, la complejidad de la inferencia está en función del número de hechos base en la base de datos. Es fácil demostrar que la complejidad de los datos del encadenamiento hacia delante es polinómica.

COMPLEJIDAD DE DATOS



- Podemos considerar subclases de reglas para las cuales el emparejamiento sea eficiente. Esencialmente, cada cláusula Datalog se puede ver como la definición de un PSR, así que el emparejamiento será tratable sólo cuando el correspondiente PSR lo sea. El Capítulo 5 describe varias familias tratables de PSR. Por ejemplo, si el grafo de restricciones (el grafo cuyos nodos son las variables y cuyos arcos son las restricciones) forma un árbol, entonces el PSR se puede resolver en tiempo lineal. El mismo resultado se obtiene exactamente con el emparejamiento de reglas. Por ejemplo, si eliminamos Sur de Australia del mapa de la Figura 9.5, la cláusula resultante será

$$Dif(wa, nt) \wedge Dif(nt, q) \wedge Dif(q, nsw) \wedge Dif(nsw, v) \Rightarrow Coloreable()$$

Que se corresponde con el PSR reducido de la Figura 5.11. Los algoritmos para resolver los PSR con estructura de árbol se pueden aplicar de forma directa al problema de emparejamiento de reglas.

- Podemos trabajar duro para eliminar los intentos de emparejamiento de reglas redundantes en el encadenamiento hacia delante, que es el tema de la siguiente sección.

Encadenamiento hacia delante incremental

Cuando mostramos cómo trabaja el encadenamiento hacia delante mediante el ejemplo del crimen hicimos trampa; en concreto, omitimos alguno de los emparejamientos de reglas que se muestran en el algoritmo de la Figura 9.3. Por ejemplo, en la segunda iteración, la regla

$$Misil(x) \Rightarrow Arma(x)$$

 se empareja con $Misil(M_1)$ (otra vez), y desde luego, la conclusión $Arma(M_1)$ ya se conoce y por tanto no pasa nada. Este tipo de emparejamientos de reglas redundantes se puede evitar si hacemos la siguiente observación: *cada hecho nuevo inferido en una iteración t debe ser derivado de al menos un hecho nuevo inferido en la iteración t - 1*. Esto es cierto porque cualquier inferencia que no necesite un hecho nuevo de la iteración $t - 1$ podía haberse realizado ya en la iteración $t - 1$.

Esta observación nos lleva de forma natural a un algoritmo de encadenamiento hacia delante incremental, en el que en cada iteración t comprobamos una regla sólo si su premisa incluye algún conjuntor p_i que se unifique con un hecho p'_i que se haya inferido en la iteración $t - 1$. El paso de emparejamiento de regla fija que p_i se empareje con p'_i , pero permite que los otros conjuntores de la regla se emparejen con hechos de cualquier iteración previa. Este algoritmo genera exactamente los mismos hechos en cada iteración que en el algoritmo de la Figura 9.3, pero mucho más eficientemente.

Con una indexación adecuada, es fácil identificar todas las reglas que se pueden disparar por un hecho dado, y en efecto, muchos sistemas reales operan en un modo de «actualización» en donde el encadenamiento hacia delante ocurre en respuesta a cada nuevo hecho que es DICHO al sistema. Las inferencias se generan en cascada a través del conjunto de reglas hasta que se encuentra un punto fijo y entonces, el proceso comienza otra vez con el siguiente hecho nuevo.

Generalmente, sólo una pequeña fracción de las reglas de la base de conocimiento es realmente disparada por la adición de un hecho nuevo. Esto significa que se realiza una gran cantidad de trabajo redundante en construir emparejamientos parciales repetidamente que tienen algunas de sus premisas insatisfechas. Nuestro ejemplo del crimen es más bien demasiado pequeño como para mostrar esto de forma efectiva, pero fíjese en que se construye un emparejamiento parcial en la primera iteración entre la regla

$$\text{Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Hostil}(z) \Rightarrow \text{Criminal}(x)$$

y el hecho *Americano(West)*. Este emparejamiento parcial entonces se descarta y se reconstruye en la segunda iteración (cuando la regla tiene éxito). Sería mejor retener y completar gradualmente el emparejamiento parcial a medida que llegan nuevos hechos, que descartarlo.

RETE

El algoritmo **rete**³ fue el primero en aplicarse seriamente a este problema. El algoritmo preprocesa el conjunto de reglas de la base de conocimiento para construir una especie de red de flujo de datos en la que cada nodo es un literal de las premisas de la regla. Las ligaduras de las variables fluyen a través de la red y se eliminan cuando fallan al emparejar un literal. Si dos literales en una regla comparten una variable (por ejemplo, *Vende(x, y, z) \wedge Hostil(z)* en el ejemplo del crimen) entonces las ligaduras de cada literal se filtran mediante un nodo de igualdad. Una ligadura de variables que se encuentra con un nodo con un literal *n*-ario, como *Vende(x, y, z)* podría tener que esperar a que se establezcan las ligaduras de las otras variables para que el proceso continúe. En un punto dado, el estado de una red rete captura todos los emparejamientos parciales de las reglas, evitando una gran cantidad de cálculo.

SISTEMAS DE PRODUCCIÓN

Las redes rete, y varias mejoras basadas en ellas, han sido un componente clave de los denominados **sistemas de producción**, que fueron de entre los sistemas de encadenamiento hacia delante los de uso más generalizado⁴. El sistema XCON (originalmente llamado R1, McDermott, 1982) se construyó utilizando una arquitectura de sistema de producción. El XCON contenía varios miles de reglas para el diseño de configuraciones de componentes de computador para los clientes de Digital Equipment Corporation. Fue uno de los primeros éxitos comerciales claros en el campo emergente de los sistemas expertos. Muchos otros sistemas similares han sido construidos utilizando la misma tecnología base, que ha sido implementada en el lenguaje de propósito general OPS-5.

ARQUITECTURAS COGNITIVAS

Los sistemas de producción también son populares en las **arquitecturas cognitivas**, es decir, los modelos sobre el razonamiento humano como el ACT (Anderson, 1983) y el SOAR (Laird *et al.*, 1987). En estos sistemas, la «memoria de trabajo» del sistema modela la memoria a corto plazo, y las reglas forman parte de la memoria a largo plazo. En cada ciclo de ejecución, las producciones se emparejan con los hechos en la memoria de trabajo. Una producción cuyas condiciones se satisfacen puede añadir o eliminar hechos en la memoria de trabajo. En contraste con la típica situación en una base de datos, a menudo, los sistemas de producción tienen muchas reglas y relativamente pocos hechos. Con la tecnología de emparejamiento adecuadamente optimizada, algunos sistemas modernos pueden operar sobre un millón de reglas en tiempo real.

³ Rete es red en Latín. En inglés, su pronunciación rima con *treaty*.

⁴ La palabra **producción** de los **sistemas de producción** denota una regla de condición-acción.

Hechos irrelevantes

La última fuente de ineficiencia en el encadenamiento hacia delante parece ser intrínseco en el enfoque y también surge en la lógica proposicional. (Véase Apartado 7.5.) El encadenamiento hacia delante realiza todas las inferencias permitidas basadas en los hechos conocidos, *aunque estos sean irrelevantes respecto al objetivo*. En nuestro ejemplo del crimen, no había reglas capaces de trazar conclusiones irrelevantes, así que la falta de dirección no era un problema. En otros casos (por ejemplo, si tenemos varias reglas que describan los hábitos alimenticios de los americanos y los precios de los misiles), PREGUNTA-EHD-LPO generará muchas conclusiones irrelevantes.

Una forma de evitar trazar conclusiones irrelevantes es utilizar el encadenamiento hacia atrás, tal como describiremos en el Apartado 9.4. Otra solución es restringir el encadenamiento hacia delante a un subconjunto seleccionado de las reglas; este enfoque se discutió en el contexto proposicional. Un tercer enfoque ha surgido de la comunidad de las bases de datos deductivas, en las que el encadenamiento hacia delante es la herramienta estándar. La idea es rescribir el conjunto de reglas, utilizando información del objetivo, de tal manera que sólo se consideran las ligaduras de variables relevantes (aquellas que pertenecen al denominado **conjunto mágico**) durante la inferencia hacia delante. Por ejemplo, si el objetivo es *Criminal(West)*, la regla que concluye *Criminal(x)* será rescrita para incluir un conjuntor extra que restringe el valor de la *x*:

$$\text{Mágico}(x) \wedge \text{Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Hostil}(z) \Rightarrow \text{Criminal}(x)$$

El hecho *Mágico(West)* también se añade a la BC. De esta manera, incluso si la base de conocimiento contiene hechos acerca de millones de americanos, sólo se considerará el Coronel West durante el proceso de inferencia hacia delante. El proceso completo para definir los conjuntos mágicos y rescribir la base de conocimiento es demasiado complejo para incluirlo aquí, pero la idea básica es realizar una especie de inferencia hacia atrás «genérica» desde el objetivo para resolver qué ligaduras de las variables necesitan ser restringidas. Por lo tanto, se puede pensar en el enfoque del conjunto mágico como en un tipo híbrido entre la inferencia hacia delante y el preprocesamiento hacia atrás.

CONJUNTO MÁGICO

9.4 Encadenamiento hacia atrás

La segunda gran familia de algoritmos de inferencia lógica utiliza el enfoque de **encadenamiento hacia atrás** que se introdujo en el Apartado 7.5. Estos algoritmos trabajan hacia atrás desde el objetivo, encadenando a través de las reglas hasta encontrar los hechos conocidos que soportan la demostración. Describiremos el algoritmo básico, y entonces describiremos cómo se utiliza en la **programación lógica**, que es la forma más ampliamente utilizada de razonamiento automático. También veremos que el encadenamiento hacia atrás presenta algunas desventajas comparado con el encadenamiento hacia delante, y veremos mecanismos para vencerlas. Por último, veremos la estrecha conexión entre la programación lógica y los problemas de satisfacción de restricciones.

Un algoritmo de encadenamiento hacia atrás

La Figura 9.6 muestra un algoritmo sencillo de encadenamiento hacia atrás, el PREGUNTA-EHA-LPO. El algoritmo se invoca con una lista de objetivos que contiene un solo elemento, la petición original, y devuelve el conjunto de todas las sustituciones que satisfacen la petición. La lista de objetivos se puede ver como una «pila» a la espera de ser procesada; si *todos* los objetivos se pueden satisfacer, entonces la rama actual de la demostración tiene éxito. El algoritmo toma el primer objetivo de la lista y encuentra cada cláusula de la base de conocimiento cuyo literal positivo, o **cabeza**, se unifica con él. Cada una de estas cláusulas crea una nueva llamada recursiva en la que las premisas, o **cuerpo**, de la cláusula se añaden a la pila de objetivos. Recuerda que los hechos son cláusulas con cabeza y sin cuerpo, así, cuando el objetivo se unifica con un hecho conocido, no se añaden más subobjetivos a la pila, y el objetivo se resuelve. La Figura 9.7 es el árbol de demostración para derivar *Criminal(West)* a partir de las sentencias (9.3) hasta la (9.10).

COMPOSICIÓN

El algoritmo utiliza la **composición** de sustituciones. COMPON(θ_1, θ_2) es la sustitución cuyo efecto es idéntico a aplicar cada sustitución en orden. Es decir,

$$\text{SUST}(\text{COMPON}(\theta_1, \theta_2), p) = \text{SUST}(\theta_2, \text{SUST}(\theta_1, p))$$

En el algoritmo, las ligaduras de variables actuales, que se almacenan en θ , están compuestas por las ligaduras que resultan de unificar el objetivo con la cabeza de la cláusula, y dan un conjunto nuevo de ligaduras para la siguiente llamada recursiva de la función.

El encadenamiento hacia atrás, tal como lo hemos escrito, es claramente un algoritmo de búsqueda de primero en profundidad. Esto significa que sus requerimientos de espacio son lineales respecto al tamaño de la demostración (olvidando por ahora, el espacio requerido para acumular la solución). También significa que el encadenamiento hacia atrás (a diferencia del encadenamiento hacia delante) sufre algunos problemas con

función PREGUNTA-EHA-LPO($BC, \text{objetivos}, \theta$) **devuelve** un conjunto de sustituciones
entradas: BC , una base de conocimiento

objetivos , una lista de conjuntos que forman la petición (θ ya aplicada)
 θ , la sustitución actual, inicialmente la sustitución vacía $\{\}$

variables locales: respuestas , un conjunto de sustituciones, inicialmente vacío

si objetivos está vacío **entonces devolver** $\{\theta\}$

$q' \leftarrow \text{SUST}(\theta, \text{PRIMERO}(\text{objetivos}))$

para cada sentencia r **en** BC **hacer** donde ESTANDARIZAR-VAR(r) = $(p_1 \wedge \dots \wedge p_n \Rightarrow q)$
 y $\theta' \leftarrow \text{UNIFICA}(q, q')$ tiene éxito

$\text{nuevos_objetivos} \leftarrow [p_1, \dots, p_n | \text{RESTO}(\text{objetivos})]$

$\text{respuestas} \leftarrow \text{PREGUNTA-EHA-LPO}(BC, \text{nuevos_objetivos}, \text{COMPON}(\theta', \theta)) \cup \text{respuestas}$

devolver respuestas

Figura 9.6 Un algoritmo sencillo de encadenamiento hacia atrás.

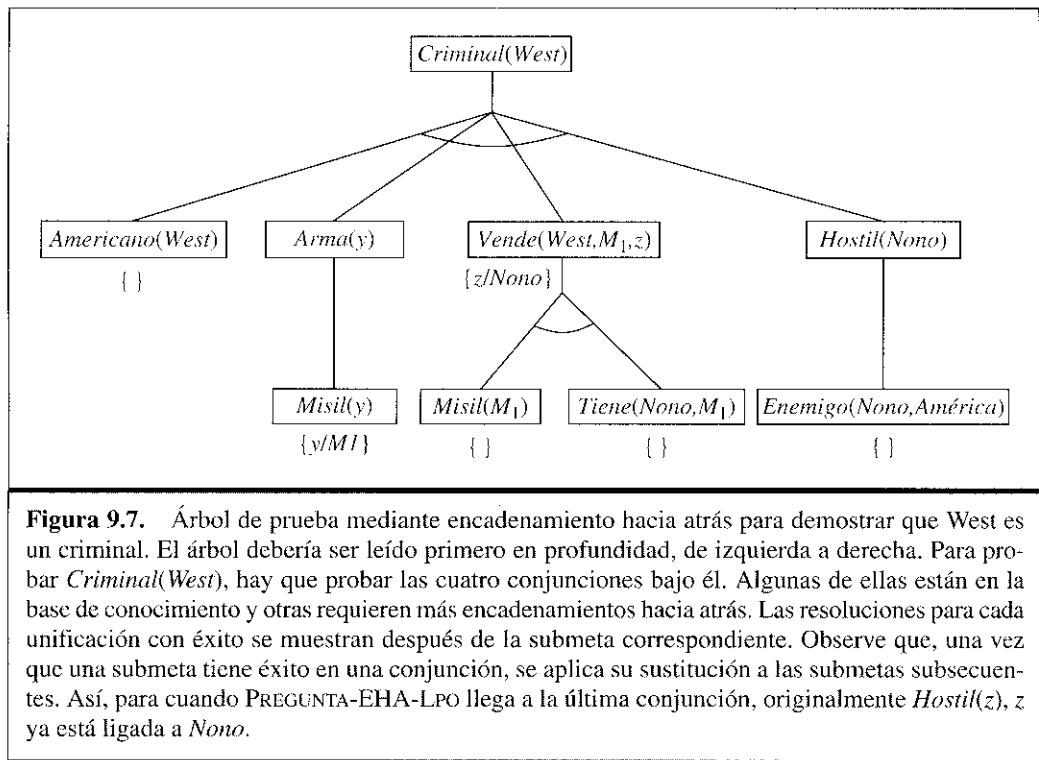


Figura 9.7. Árbol de prueba mediante encadenamiento hacia atrás para demostrar que West es un criminal. El árbol debería ser leído primero en profundidad, de izquierda a derecha. Para probar `Criminal(West)`, hay que probar las cuatro conjunciones bajo él. Algunas de ellas están en la base de conocimiento y otras requieren más encadenamientos hacia atrás. Las resoluciones para cada unificación con éxito se muestran después de la submeta correspondiente. Observe que, una vez que una submeta tiene éxito en una conjunción, se aplica su sustitución a las submetas subsecuentes. Así, para cuando PREGUNTA-EHA-LPO llega a la última conjunción, originalmente `Hostil(z)`, z ya está ligada a `Nono`.

los estados repetidos y la incompletitud. Discutiremos estos problemas y algunas soluciones potenciales, pero primero veremos cómo se utiliza el encadenamiento hacia atrás en los sistemas de programación lógica.

Programación lógica

La programación lógica es una tecnología que está bastante relacionada con abarcar el ideal declarativo descrito en el Capítulo 7: dichos sistemas se construirían expresando el conocimiento en un lenguaje formal y los problemas se resolverían ejecutando procesos de inferencia sobre dicho conocimiento. El ideal se resume en la ecuación de Robert Kowalski,

$$\text{Algoritmo} = \text{Lógica} + \text{Control}$$

PROLOG

Prolog es, de lejos, el lenguaje de programación lógica más extensamente utilizado. Sus usuarios se cuentan en cientos de miles. Se utilizó inicialmente como un lenguaje de prototipado rápido y para las tareas de manipulación de símbolos, como la escritura de compiladores (Van Roy, 1990), y en el análisis sintáctico del lenguaje natural (Pereira y Warren, 1980). Muchos sistemas expertos se han escrito en Prolog, para dominios legales, médicos, financieros y muchos otros.

Los programas en Prolog son conjuntos de cláusulas positivas escritas en una notación algo diferente a la estándar de la lógica de primer orden. Prolog utiliza las letras mayúsculas en las variables y las minúsculas en las constantes. Las cláusulas están escritas

con la cabeza precediendo al cuerpo; «`: -`» se utiliza para las implicaciones a la izquierda, las comas separan los literales en el cuerpo, y el punto indica el final de la sentencia:

```
criminal(X) :- americano(X), arma(Y), vende(X, Y, Z), hostil(Z)
```

El Prolog incluye una «sintaxis edulcorada» para la notación de listas y la aritmética. Como ejemplo, aquí tenemos un programa en Prolog para `unir(X, Y, Z)`, que tiene éxito si la lista `Z` es el resultado de unir las listas `X` y `Y`:

```
unir([], Y, Y)
unir([A|X], Y, [A|Z]) :- unir(X, Y, Z)
```

En castellano, podemos leer estas cláusulas como (1) `unir` una lista vacía a una lista `Y` genera la misma lista `Y`, y (2) `[A|Z]` es el resultado de unir `[A|X]` a `Y`, dado que `Z` es el resultado de unir `X` a `Y`. Esta definición de `unir` se asemeja bastante a la definición correspondiente en Lisp, pero realmente es mucho más potente. Por ejemplo, podemos realizar la petición `unir(A, B, [1, 2])`: ¿qué dos listas se pueden unir para dar `[1, 2]`? Obtenemos las soluciones hacia atrás

```
A = []      B = [1, 2]
A = [1]     B = [2]
A = [1, 2]  B = []
```

La ejecución de los programas en Prolog se hace mediante el encadenamiento hacia atrás en primero en profundidad, donde las cláusulas se van intentando en el orden en el que se han escrito en la base de conocimiento. Algunos aspectos del Prolog caen fuera de la inferencia lógica estándar:

- Hay un conjunto de funciones incorporadas para la aritmética. Los literales que utilizan estos símbolos de función se «demuestran» mediante la ejecución de código, y no realizando inferencias adicionales. Por ejemplo, el objetivo «`X` es `4+3`» tiene éxito con la `X` ligada a 7. Por el otro lado, el objetivo «`5` es `X+Y`» falla, ya que las funciones incorporadas no pueden resolver ecuaciones arbitrarias⁵.
- Hay predicados incorporados que tienen efectos colaterales cuando se ejecutan. Estos son predicados de entrada y salida, y predicados de `asertar/retractar` para modificar la base de conocimiento. Este tipo de predicados no tienen su homólogo en la lógica y pueden producir algunos efectos confusos, por ejemplo, si los hechos se asertan en una rama del árbol de demostración, éste finalmente falla.
- El Prolog permite un tipo de negación denominada **negación por fallo**. Un objetivo negado `no P` se considera demostrado si el sistema falla al probar `P`. Así, la sentencia

```
vivo(X) :- no muerto(X)
```

se puede leer como «Todo el mundo está vivo si no es probable que esté muerto».

- El Prolog tiene un operador de igualdad, `=`, pero le falta todo el poder de la igualdad lógica. Un objetivo de igualdad tiene éxito si los dos términos son *unificables* y falla de otro modo. Así, `X+Y=2+3` tiene éxito con la ligadura de `X` a 2 y de `Y` a 3, pero `estrellaMatinal=estrellaNocturna` falla. (En la lógica clásica, la

⁵ Fíjese en que si se proporcionaran los axiomas de Peano, este tipo de objetivos se podrían resolver mediante inferencia en un programa en Prolog.

última igualdad podría ser o no ser verdadera.) No se pueden asertar hechos o reglas acerca de la igualdad.

- La **comprobación de ocurrencias** se omite en el algoritmo de unificación del Prolog. Esto significa que se pueden realizar algunas inferencias no sólidas: éstas son rara vez un problema excepto cuando se utiliza el Prolog en la demostración de teoremas matemáticos.

Las decisiones tomadas en el diseño del Prolog representan un compromiso entre la eficiencia en la ejecución y el ideal declarativo; en tanto se entendía la eficiencia en el momento en que el Prolog se diseñó. Volveremos a este tema más tarde, al ver cómo se implementó el Prolog.

Implementación eficiente de programas lógicos

La ejecución de un programa en Prolog se puede realizar en dos modos: interpretado o compilado. La interpretación «esencialmente» acumula la ejecución del algoritmo PREGUNTA-EHA-LPO de la Figura 9.6, con el programa como la base de conocimiento. Decimos «esencialmente» porque los intérpretes de Prolog contienen una variedad de mejoras diseñadas para maximizar la velocidad. Aquí sólo consideramos dos.

Primero, en vez de construir la lista de todas las respuestas posibles para cada subobjetivo antes de continuar con el siguiente, los intérpretes de Prolog generan una respuesta y una «esperanza» de generar el resto cuando la respuesta actual haya sido totalmente explorada. Esta esperanza se denomina **punto de elección**. Cuando una búsqueda de primero en profundidad completa su exploración de las soluciones posibles que surgen de la respuesta actual y dan marcha atrás al punto de elección, el punto de elección se expande para tratar la nueva respuesta para el subobjetivo y el nuevo punto de elección. Este enfoque ahorra tiempo y espacio. También proporciona una interfaz muy sencilla para la depuración ya que cada vez sólo hay un único camino solución en consideración.

Segundo, nuestra sencilla implementación del PREGUNTA-EHA-LPO pierde una gran cantidad de tiempo en generar y componer las sustituciones. El Prolog implementa las

PUNTO DE ELECCIÓN

```

procedimiento UNIR(ax, y, az, continuación)
  pista ← PUNTERO-GLOBAL-PISTA()
  si ax = [] y UNIFICA(y, az) entonces LLAMA(continuación)
  REINICIALIZA-PISTA(pista)
  a ← VARIABLE-NUEVA()
  x ← VARIABLE-NUEVA()
  z ← VARIABLE-NUEVA()
  si UNIFICA(ax, [a - x]) y UNIFICA(az, [a - z]) entonces UNIR(x, y, z, continuación)

```

Figura 9.8 Pseudocódigo que representa el resultado de compilar el predicado Unir. La función VARIABLE-NUEVA devuelve una variable nueva, distinta de todas las otras variables utilizadas hasta el momento. El procedimiento LLAMA(*continuación*) continúa la ejecución con la continuación especificada.

sustituciones utilizando variables lógicas de las que se pueden recordar sus ligaduras actuales. En cualquier instante de tiempo, todas las variables o no están ligadas, o están ligadas a algún valor. Juntas, estas variables y valores definen implícitamente la sustitución para la rama actual de la demostración. Extender el camino sólo puede añadir nuevas ligaduras de variables, porque un intento de añadir una ligadura distinta para una variable ya ligada generaría un fallo en la unificación. Cuando un camino en la búsqueda falla, el Prolog vuelve al punto de elección previo, y entonces podría tener que desligar algunas variables. Esto se hace guardando la pista de todas las variables que han sido ligadas en una pila llamada **pista**. Como cada nueva variable se liga mediante UNIFICA-VAR, la variable se coloca en la pila. Cuando el objetivo falla y es el momento de volver al punto de elección previo, cada una de las variables se desliga siendo eliminada de la pila.

Incluso los intérpretes de Prolog más eficientes requieren de varios miles de instrucciones de máquina por cada paso de inferencia debido al coste del índice de consulta, la unificación, y la construcción de la pila de las llamadas recursivas. En efecto, el intérprete siempre se comporta como si nunca hubiera visto el programa antes; por ejemplo, tiene que *encontrar* las cláusulas que emparejan con el objetivo. Por el otro lado, un programa compilado de Prolog es un procedimiento de inferencia para un conjunto específico de cláusulas, así que *se conoce* qué cláusulas emparejan con el objetivo. El Prolog básicamente genera un demostrador de teoremas en miniatura, y para ello, elimina mucho de los costes de la interpretación. También es posible **abrir-el-código** de la rutina de unificación en cada diferente llamada, y entonces, evitar el análisis explícito de la estructura de los términos. (Para más detalles sobre codificación abierta de la unificación, véase Warren *et al.*, (1977).)

El conjunto de instrucciones de los computadores actuales nos dan una similitud muy pobre con la semántica del Prolog, así que muchos compiladores de Prolog compilan a un lenguaje intermedio en vez de directamente al lenguaje máquina. El lenguaje intermedio más popular es el *Warren Abstract Machine*, o WAM, nombrado así gracias a David H. D. Warren, uno de los implementadores del primer compilador de Prolog. El WAM es un conjunto abstracto de instrucciones que es apropiado para el Prolog y se puede interpretar o traducir a lenguaje máquina. Otros compiladores traducen el Prolog a un lenguaje de alto nivel, como Lisp o C, y entonces utilizan el compilador de ese lenguaje para traducirlo al lenguaje máquina. Por ejemplo, la definición del predicado `Unir` se puede compilar en el código que se muestra en la Figura 9.8. Hay varios temas que vale la pena tener en cuenta:

- Mejor que tener que buscar las cláusulas `Unir` en la base de conocimiento, las cláusulas se pueden convertir en procedimientos y entonces las inferencias se llevan a cabo simplemente llamando al procedimiento.
- Tal como hemos descrito antes, las ligaduras de las variables se mantienen en la pila. El primer paso del procedimiento guarda el estado actual de la pila, y así se puede restaurar mediante `REINICIALIZA-PILA` si la cláusula falla. Esto deshace las ligaduras generadas por la primera llamada a `UNIFICA`.
- El truco consiste en el uso de las **continuaciones** para implementar los puntos de elección. Se puede pensar en una continuación como en el empaquetamiento de un procedimiento y una lista de argumentos que juntos definen lo que se debe ha-

PISTA

CÓDIGO ABIERTO

CONTINUACIONES

cer si el objetivo actual ha tenido éxito. Esto no ocurriría con un procedimiento como UNIR cuando tuviera éxito el objetivo, porque podría tenerlo por diversos caminos, y cada uno de ellos debería ser explorado. El argumento de la continuación resuelve este problema porque puede ser llamado cada vez que el objetivo tiene éxito. En el código de UNIR, si el primer argumento está vacío, entonces el predicado UNIR ha tenido éxito. Entonces LLAMAMOS a la continuación con las ligaduras adecuadas en la pila, para hacer seguidamente lo que se debería hacer. Por ejemplo, si la llamada a UNIR fuera en el nivel superior, la continuación imprimaría las ligaduras de las variables.

Antes del trabajo de Warren sobre compilación en Prolog, la programación lógica era demasiado lenta para un uso general. Los compiladores de Warren y otros códigos de Prolog permiten alcanzar velocidades que son competitivas con C en una variedad de puntos de referencia estándar (Van Roy, 1990). Por supuesto, el hecho de que uno pueda escribir un planificador o un analizador sintáctico de lenguaje natural en unas pocas docenas de líneas de código en Prolog lo hace algo más deseable que el prototipado en C, principalmente para proyectos de investigación en IA de escala pequeña.

La paralelización también puede proporcionar una aceleración sustancial. Hay dos fuentes principales de paralelismo. La primera, denominada **paralelismo-O**, viene de la posibilidad de unificar el objetivo con muchas cláusulas distintas de la base de conocimiento. Esto nos da una rama independiente en el espacio de búsqueda que nos puede llevar a una solución potencial, y todas estas ramas se pueden resolver en paralelo. La segunda, denominada **paralelismo-Y**, viene de la posibilidad de resolver en paralelo cada conjuntor del cuerpo de una implicación. El paralelismo-Y es más difícil de alcanzar, porque la solución global para la conjunción requiere ligaduras consistentes para todas las variables. Cada rama conjuntiva debe comunicarse con las otras ramas para asegurar una solución global.

PARALELISMO-O

PARALELISMO-Y

Inferencia redundante y bucles infinitos

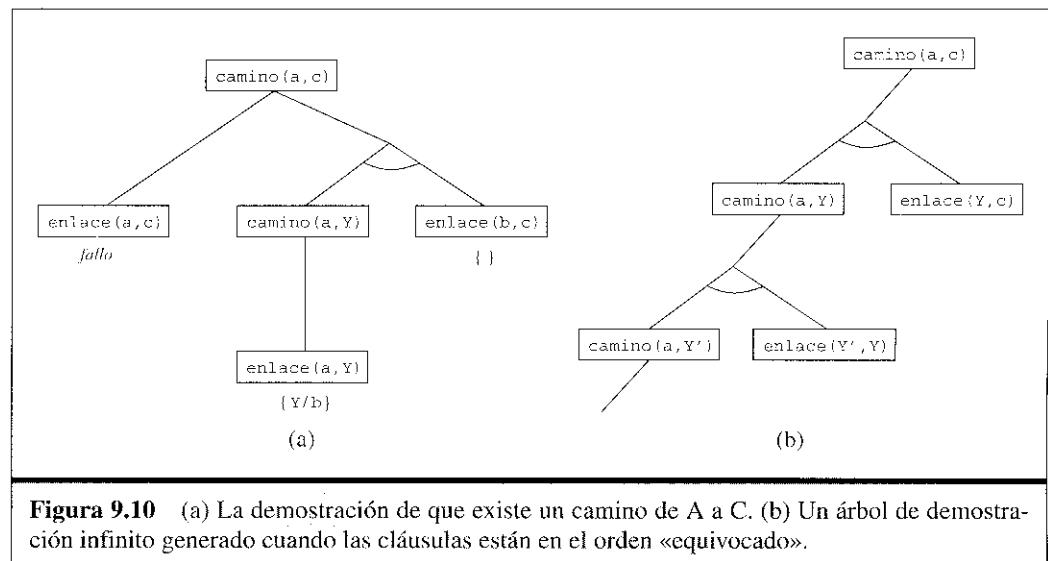
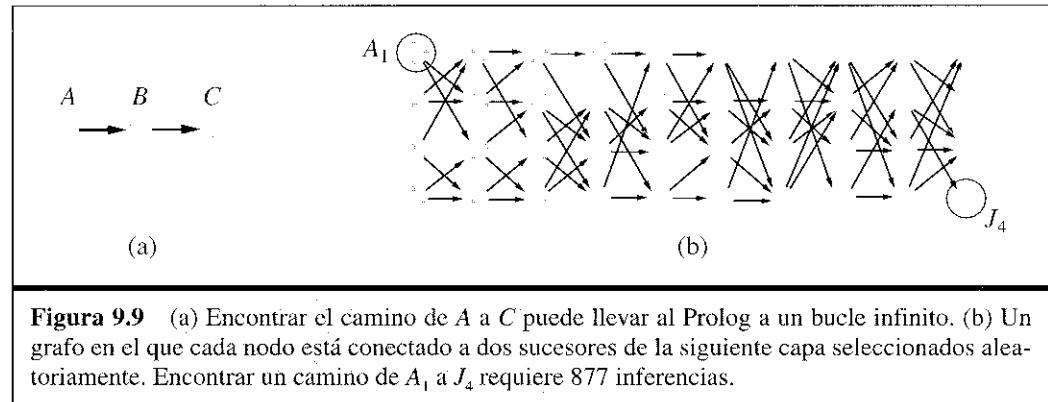
Ahora volvemos al talón de Aquiles del Prolog: la desunión entre la búsqueda de primero en profundidad y los árboles de búsqueda que incluyen estados repetidos y caminos infinitos. Considere el siguiente programa lógico, que decide si existe un camino entre dos nodos de un grafo dirigido:

```
camino(X, Z) :- enlace(X, Z)
camino(X, Z) :- enlace(X, Y), enlace(Y, Z)
```

En la Figura 9.9(a) se muestra un simple grafo de tres nodos, descrito por los hechos enlace(a, b) y enlace(b, c). Con este programa, la petición `camino(a, c)` genera la demostración que se muestra en la Figura 9.10(a). Por el otro lado, si ponemos las dos cláusulas en el orden

```
camino(X, Z) :- enlace(X, Y), enlace(Y, Z)
camino(X, Z) :- enlace(X, Z)
```

el Prolog seguirá un camino infinito como el que se ve en la Figura 9.10(b). Por lo tanto, el Prolog es **incompleto** como demostrador de teoremas con cláusulas positivas (incluso



con los programas Datalog, como muestra este ejemplo) debido a que para algunas bases de conocimiento falla al demostrar sentencias que están implicadas. Fíjese que el encadenamiento hacia delante no sufre de este problema: una vez $\text{camino}(a,b)$, $\text{camino}(b,c)$ y $\text{camino}(a,c)$ son inferidas, el encadenamiento hacia delante finaliza con éxito.

El encadenamiento hacia atrás en primero en profundidad también tiene problemas con los cálculos redundantes. Por ejemplo, cuando se busca un camino de A_1 a J_4 , como se muestra en la Figura 9.9(b), el Prolog realiza 877 inferencias, muchas de las cuales consisten en encontrar todos los caminos posibles a nodos a los que el objetivo no puede de enlazarse. Esto es similar al problema de los estados repetidos que se habló en el Capítulo 3. La cantidad total de inferencias puede ser exponencial respecto al número de hechos base que son generados. Si en vez de esto aplicamos el encadenamiento hacia delante, como mucho se pueden generar n^2 hechos $\text{camino}(X,Y)$ enlazando n nodos. Para el problema de la Figura 9.9(b) sólo se necesitan 62 inferencias.

El encadenamiento hacia delante aplicado en problemas de búsqueda sobre grafos es un ejemplo de la **programación dinámica**, en la que las soluciones de los subproblemas se van construyendo incrementalmente a partir de subproblemas más pequeños y se guardan para evitar posteriores recálculos. Obtenemos un efecto similar en el sistema de encadenamiento hacia atrás utilizando la **memorización**, es decir, guardando las soluciones de los subobjetivos a medida que se encuentran y entonces se reutilizan aquellas soluciones cuando el subobjeivo se repite, en vez de repetir el cálculo ya realizado. Este es el enfoque que se utiliza en los sistemas de **programación lógica basada en tablas**, que utilizan unos mecanismos de almacenamiento y recuperación eficientes para realizar la memorización. La programación lógica basada en tablas combina la orientación al objetivo del encadenamiento hacia atrás con la eficiencia de la programación dinámica del encadenamiento hacia delante. Y además es un proceso completo para los programas Datalog, lo que significa que el programador no necesita preocuparse más acerca de los bucles infinitos.

Programación lógica con restricciones

En nuestra exposición acerca del encadenamiento hacia delante (Apartado 9.3) mostramos cómo se podían codificar los problemas de satisfacción de restricciones (PSR) mediante cláusulas positivas. El Prolog estándar resuelve este tipo de problemas exactamente de la misma manera como lo hace el algoritmo con *backtracking* dado en la Figura 5.3.

Ya que el *backtracking* (vuelta hacia atrás) enumera los dominios de las variables, sólo trabaja con PSRs con **dominio finito**. En términos de Prolog, debe haber un número finito de soluciones para cada objetivo con las variables desligadas. (Por ejemplo, el objetivo `dif(q, sa)`, que dice que Queensland y el Sur de Australia deben tener colores diferentes, tiene seis soluciones si se permiten tres colores.) Los PSRs con dominios infinitos (por ejemplo con variables enteras o reales) requieren algoritmos bastante diferentes, tales como la propagación de ligaduras o la programación lineal.

La siguiente cláusula tiene éxito si tres números satisfacen la desigualdad del triángulo:

```
triangulo(X, Y, Z) :-  
    X >= 0, Y >= 0, Z >= 0, X + Y >= Z, Y + Z >= X, X + Z >= Y
```

Si le hacemos al Prolog la petición `triangulo(3, 4, 5)` trabaja bien. Por el otro lado, si preguntamos por `triangulo(3, 4, Z)`, no se encontrará ninguna solución porque el subobjetivo `Z >= 0` no puede ser manejado por el Prolog. La dificultad es que en Prolog las variables deben estar en uno o dos estados: ligadas o desligadas a un término en particular.

Ligar una variable a un término concreto se puede ver como un tipo extremo de restricción, a saber, una restricción de igualdad. La **programación lógica con restricciones** (PLR) permite que las variables estén más bien *restringidas* que *ligadas*. Una solución para un programa lógico con restricciones es el conjunto más específico de restricciones en las variables de la petición que se pueden derivar de la base de conocimiento. Por ejemplo, la solución a la petición de `triangulo(3, 4, Z)` es la restricción

$7 \geq z \geq 1$. Los programas lógicos con restricciones son tan sólo un caso especial de la PLR en los que las restricciones solución deben ser restricciones de igualdad, o ligaduras.

Los sistemas de PLR incorporan diversos algoritmos de resolución de restricciones para las restricciones que se permiten en el lenguaje. Por ejemplo, un sistema que permite desigualdades lineales en variables reales podría incluir un algoritmo de programación lineal para resolver dichas restricciones. Los sistemas de PLR también pueden adoptar un enfoque mucho más flexible para resolver las peticiones de la programación lógica estándar. Por ejemplo, en vez de primero en profundidad o *backtracking* (vuelta hacia atrás) de izquierda a derecha, podrían utilizar uno de los algoritmos, más eficientes, que se mostraron en el Capítulo 5, incluyendo las heurísticas de ordenación de los conjuntos, salto atrás, condicionamiento del corte del conjunto, etc. Por lo tanto, los sistemas de PLR combinan elementos de los algoritmos de satisfacción de restricciones, de la programación lógica, y de las bases de datos deductivas.

Los sistemas de PLR también pueden tomar ventaja de la variedad de optimizaciones de búsqueda descritas en el Capítulo 5, tales como la ordenación de valores y variables, la comprobación hacia delante y el *backtracking* inteligente. Se han definido una gran diversidad de sistemas que permiten al programador un mayor control sobre el orden de búsqueda en la ejecución de la inferencia. Por ejemplo, el lenguaje MRS (Genesereth y Smith, 1981; Russell, 1985) le permite al programador escribir **meta reglas** para determinar qué conjunto se intenta primero. El usuario podría escribir una regla diciendo que el objetivo con menos variables debería intentarse antes o podría escribir reglas específicas del dominio para predicados concretos.

META REGLAS

9.5 Resolución

La última de nuestras tres familias de sistemas lógicos está basada en la **resolución**. En el Capítulo 7 vimos que la resolución proposicional es un procedimiento de inferencia mediante refutación que es completo para la lógica proposicional. En esta sección veremos cómo ampliar la resolución a la lógica de primer orden.

El tema de la existencia de procedimientos de demostración completos concierne directamente a los matemáticos. Si se puede encontrar un procedimiento de demostración completo para los enunciados matemáticos, pueden suceder dos cosas: primero, todas las conjecturas se pueden establecer mecánicamente; segundo, todas las matemáticas se pueden establecer como la consecuencia lógica de un conjunto de axiomas fundamentales. El tema de la completitud por lo tanto ha generado algunos de los trabajos matemáticos más importantes del siglo xx. En 1930, el matemático alemán Kurt Gödel demostró el primer **teorema de la completitud** para la lógica de primer orden, mostrando que una sentencia implicada tiene una demostración finita. (Realmente no se encontró un procedimiento *práctico* de demostración hasta que J. A. Robinson publicó su algoritmo de resolución en 1965.) En 1931, Gödel demostró el aún más famoso **teorema de la incompletitud**. El teorema muestra que un sistema lógico que incluye el principio de la inducción (sin el cual muy pocas de las matemáticas discretas se pueden construir) es

TEOREMA DE LA COMPLETITUD

TEOREMA DE LA INCOMPLETITUD

necesariamente incompleto. De aquí, que haya sentencias implicadas, pero no haya disponible una demostración finita en el sistema. La aguja puede estar en un pajar metafórico, pero ningún procedimiento puede garantizar que será encontrada.

A pesar del teorema de Gödel, los demostradores de teoremas basados en la resolución han sido utilizados para derivar teoremas matemáticos, incluyendo varios de los que no se conocían, previamente una demostración. Los demostradores de teoremas también han sido utilizados para verificar diseños de *hardware* y para generar programas correctos desde el punto de vista lógico, entre otras aplicaciones.

Formas normales conjuntivas en lógica de primer orden

Como en el caso proposicional, la resolución en primer orden requiere que las sentencias estén en la **forma normal conjuntiva** (FNC), es decir, una conjunción de cláusulas, donde cada cláusula es una disyunción de literales⁶. Los literales pueden contener variables, que se asume están cuantificadas universalmente. Por ejemplo, la sentencia

$$\forall x \text{ Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Hostil}(z) \Rightarrow \text{Criminal}(x)$$

se transforma a FNC como

$$\neg \text{Americano}(x) \vee \neg \text{Arma}(y) \vee \neg \text{Vende}(x, y, z) \vee \neg \text{Hostil}(z) \vee \text{Criminal}(x)$$



Cada sentencia en lógica de primer orden se puede convertir a una sentencia en FNC que es equivalente inferencialmente. En concreto, la sentencia en FNC será *insatisfacible* sólo cuando la sentencia original lo sea, así disponemos de una base para hacer demostraciones mediante contradicción con sentencias en FNC.

El procedimiento para la conversión a la FNC es muy parecido al del caso proposicional que vimos en el Apartado 7.5. La principal diferencia viene de la necesidad de eliminar los cuantificadores universales. Mostraremos el procedimiento transformando la sentencia «Todo el mundo que ama a los animales es amado por alguien», o

$$\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Ama}(x, y)] \Rightarrow [\exists y \text{ Ama}(y, x)]$$

Los pasos a seguir son los siguientes:

- **Eliminación de las implicaciones:**

$$\forall x [\neg \forall y \neg \text{Animal}(y) \vee \text{Ama}(x, y)] \vee [\exists y \text{ Ama}(y, x)]$$

- **Anidar las \neg :** además de las reglas usuales para las conectivas negadas, necesitamos las reglas para los cuantificadores negados. De este modo tenemos

$$\begin{array}{ll} \neg \forall x p & \text{se convierte en } \exists x \neg p \\ \neg \exists x p & \text{se convierte en } \forall x \neg p \end{array}$$

⁶ Una cláusula también se puede representar como una implicación con una conjunción de átomos a la izquierda y una disyunción de átomos a la derecha, tal como se muestra en el Ejercicio 7.12. Esta forma, que algunas veces se denomina **forma de Kowalski** cuando se escribe con un símbolo de implicación de derecha a izquierda (Kowalski, 1979b), a menudo es más fácil de leer.

Nuestra sentencia pasa a través de las siguientes transformaciones:

$$\begin{aligned} \forall x & [\exists y \neg(\neg Animal(y) \vee Ama(x, y))] \vee [\exists y Ama(y, x)] \\ \forall x & [\exists y \neg\neg Animal(y) \wedge \neg Ama(x, y)] \vee [\exists y Ama(y, x)] \\ \forall x & [\exists y Animal(y) \wedge \neg Ama(x, y)] \vee [\exists y Ama(y, x)] \end{aligned}$$

Fíjese en que el cuantificador universal ($\forall y$) de la premisa de la implicación se convierte en un cuantificador existencial. La sentencia ahora se lee «Hay algún animal que x no ama, o (si éste no es el caso) alguien ama a x ». Está claro que el significado de la sentencia original se mantiene.

- **Estandarizar las variables:** para las sentencias del tipo $(\forall x P(x)) \vee (\exists x Q(x))$, que utilizan el mismo nombre de variable dos veces, se cambia una de las dos variables. Esto evita la confusión posterior al eliminar los cuantificadores. Así tenemos

$$\forall x [\exists y Animal(y) \wedge \neg Ama(x, y)] \vee [\exists z Ama(z, x)].$$

SKOLEMIZACIÓN

- **Skolemizar:** la **Skolemización** es el proceso de borrar los cuantificadores existenciales mediante su eliminación. En el caso más sencillo, se aplica la regla de la Especificación Existencial del Apartado 9.1: transformar $\exists x P(x)$ a $P(A)$, donde A es una constante nueva. Si aplicamos esta regla a nuestra sentencia del ejemplo, obtenemos

$$\forall x [Animal(A) \wedge \neg Ama(x, A)] \vee Ama(B, x)$$

en donde se obtiene un significado totalmente erróneo: la sentencia dice que todo el mundo o no puede amar al animal A o que es amado por la entidad B . De hecho, nuestra sentencia original le permite a cada persona no poder amar a un animal diferente o ser amado por otra persona. Por lo tanto, lo que queremos es obtener las entidades de Skolem que dependen de x :

$$\forall x [Animal(F(x)) \wedge \neg Ama(x, F(x))] \vee Ama(G(x), x)$$

FUNCIÓN DE SKOLEM

Aquí F y G son **funciones de Skolem**. La regla general es que los argumentos de la función de Skolem dependen de las variables cuantificadas universalmente cuyo ámbito abarca a los cuantificadores existenciales. Igual que con la Especificación Existencial, la sentencia skolemizada es *satisfacible* sólo cuando la sentencia original lo es.

- **Eliminar los cuantificadores universales:** en este punto, todas las variables que quedan deben estar cuantificadas universalmente. Más aún, la sentencia es equivalente a una en la que todos los cuantificadores universales se han desplazado a la izquierda. Por lo tanto podemos eliminar los cuantificadores universales:

$$[Animal(F(x)) \wedge \neg Ama(x, F(x))] \vee Ama(G(x), x)$$

- **Distribuir la \wedge respecto la \vee :**

$$[Animal(F(x)) \vee Ama(G(x), x)] \wedge [\neg Ama(x, F(x)) \vee Ama(G(x), x)]$$

Este paso también puede requerir extraer las conjunciones y disyunciones anidadas.

La sentencia ahora está en FNC y está compuesta por dos cláusulas. Es algo ilegible. (Podría ser de ayuda explicar que la función de Skolem $F(x)$ se refiere al animal que potencialmente no ama x , mientras que $G(x)$ se refiere a alguien que podría amar a x .) Afortunadamente, las personas rara vez necesitan ver las sentencias en FNC (el proceso de traducción es muy fácilmente automatizable).

La regla de inferencia de resolución

La regla de la resolución para la lógica de primer orden es simplemente una versión elevada de la regla de resolución proposicional que hemos dado en el Apartado 7.5. Dos cláusulas, que se asume están con las variables estandarizadas y así no comparten ninguna variable, se pueden resolver si contienen literales complementarios. Los literales proposicionales complementarios son complementarios si uno es la negación del otro, los literales en lógica de primer orden son complementarios si uno se *unifica* con la negación del otro. De este modo tenemos

$$\frac{\ell_1 \vee \dots \vee \ell_k \quad m_1 \vee \dots \vee m_n}{\text{SUST}(\theta, \ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

donde $\text{UNIFICA}(\ell_i, \neg m_j) = \theta$. Por ejemplo, podemos resolver las dos cláusulas

$$[\text{Animal}(F(x)) \vee \text{Ama}(G(x), x)] \quad \text{y} \quad [\neg \text{Ama}(u, v) \vee \neg \text{Mata}(u, v)]$$

eliminando los literales complementarios $\text{Ama}(G(x), x)$ y $\neg \text{Ama}(u, v)$, con el unificador $\theta = \{u/G(x), v/x\}$, para producir la cláusula **resolvente**

$$[\text{Animal}(F(x)) \vee \neg \text{Mata}(G(x), x)]$$

RESOLUCIÓN BINARIA

La regla que acabamos de dar es la regla de **resolución binaria**, porque resuelve exactamente dos literales. La regla de resolución binaria por sí misma no nos da un procedimiento de inferencia completo. La regla de resolución general resuelve los subconjuntos de los literales en cada cláusula que es unificable. Un enfoque alternativo es ampliar la **factorización** (la eliminación de los literales redundantes) en el caso de la lógica de primer orden. La factorización proposicional reduce dos literales a uno si son *idénticos*; la factorización de primer orden reduce dos literales a uno si éstos son *unificables*. El unificador debe ser aplicado a la cláusula entera. La combinación de la resolución binaria con la factorización sí es completa.

Demostraciones de ejemplo

La resolución demuestra que $BC \models \alpha$ probando que $BC \wedge \neg \alpha$ es *insatisfacible*, por ejemplo, derivando la cláusula vacía. El enfoque algorítmico es idéntico al caso proposicional, descrito en la Figura 7.12, así que no lo repetiremos aquí. En vez de ello, daremos dos demostraciones de ejemplo. La primera es el ejemplo del crimen del Apartado 9.3. Las sentencias en FNC son:

$$\begin{aligned}
 & \neg\text{Americano}(x) \vee \neg\text{Arma}(y) \vee \neg\text{Vende}(x, y, z) \vee \neg\text{Hostil}(z) \vee \text{Criminal}(x) \\
 & \neg\text{Misil}(x) \vee \neg\text{Tiene}(Nono, x) \vee \text{Vende}(West, x, Nono) \\
 & \neg\text{Enemigo}(x, \text{América}) \vee \text{Hostil}(x) \\
 & \neg\text{Misil}(x) \vee \text{Arma}(x) \\
 & \text{Tiene}(Nono, M_1). \quad \text{Misil}(M_1) \\
 & \text{Americano}(West). \quad \text{Enemigo}(Nono, \text{América})
 \end{aligned}$$

También incluimos el objetivo negado, $\neg\text{Criminal}(West)$. La demostración por resolución se muestra en la Figura 9.11. Fíjese en la estructura: un comienzo de la «columna» sencilla, con la cláusula objetivo, que se resuelve a través de las cláusulas de la base de conocimiento hasta que se genera la cláusula vacía. Esto es algo característico de la resolución con las bases de conocimiento con cláusulas de Horn. De hecho, las cláusulas a lo largo de la columna se corresponden *exactamente* con los valores consecutivos de las variables de los *objetivos* del algoritmo de encadenamiento hacia atrás de la Figura 9.6. Esto se debe a que siempre escogemos resolver con una cláusula cuyo literal positivo se unifique con el literal más a la izquierda de la cláusula «actual» de la columna; y esto es lo que ocurre exactamente con el encadenamiento hacia atrás. De esta manera, el encadenamiento hacia atrás es un caso especial de la resolución, con una estrategia de control concreta para decidir qué resolución se realiza a continuación.

Nuestro segundo ejemplo hace uso de la Skolemización y trata con cláusulas que no son positivas. Esto genera una estructura de demostración algo más compleja. En lenguaje natural el problema se presenta así:

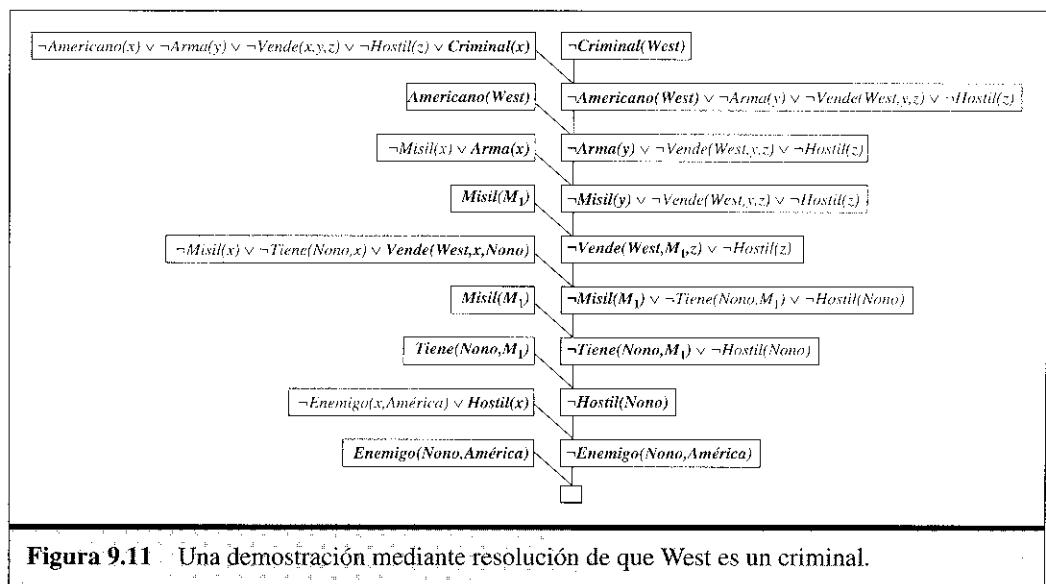
Todo el mundo que ama a todos los animales es amado por alguien.

Cualquiera que mate a un animal no es amado por nadie.

Jack ama a todos los animales.

Jack o Curiosity mataron a la gata, que se llama Tuna.

¿Mató Curiosity a la gata?



Primero expresamos, en lógica de primer orden, las sentencias originales, algún conocimiento de base, y el objetivo G negado:

- A. $\forall x [\forall y \text{Animal}(y) \Rightarrow \text{Ama}(x, y)] \Rightarrow [\exists y \text{Ama}(y, x)]$
 - B. $\forall x [\forall y \text{Animal}(y) \wedge \text{Mata}(x, y)] \Rightarrow [\forall z \neg \text{Ama}(z, x)]$
 - C. $\forall x \text{Animal}(x) \Rightarrow \text{Ama}(\text{Jack}, x)$
 - D. $\text{Mata}(\text{Jack}, \text{Tuna}) \vee \text{Mata}(\text{Curiosity}, \text{Tuna})$
 - E. $\text{Gata}(\text{Tuna})$
 - F. $\forall x \text{Gata}(x) \Rightarrow \text{Animal}(x)$
 - G. $\neg \text{Mata}(\text{Curiosity}, \text{Tuna})$

Ahora aplicamos el procedimiento de conversión para transformar cada sentencia a FNC:

- A1. $\text{Animal}(F(x)) \vee \text{Ama}(G(x), x)$
 A2. $\neg\text{Ama}(x, F(x)) \vee \text{Ama}(G(x), x)$

B. $\neg\text{Animal}(y) \vee \neg\text{Mata}(x, y) \vee \neg\text{Ama}(z, x)$
 C. $\neg\text{Animal}(x) \vee \text{Ama}(\text{Jack}, x)$
 D. $\text{Mata}(\text{Jack}, \text{Tuna}) \vee \text{Mata}(\text{Curiosity}, \text{Tuna})$
 E. $\text{Gata}(\text{Tuna})$
 F. $\neg\text{Gata}(x) \vee \text{Animal}(x)$
 G. $\neg\text{Mata}(\text{Curiosity}, \text{Tuna})$

La demostración mediante resolución de que Curiosity mató a la gata se muestra en la Figura 9.12. En lenguaje natural, la demostración se podría parafrasear como sigue:

Supongamos que Curiosity no mató a Tuna. Sabemos que lo hizo Jack o Curiosity, por lo tanto tiene que ser Jack. Tuna es una gata y los gatos son animales. Ya que cualquiera que mate a un animal no es amado por nadie, entonces deberíamos pensar que nadie ama a Jack. Por el otro lado, Jack ama a los animales, así que alguien lo ama, y entonces llegamos a una contradicción. De todo ello concluimos que Curiosity mató a la gata.

La demostración responde a la pregunta «¿Curiosity mató a la gata?», pero a menudo queremos realizar preguntas más generales, del tipo «¿Quién mató a la gata?». La resolución puede responder a este tipo de preguntas, pero necesita hacer un poco más de esfuerzo para obtener la respuesta. El objetivo es $\exists w \text{ Mata}(w, Tuna)$, que cuando se niega

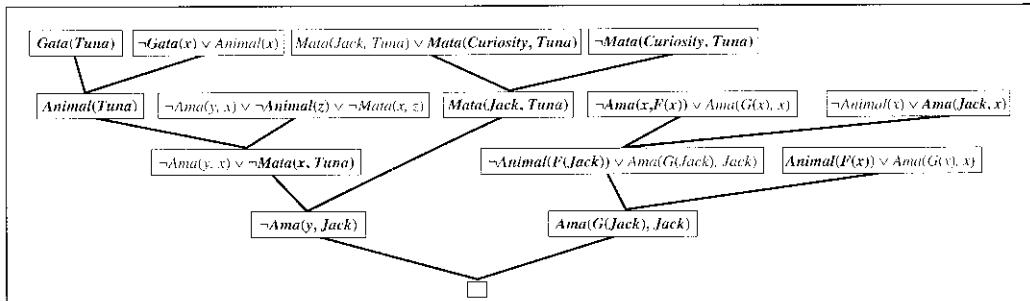


Figura 9.12 Una demostración mediante resolución de que Curiosity mató a la gata. Fíjese en el uso de la factorización en la derivación de la cláusula $\text{Ama}(G(\text{Jack}), \text{Jack})$.

DEMOSTRACIÓN NO CONSTRUCTIVA**LITERAL DE RESPUESTA****COMPLETITUD DE LA REFUTACIÓN**

queda de la forma $\neg \text{Mata}(w, \text{Tuna})$ en FNC. Al repetir la demostración de la Figura 9.12 con el nuevo objetivo negado, obtenemos un árbol de demostración algo similar, pero con la sustitución $\{w/\text{Curiosity}\}$ en uno de los pasos. Así, en este caso, averiguar quién mató a la gata es justamente una forma de guardar la pista de las ligaduras de la variable de la petición durante la demostración.

Desafortunadamente, la resolución puede generar **demostraciones no constructivas** cuando trabaja con objetivos existenciales. Por ejemplo, $\neg \text{Mata}(w, \text{Tuna})$ se resuelve con $\text{Mata}(\text{Jack}, \text{Tuna}) \vee \text{Mata}(\text{Curiosity}, \text{Tuna})$ para dar $\text{Mata}(\text{Jack}, \text{Tuna})$, y que se resuelve, otra vez, con $\neg \text{Mata}(w, \text{Tuna})$ para obtener la cláusula vacía. Fíjese en que w tiene dos ligaduras diferentes en esta demostración; la resolución nos dice que sí, que alguien mató a Tuna: Jack o Curiosity. ¡Esto no nos sorprende! Una solución consiste en restringir los pasos permitidos de la resolución de manera que las variables de la petición se puedan ligar una sola vez en una demostración dada; entonces necesitamos ser capaces de volver sobre las ligaduras posibles. Otra solución es añadir un **literal de respuesta** especial al objetivo negado, que se convierte en $\neg \text{Mata}(w, \text{Tuna}) \vee \text{Respuesta}(w)$. Entonces, ahora el proceso de resolución genera una respuesta cuando se genera una cláusula con sólo el literal de respuesta. Para la demostración de la Figura 9.12 sería $\text{Respuesta}(\text{Curiosity})$. La demostración no constructiva generaría la cláusulas $\text{Respuesta}(\text{Curiosity}) \vee \text{Respuesta}(\text{Jack})$, que no constituye en sí una respuesta.

Compleitud de la resolución

En esta sección damos una demostración de la completitud de la resolución. Se puede saltar sin temor por aquellos que están dispuestos a desafiar la fe.

Demostraremos que la resolución es un procedimiento de **refutación completa**, lo que significa que *si* un conjunto de sentencias es *insatisfacible*, entonces la resolución siempre será capaz de derivar una contradicción. La resolución no se puede utilizar para generar todas las consecuencias lógicas de un conjunto de sentencias, pero se puede utilizar para establecer si una sentencia dada se deduce de un conjunto de sentencias. De aquí, que se pueda utilizar para encontrar todas las respuestas a una pregunta dada, utilizando el método del objetivo negado que se ha descrito previamente en este capítulo.

Tomaremos como premisa que cualquier sentencia en lógica de primer orden (sin igualdad) se puede transformar en un conjunto de cláusulas en FNC. Esto se puede demostrar por inducción sobre la forma de la sentencia, utilizando las sentencias atómicas como el caso base (Davis y Putnam, 1960). Nuestro objetivo, por lo tanto, es demostrar lo siguiente: *si S es un conjunto de cláusulas insatisfacible, entonces la aplicación de un número finito de pasos de resolución sobre S nos dará una contradicción*.

Nuestro esbozo de la demostración se basa en la demostración original de Robinson, con algunas simplificaciones cogidas de Genesereth y Nilsson (1987). La estructura básica de la demostración se muestra en la Figura 9.13; que procede como sigue:

1. Primero, observamos que si S es *insatisfacible*, entonces debe existir un subconjunto de *instancias base* de las cláusulas de S que también es *insatisfacible* (teorema de Herbrand).

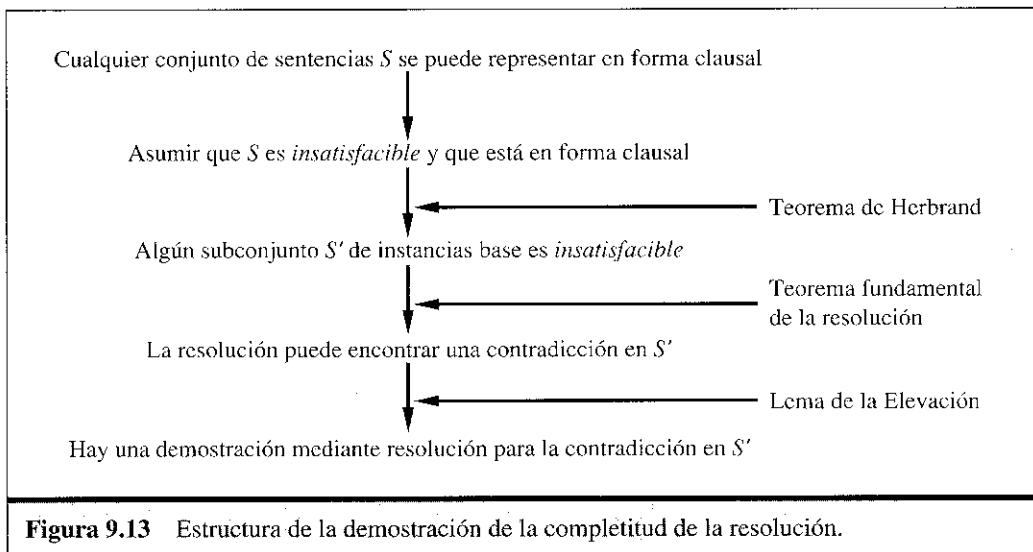


Figura 9.13 Estructura de la demostración de la completitud de la resolución.

2. Entonces apelamos al **teorema fundamental de la resolución** del Capítulo 7, que establece que la resolución proposicional es completa para sentencias base.
3. Luego utilizamos el **lema de la elevación** que muestra que para cualquier demostración mediante resolución proposicional que utiliza un conjunto de sentencias base, existe su correspondiente demostración mediante resolución de primer orden que utiliza las sentencias de primer orden de las que se obtuvieron las sentencias base.

Para llevar a cabo el primer paso necesitaremos tres conceptos nuevos:

- **Universo de Herbrand:** si S es un conjunto de cláusulas, entonces H_s , el universo de Herbrand de S , es el conjunto de todos los términos base que se pueden construir a partir de:

- a) Los símbolos de función en S , si hay alguno.
- b) Los símbolos de constante en S , si hay alguno; sino, el símbolo de constante A .

Por ejemplo, si S contiene la cláusula $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, entonces H_s es el siguiente conjunto de términos base:

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \dots\}.$$

UNIVERSO DE HERBRAND

SATURACIÓN

BASE DE HERBRAND

- **Saturación (Instancias Base):** si S es el conjunto de cláusulas, y P es el conjunto de términos base, entonces $P(S)$, la saturación de S respecto a P . Es el conjunto de las cláusulas base que se obtienen de aplicar todas las sustituciones posibles consistentes de términos base en P , sobre las variables de S .
- **Base de Herbrand:** la saturación (o las instancias base) de un conjunto S de cláusulas con respecto a su universo de Herbrand se denomina Base de Herbrand de S , que se escribe $Hs(S)$. Por ejemplo, si S contiene tan sólo la cláusula que hemos dado anteriormente, entonces $Hs(S)$ es el conjunto finito de cláusulas

EL TEOREMA DE INCOMPLETITUD DE GÖDEL

Ampliando un poco el lenguaje de la lógica de primer orden para permitir el uso de la **inducción matemática**, Gödel fue capaz de demostrar, en su **teorema de la incompletitud**, que hay sentencias aritméticas que son verdaderas y que no se pueden demostrar.

La demostración del teorema de la incompletitud es algo que va más allá del alcance de este libro, y ocupa, como lo hace, un mínimo de 30 páginas, pero aquí podemos dar una breve pista. Comenzamos con la teoría lógica de los números. En esta teoría, hay una sola constante, el 0, y una sola función, S (la función sucesor). En el modelo deseado, $S(0)$ denota 1, $S(S(0))$ denota 2, etc.; por lo tanto, el lenguaje tiene nombres para todos los números naturales. El vocabulario también incluye los símbolos de función $+$, \times , y Exp (exponenciación) y el conjunto habitual de conectivas lógicas y cuantificadores. El primer paso consiste en fijarse en que el conjunto de sentencias que podemos escribir en este lenguaje se puede enumerar. (Imagine definir un orden alfabetico sobre los símbolos y entonces organizar, por orden alfabetico, cada conjunto de sentencias de longitud 1, 2, etc.) Entonces podemos enumerar cada sentencia α con un único número natural $\# \alpha$ (el **número de Gödel**). Esto es crucial: la teoría de los números contiene un nombre para cada una de sus sentencias. De forma similar, podemos enumerar cada posible demostración D con un número de Gödel $G(D)$, porque una demostración simplemente es una secuencia finita de sentencias.

Ahora suponga que tenemos un conjunto A de sentencias que es enumerable recursivamente y que es el conjunto de los enunciados acerca de los números naturales que son verdaderos. Y dado que A se puede nombrar mediante un conjunto de enteros, podemos imaginar escribir en nuestro lenguaje una sentencia $\alpha(j, A)$ de la siguiente manera:

$\forall i \ i$ no es el número de Gödel de una demostración de una sentencia cuyo número de Gödel es j , donde la demostración sólo utiliza las premisas de A .

Entonces dejemos que σ sea la sentencia $\alpha(\# \sigma, A)$, es decir, una sentencia que define su propia improbabilidad a partir de A . (Que esta sentencia siempre existe es verdadero, pero no del todo obvio.)

Ahora hacemos el siguiente argumento ingenioso: supongamos que σ es probable a partir de A ; entonces σ es falsa (porque σ dice que no se puede demostrar). Pero entonces tenemos una sentencia falsa que es probable a partir de A , y por tanto A no tiene sólo sentencias verdaderas (generando una violación de nuestra premisa). Por lo tanto σ no es probable a partir de A . Y esto es exactamente lo que σ proclama; de aquí que σ sea una sentencia verdadera.

Así hemos demostrado (eliminando 29 páginas y media) que para cualquier conjunto de sentencias verdaderas acerca de la teoría de los números, y en concreto, cualquier conjunto de los axiomas básicos, hay otras sentencias verdaderas que *no se pueden* demostrar a partir de esos axiomas. Esto establece, entre otras cosas, que nunca podemos demostrar todos los teoremas de las matemáticas *a partir de un sistema de axiomas dado*. Está claro que esto fue un importante descubrimiento para los matemáticos. Su utilidad para la IA ha sido ampliamente debatida, comenzando con las especulaciones por el propio Gödel. Retomaremos este debate en el Capítulo 26.

$$\begin{aligned} & \{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B), \\ & \neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B), \\ & \neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B), \\ & \neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \dots \} \end{aligned}$$

TEOREMA DE
HERBRAND

Estas definiciones nos permiten definir una variante del **teorema de Herbrand** (Herbrand, 1930):

Si un conjunto de cláusulas S es *insatisfacible*, entonces debe existir un subconjunto finito de $Hs(S)$ que también sea *insatisfacible*.

Sea S' el subconjunto finito de sentencias base. Ahora podemos apelar al teorema fundamental de la resolución (Apartado 7.5) para demostrar que el **cierre de la resolución** $CR(S')$ contiene la cláusula vacía. Es decir, ejecutando la resolución proposicional para obtener una conclusión a partir de S' genera una contradicción.

Ahora acabamos de establecer que siempre hay una demostración mediante resolución que involucra a algún subconjunto finito de la base de Herbrand de S , el siguiente paso consistirá en demostrar que hay una demostración mediante resolución utilizando las propias cláusulas de S , que no son necesariamente cláusulas base. Comenzamos teniendo en cuenta una sola aplicación de la regla de la resolución. El lema básico de Robinson implica el siguiente hecho:

Sean C_1 y C_2 dos cláusulas con variables no compartidas, y sean C'_1 y C'_2 las instancias base de C_1 y C_2 . Si C' es el resolvente de C'_1 y C'_2 entonces debe existir una cláusula C tal que (1) C sea el resolvente de C_1 y C_2 y (2) C' sea una instancia base de C .

LEMA DE ELEVACIÓN

Esto se llama un **lema de elevación**, porque eleva un paso de demostración a partir de cláusulas base a cláusulas de primer orden generales. Para demostrar su lema de elevación básico, Robinson tenía que inventar la unificación y derivar todas las propiedades de los unificadores más generales. Más que repetir aquí la demostración, simplemente ilustramos el lema:

$$\begin{aligned} C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\ C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\ C'_1 &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C'_2 &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\ C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B) \end{aligned}$$

Vemos que efectivamente, C' es una instancia base de C . Por lo general, para que C'_1 y C'_2 tengan algún resolvente, deben estar construidos aplicando primero el unificador más general a C_1 y C_2 partiendo de una pareja suya de literales complementarios. Del lema de la elevación, es fácil derivar un enunciado similar acerca de cualquier secuencia de aplicaciones de la regla de la resolución:

Para cualquier cláusula C' del cierre de la resolución de S' hay una cláusula C en el cierre de la resolución de S , tal que C' es una instancia base de C y la derivación de C es de la misma longitud que la derivación de C' .

De este hecho, se sigue que si aparece la cláusula vacía en el cierre de la resolución de S' , también debe aparecer en el cierre de la resolución de S . Esto se debe a que la cláu-

sula vacía no puede ser una instancia base de cualquier otra cláusula. Para recapitular: hemos demostrado que si S es *insatisfacible*, entonces hay una derivación finita de la cláusula vacía mediante la regla de la resolución.

La elevación de la demostración del teorema a partir de las cláusulas base a las cláusulas de primer orden nos proporciona un incremento vasto de poder. Este incremento viene del hecho de que la demostración en primer orden necesita variables instanciadas sólo a medida que es necesario para la demostración, mientras que los métodos con cláusulas base eran necesarios para evaluar un número enorme de instanciaciones arbitrarias.

Manejar la igualdad

Ninguno de los métodos de inferencia descritos hasta ahora manejan la igualdad. Hay tres enfoques diferentes que se pueden tomar. El primer enfoque consiste en axiomatizar la igualdad (anotar las sentencias con la relación de igualdad en la base de conocimiento). Necesitamos decir que la igualdad es reflexiva, simétrica y transitiva, y también necesitamos decir que podemos sustituir objetos iguales en cualquier predicado o función.

Por lo que necesitamos tres axiomas básicos, y también uno para cada predicado y función:

$$\begin{aligned} & \forall x \ x = x \\ & \forall x, y \ x = y \Rightarrow y = x \\ & \forall x, y, z \ x = y \wedge y = z \Rightarrow x = z \\ & \forall x, y \ x = y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\ & \forall x, y \ x = y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\ & \vdots \\ & \forall w, x, y, z \ w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z)) \\ & \forall w, x, y, z \ w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z)) \\ & \vdots \end{aligned}$$

Dadas estas sentencias, un procedimiento de inferencia estándar como la resolución puede realizar tareas que requieran razonamiento con igualdad, tales como la resolución de ecuaciones matemáticas.

Otra forma de tratar la igualdad es mediante una regla de inferencia adicional. La regla más sencilla, denominada **demodulación**, toma una cláusula unitaria $x = y$ y sustituye la y por cualquier término que se unifica con x en cualquier otra cláusula. Más formalmente, tendríamos

- **Demodulación:** para cualquier término x, y y z , donde $\text{UNIFICA}(x, z) = \theta$ y $m_n[z]$ es un literal que contiene a z :

$$\frac{x = y, \quad m_1 \vee \dots \vee m_n[z]}{m_1 \vee \dots \vee m_n[\text{SUST}(\theta, y)]}$$

La demodulación se utiliza por lo general para simplificar expresiones que utilizan grupos de aserciones del tipo $x + 0 = x$, $x^1 = x$, etc. La regla también se puede ampliar para manejar cláusulas no unitarias en las que aparece una igualdad de literales:

PARAMODULACIÓN

- **Paramodulación:** para cualquier término x, y y z , donde $\text{UNIFICA}(x, z) = \theta$,

$$\frac{\ell_1 \vee \dots \vee \ell_k \vee x = y, m_1 \vee \dots \vee m_n[z]}{\text{SUST}(\theta, \ell_1 \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_n[y])}$$

A diferencia de la demodulación, la paramodulación nos lleva a un procedimiento de inferencia completo para la lógica de primer orden con igualdad.

Un tercer enfoque maneja el razonamiento con igualdad insertándolo por entero en un algoritmo de unificación extendida. Es decir, los términos son unificables si son *probablemente* iguales bajo alguna sustitución, donde «probablemente» indica algún tipo de razonamiento con igualdad. Por ejemplo, los términos $1 + 2$ y $2 + 1$ normalmente no son unificables, pero un algoritmo de unificación que supiera que $x + y = y + x$ podría unificarlos con la sustitución vacía. Este tipo de **unificación de ecuaciones** se puede realizar mediante algoritmos eficientes diseñados para los axiomas concretos que se vayan a utilizar (comutatividad, asociatividad, etc.); más que a través de una inferencia explícita con dichos axiomas. Los demostradores de teoremas que utilizan esta técnica están estrechamente relacionados con los sistemas de programación lógica que se describieron en el Apartado 9.4.

UNIFICACIÓN DE ECUACIONES

Estrategias de resolución

Sabemos que las aplicaciones repetidas de la regla de inferencia de resolución finalmente encontrarán una demostración si ésta existe. En esta subsección, examinaremos las estrategias que ayudan a encontrar demostraciones *eficientemente*.

Resolución unitaria

Esta estrategia prefiere realizar las resoluciones cuando una de las sentencias es un único literal (también conocida como **cláusula unitaria**). La idea en que se basa la estrategia es que estamos intentando producir una cláusula vacía, así que podría ser bueno preferir inferencias que produzcan cláusulas más cortas. Resolviendo una sentencia unitaria (como P) con otra sentencia (como $\neg P \vee \neg Q \vee R$) siempre obtenemos una cláusula (en este caso $\neg Q \vee R$) que es más corta que la anterior. Cuando se intentó por primera vez la estrategia de preferencia unitaria en 1964 se obtuvo a una aceleración considerable del proceso, y se hizo viable demostrar teoremas que no se podían manejar sin dicha preferencia. Sin embargo, la preferencia unitaria por sí misma no reduce lo suficiente el factor de ramificación en problemas de tamaño mediano para hacerlos resolubles mediante la resolución. No obstante, es una heurística útil que se puede combinar con otras estrategias.

RESOLUCIÓN UNITARIA

La **resolución unitaria** es una variante restringida de la resolución, en la que cada paso de resolución debe involucrar a una cláusula unitaria. En general, la resolución unitaria es incompleta, pero es completa con las bases de conocimiento de Horn. Las demostraciones mediante resolución unitaria aplicadas a las bases de conocimiento de Horn se parecen bastante al encadenamiento hacia delante.

Resolución mediante conjunto soporte

CONJUNTO SOPORTE

Las preferencias que intentan antes ciertas resoluciones son de alguna ayuda, pero por lo general es más efectivo intentar eliminar completamente algunas resoluciones potenciales. La estrategia del conjunto soporte hace justamente eso. Comienza por identificar un subconjunto de sentencias denominado **conjunto soporte**. Cada resolución combina una sentencia del conjunto soporte con otra sentencia y añade el resolvente al conjunto soporte. Si el conjunto soporte es relativamente pequeño respecto a la base de conocimiento, el espacio de búsqueda se reduce drásticamente.

Tenemos que ser cautos con este enfoque, porque una elección errónea del conjunto soporte hará que el algoritmo sea incompleto. Sin embargo, si elegimos el conjunto soporte S de tal manera que el resto de sentencias sean conjuntamente *satisfacibles*, entonces la resolución mediante el conjunto soporte será completa. Un enfoque habitual es utilizar una petición negada como el conjunto soporte, bajo la asunción que la base de conocimiento original es consistente. (Después de todo, si es consistente, entonces el hecho de la petición que se sigue es vacuo.) La estrategia del conjunto soporte tiene la ventaja adicional de generar árboles de demostración que a menudo son fáciles de entender por las personas, ya que éstas están orientadas al objetivo.

Resolución lineal

RESOLUCIÓN DE ENTRADA

RESOLUCIÓN LINEAL

En la estrategia de **resolución de entrada** cada resolución combina una de las sentencias de entrada (de la BC o de la petición) con alguna otra sentencia. La demostración de la Figura 9.11 sólo utiliza resoluciones de entrada y tiene la forma característica de una «espina de pez» con sentencias simples que se van combinando en la columna. En las bases de conocimiento de Horn, el Modus Ponens es un tipo de estrategia de resolución de entrada, porque combina una implicación de la BC original con algunas otras sentencias. Por lo tanto, no nos sorprende que la resolución de entrada sea completa con las bases de conocimiento que están en forma de Horn, pero es incompleta en el caso general. La estrategia de **resolución lineal** es una generalización leve que permite que P y Q se resuelvan juntas aunque P esté en la BC original o P sea un parente de Q en el árbol de demostración. La resolución lineal es completa.

Subsunción

SUBSUNCIÓN

El método de **subsunción** elimina todas las sentencias que están subsumidas por (por ejemplo, son más específicas que) una sentencia existente en la BC. Por ejemplo, si $P(x)$ está en la BC, entonces no tiene sentido añadir $P(A)$ y aún menos sentido tiene añadir $P(A) \vee Q(B)$. La subsunción ayuda a mantener la BC con un tamaño relativamente pequeño, y de esta manera nos ayuda a establecer que el espacio de búsqueda sea, también, relativamente pequeño.

Demostradores de teoremas

Los demostradores de teoremas (también conocidos como razonadores automáticos) se diferencian de la programación lógica en dos cosas. Primero, muchos lenguajes de programación lógica sólo manejan cláusulas de Horn, mientras que los demostradores de teoremas aceptan la lógica de primer orden en su totalidad. Segundo, los programas en Prolog entrelazan la lógica y el control. La elección del programador sobre $A :- B, C$ en vez de $A :- C, B$ afecta a la ejecución del programa. En muchos demostradores de teoremas, la forma sintáctica escogida para las sentencias no afecta a los resultados. Los demostradores de teoremas todavía necesitan la información de control para poder operar eficientemente, pero por lo general, dicha información se mantiene separada de la base de conocimiento, en vez de formar parte de la propia representación del conocimiento. Gran parte de la investigación en demostradores de teoremas se basa en encontrar estrategias de control que sean de utilidad general al mismo tiempo que aumenten la velocidad.

Diseño de un demostrador de teoremas

En esta sección describiremos el demostrador de teoremas OTTER (*Organized Techniques for Theorem-proving and Effective Research*) (McCune, 1992), con una atención especial en su estrategia de control. Al preparar un problema para OTTER el usuario debe dividir el conocimiento en cuatro partes:

- Un conjunto de cláusulas, conocido como el **conjunto soporte** (o *cs*), que define los hechos relevantes acerca del problema. Cada paso de la resolución resuelve un miembro del conjunto soporte frente a otro axioma, de esta manera la búsqueda se focaliza en el conjunto soporte.
- Un conjunto de **axiomas utilizables** que son externos al conjunto soporte. Éstos proporcionan el conocimiento base acerca del dominio del problema. El límite entre lo que forma parte del problema (y por lo tanto del *cs*) y lo que es conocimiento base (y por lo tanto de los axiomas utilizables) se deja a elección del usuario.
- Un conjunto de ecuaciones conocido como **rescritores** o **demoduladores**. Aunque los demoduladores son ecuaciones, siempre se aplican en la dirección izquierda a derecha. Por lo tanto, definen la forma canónica en la que todos los términos serán simplificados. Por ejemplo, el demodulador $x + 0 = x$ dice que cada término del tipo $x + 0$ debe reemplazarse por el término x .
- Un conjunto de parámetros y cláusulas que definen la estrategia de control. En concreto, el usuario especifica una función heurística para controlar la búsqueda, y una función de filtrado para eliminar aquellos subobjetivos que no interesan.

OTTER trabaja resolviendo, de forma continuada, un elemento del conjunto soporte frente a uno de los axiomas utilizables. A diferencia del Prolog, utiliza un tipo de búsqueda de primero el mejor. Su función heurística mide el «peso» de cada cláusula, donde las más ligeras tienen preferencia. La elección exacta de la heurística depende del usuario,

pero por lo general, el peso de una cláusula debería ser correlacionada con su tamaño y dificultad. Las cláusulas unitarias se tratan como ligeras; de este modo, la búsqueda se puede ver como una generalización de la estrategia de preferencia unitaria. En cada paso, OTTER mueve la cláusula «más ligera» del conjunto soporte a una lista, a la que llamamos lista utilizable, y añade a dicha lista las consecuencias inmediatas de resolver la cláusula con los elementos de la lista. OTTER finaliza cuando ha encontrado una refutación o no hay más cláusulas en el conjunto soporte. En la Figura 9.14 se muestra el algoritmo en más detalle.

```

procedimiento OTTER(cs, utilizable)
  entradas: cs, el conjunto soporte: cláusulas que definen el problema (una variable global)
  utilizable, conocimiento base potencialmente relevante para el problema

  repetir
    cláusula  $\leftarrow$  el miembro más ligero de cs
    mover cláusula de cs a utilizable
    PROCESA(INFIERE(cláusula, utilizable), cs)
  hasta cs = [] o se ha encontrado una refutación



---


función INFIERE(cláusula, utilizable) devuelve cláusulas

  resuelve cláusula con cada miembro de utilizable
  devolver las cláusulas resultantes después de aplicar FILTRO



---


procedimiento PROCESA(cláusulas, cs)
  para cada cláusula en cláusulas hacer
    cláusula  $\leftarrow$  SIMPLIFICA(cláusula)
    fusiona literales idénticos
    descarta la cláusula si es una tautología
    cs  $\leftarrow$  [cláusula - cs]
    si cláusula no tiene literales entonces se ha encontrado una refutación
    si cláusula tiene un literal entonces averigua si hay refutación unitaria

```

Figura 9.14 Esquema del demostrador de teoremas OTTER. El control heurístico se aplica en la selección de la cláusula «más ligera» y en la función FILTRO, que elimina las cláusulas que no se consideran interesantes.

Ampliar el Prolog

Una alternativa en la construcción de un demostrador de teoremas es comenzar con un compilador de Prolog y ampliarlo para conseguir un razonador sólido y completo para toda la lógica de primer orden. Este era el enfoque que se tomó en el *Prolog Technology Theorem Prover*, o PTTP (Stickel, 1988). El PTTP incluye cinco cam-

bios significativos aplicados al Prolog para conseguir completitud y una mayor expresividad:

- La comprobación de ocurrencias se introduce en la rutina de unificación para hacerla sólida.
- La búsqueda de primero en profundidad se reemplaza por una búsqueda de profundidad iterativa. Esto hace que la estrategia de búsqueda sea completa y que tome sólo un factor constante de tiempo.
- Se permite utilizar literales negados (como $\neg P(x)$). En la implementación, hay dos rutinas separadas, una que intenta demostrar P y otra que intenta demostrar $\neg P$.
- Una cláusula con n átomos se almacena como n reglas diferentes. Por ejemplo, $A \Leftarrow B \wedge C$ también podría almacenarse como $\neg B \Leftarrow C \wedge \neg A$ y como $\neg C \Leftarrow B \wedge \neg A$. Esta técnica, conocida como **bloqueo**, significa que el objetivo actual sólo necesita ser unificado con la cabeza de cada cláusula, y aún permite un manejo adecuado de la negación.
- La inferencia se hace completa (aun con cláusulas de Horn) añadiendo la regla de resolución de entrada lineal. Si el objetivo actual se unifica con la negación de uno de los objetivos de la pila, entonces el objetivo se puede considerar resuelto. Ésta es una forma de razonamiento por contradicción. Suponga que el objetivo original es P y esto se reduce a una serie de inferencia con el objetivo $\neg P$. Esto establece que $\neg P \Rightarrow P$, que es equivalente lógicamente a P .

A pesar de estos cambios, PTTP mantiene las características que hacen que el Prolog sea rápido. La unificación todavía se hace modificando las variables directamente, desligándolas mediante el relajamiento de la pila en la vuelta hacia atrás (*backtracking*). La estrategia de búsqueda todavía se basa en la resolución de entrada, entonces cada resolución se realiza a través de una de las cláusulas del enunciado original del problema (más que con una cláusula derivada). Esto hace que sea viable compilar todas las cláusulas del enunciado original del problema.

El principal inconveniente del PTTP es que el usuario tiene que renunciar a todo el control sobre la búsqueda de las soluciones. Cada regla de inferencia se utiliza por el sistema tanto en su forma original como en su forma contrapositiva. Esto nos puede llevar a búsquedas nada intuitivas. Por ejemplo, considere la regla

$$(f(x, y) = f(a, b)) \Leftarrow (x = a) \wedge (y = b)$$

Como regla de Prolog, es algo razonable intentar demostrar que los dos términos de f son iguales. Pero el PTTP también generaría su contrapositiva:

$$(x \neq a) \Leftarrow (f(x, y) \neq f(a, b)) \wedge (y = b)$$

Lo que parece un derroche el intentar demostrar que los dos términos x y a son diferentes.

Demostradores de teoremas como asistentes

Hasta ahora, hemos enfocado los sistemas de razonamiento como agentes independientes que tienen que tomar decisiones y actuar por sí mismos. Otro uso de los demostradores de teoremas es como asistentes, proporcionando consejo a, por ejemplo, un

COMPROBADOR DE DEMOSTRACIONES

RAZONADOR

ÁLGEBRA DE ROBBINS

VERIFICACIÓN

SÍNTESIS

matemático. De este modo, un matemático actúa como un supervisor, seleccionando la estrategia para determinar qué hacer seguidamente y pidiendo al demostrador de teoremas que rellene los detalles. Este mecanismo alivia, en parte, el problema de la semi-decidibilidad, porque el supervisor puede cancelar una petición e intentar otro enfoque si la petición está consumiendo mucho tiempo. Un demostrador de teoremas también puede actuar como un **comprobador de demostraciones**, donde la demostración la da una persona en forma de un conjunto de pasos algo extenso, y las inferencias concretas que se necesitan para demostrar que cada paso es sólido las realiza el sistema.

Un **razonador Socrático** es un demostrador de teoremas cuya función PREGUNTA es incompleta, pero que siempre puede llegar a una solución si le han dado el orden correcto de peticiones. De este modo, los razonadores Socráticos son buenos asistentes, dado que haya un supervisor que envíe las series de llamadas correctas a PREGUNTA. ONTIC (McAllester, 1989) es un sistema de razonamiento Socrático para matemáticos.

Usos prácticos de los demostradores de teoremas

Los demostradores de teoremas han surgido con resultados matemáticos muy originales. El programa SAM (*Semi-Automated Mathematics*) fue el primero en demostrar un lema de la teoría de retículos (Guard *et al.*, 1969). El programa AURA también ha respondido a preguntas abiertas en diversas áreas de las matemáticas (Wos y Winker, 1983), el demostrador de teoremas Boyer-Moore (Boyer y Moore, 1979) se ha utilizado y ampliado durante muchos años y se utilizó por Natarajan Shankar para dar la primera demostración formal rigurosa sobre el Teorema de la Incompletitud de Gödel (Shankar, 1986). El programa OTTER es uno de los mejores demostradores de teoremas; se ha utilizado para resolver diversas preguntas abiertas sobre lógica combinatoria. La más famosa es la que concierne al **álgebra de Robbins**. En 1933, Herbert Robbins propuso un conjunto sencillo de axiomas que intentaban definir el álgebra Booleana, pero no se pudo encontrar ninguna prueba que los demostrara (a pesar del trabajo serio de diversos matemáticos incluyendo al propio Alfred Tarski). En octubre de 1996, después de ocho días de cálculo, el EQP (una variante del OTTER) encontró una demostración (McCune, 1997).

Los demostradores de teoremas se pueden aplicar a los problemas relacionados con la **verificación y síntesis** del *hardware* y *software*, porque ambos dominios pueden disponer de axiomatizaciones correctas. De este modo, la investigación en la demostración de teoremas se lleva a cabo en los campos del diseño del *hardware*, los lenguajes de programación, y la ingeniería del *software*, no sólo en la IA. En el caso del *software*, los axiomas definen las propiedades de cada elemento sintáctico del lenguaje de programación. (Razonar acerca de los programas es bastante parecido a razonar acerca de las acciones en el cálculo de situaciones.) Un algoritmo se verifica demostrando que sus salidas respetan las especificaciones de todas las entradas. El algoritmo RSA de encriptación de clave pública y el algoritmo de emparejamiento de texto de Boyer-Moore han sido verificados de este modo (Boyer y Moore, 1984). En el caso del *hardware*, los axiomas describen las interacciones entre las señales y los elementos del circuito. (Véase Capítulo 8 para un ejemplo.) El diseño de un sumador de 16 bits ha sido verificado por el AURA (Wojcik, 1983). Los razonadores especialmente diseñados para la verificación

SÍNTESIS DEDUCTIVA

han sido capaces de verificar CPUs enteras, incluyendo sus propiedades temporales (Sri-
vas y Bickford, 1990).

La síntesis formal de algoritmos fue uno de los primeros usos de los demostradores de teoremas, tal como perfila Cordell Green (1969a), quien trabajó sobre las ideas iniciales de Simon (1963). La idea es demostrar un teorema de manera que «exista un programa p que satisfaga ciertas especificaciones». Si la demostración está restringida a ser constructiva el programa se puede obtener. Aunque, tal como se le llama, la **síntesis deductiva** no ha sido automatizada totalmente, y aún no ha conseguido ser viable para la programación de propósito general, la síntesis deductiva guiada ha tenido éxito en el diseño de diversos algoritmos bastante sofisticados y novedosos. Los programas de síntesis para propósito específico también son un área activa de investigación. En el área de síntesis del *hardware*, el demostrador de teoremas AURA se ha aplicado en el diseño de circuitos que son más compactos que cualquier diseño anterior (Wojciechowski y Wojcik, 1983). Para muchos diseños de circuitos, la lógica proposicional es suficiente porque el conjunto de proposiciones de interés está fijado por el conjunto de elementos del circuito. La aplicación de la inferencia proposicional en la síntesis del *hardware* es actualmente una técnica estándar que tiene muchas utilidades a gran escala (véase, por ejemplo, Nowick *et al.*, (1993)).

Actualmente, estas mismas técnicas se están comenzando a aplicar en la verificación del *software*, por sistemas como el comprobador de modelos SPIN (Holzmann, 1997). Por ejemplo, el programa de control espacial del Agente Remoto fue verificado antes y después del vuelo (Havelund *et al.*, 2000).

9.6 Resumen

Hemos presentado un análisis de la inferencia lógica en lógica de primer orden, y un número de algoritmos para realizarla.

- Un primer enfoque utiliza reglas de inferencia para instanciar los cuantificadores y proposicionalizar el problema de inferencia. Por lo general este enfoque es muy lento.
- El uso de la **unificación** para obtener las sustituciones adecuadas de las variables elimina el paso de instanciación en las demostraciones de primer orden, haciendo que el proceso sea mucho más eficiente.
- Una versión elevada del **Modus Ponens** utiliza la unificación para proporcionar una regla de inferencia natural y potente, el **Modus Ponens Generalizado**. Los algoritmos de **encadenamiento hacia delante** y de **encadenamiento hacia atrás** aplican esta regla a conjuntos de cláusulas positivas.
- El Modus Ponens Generalizado es completo para las cláusulas positivas, aunque el problema de la implicación es **semidecidible**. Para los programas **Datalog** que tienen cláusulas positivas con funciones libres, la implicación es decidible.
- El encadenamiento hacia delante se utiliza en las **bases de datos deductivas**, donde se puede combinar con las operaciones de las bases de datos relacionales. También se utiliza en los **sistemas de producción**, que pueden hacer actualizaciones eficientes en conjuntos de reglas muy grandes.

- El encadenamiento hacia delante es completo en los programas Datalog y corre en tiempo polinómico.
- El encadenamiento hacia atrás se utiliza en los **sistemas de programación lógica** como el **Prolog**, que emplea una sofisticada tecnología de compilación para proporcionar una inferencia muy rápida.
- El encadenamiento hacia atrás sufre de inferencias redundantes y bucles infinitos; esto se puede aliviar mediante la **memorización**.
- La regla de inferencia de la **resolución** generalizada proporciona un sistema de demostración completo en lógica de primer orden, utilizando bases de conocimiento en forma normal conjuntiva.
- Existen diversas estrategias para reducir el espacio de búsqueda de un sistema de resolución sin comprometer la completitud. Los demostradores de teoremas eficientes, basados en la resolución, se han utilizado para proporcionar teoremas matemáticos de interés y para verificar y diseñar *hardware* y *software*.



SILOGISMO

NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

La inferencia lógica fue estudiada extensivamente por los matemáticos griegos. El tipo de inferencia estudiado más cuidadosamente por Aristóteles fue el **silogismo**, que es un tipo de regla de inferencia. Los silogismos de Aristóteles incluían elementos de la lógica de primer orden, como la cuantificación, pero estaba restringida a los predicados unarios. Los silogismos estaban categorizados mediante las «figuras» y los «modos», dependiendo del orden de los términos (a los que deberíamos llamar predicados) en las sentencias, el grado de generalidad (a lo que hoy en día deberíamos interpretar a través de los cuantificadores) aplicado a cada término, y si cada término estaba negado. El silogismo más importante es el primer modo de la primera figura:

Todo *S* es *M*.
Todo *M* es *P*.
Por lo tanto, todo *S* es *P*.

Aristóteles intentó demostrar la validez de otros silogismos mediante su «reducción» a aquellos de la primera figura. Era mucho menos preciso en describir lo que esta «reducción» debe implicar que en la caracterización de las propias figuras y modos silogísticos.

Gottlob Frege, quien desarrolló totalmente la lógica de primer orden en 1879, basó su sistema de inferencia en una gran colección de esquemas lógicamente válidos junto a una sola regla de inferencia, el Modus Ponens. Frege tomó ventaja del hecho de que el efecto de una regla de inferencia de la forma «De *P*, inferir *Q*» se puede simular aplicando el Modus Ponens a *P* junto a un esquema válido lógicamente como $P \Rightarrow Q$. Este estilo «axiomático» de exposición, utilizando el Modus Ponens junto a un número de esquemas lógicamente válidos, fue empleado por un número de lógicos después de Frege; y fue utilizado, de la forma más notable, en el *Principia Mathematica* (Whitehead y Russell, 1910).

Las reglas de inferencia, a diferencia de los esquemas axiomáticos, fueron el foco principal de la **deducción natural**, introducida por Gerhard Gentzen (1934) y por Sta-

nislaw Jaskowski (1934). A la deducción natural se le denomina «natural» porque no necesita la conversión a la (ilegible) forma normal, y porque sus reglas de inferencia intentan parecer cercanas a las personas. Prawitz (1965) proporciona un tratamiento extenso acerca de la deducción natural. Gallier (1986) utiliza el enfoque de Gentzen para exponer los cimientos teóricos de la deducción automática.

La invención de la forma clausular fue un paso crucial en el desarrollo profundo del análisis matemático en lógica de primer orden. Whitehead y Russell (1910) expusieron las denominadas *reglas de paso* (el término realmente pertenece Herbrand (1930) que se utilizan para mover los cuantificadores al frente de las fórmulas. Las constantes de Skolem y las funciones de Skolem fueron introducidas lo suficientemente por Thoralf Skolem (1920). El procedimiento general de skolemización se dio por Skolem (1928), junto al importante concepto de universo de Herbrand.

El teorema de Herbrand, que recibe el nombre del lógico francés Jacques Herbrand (1930), ha jugado un papel vital en el desarrollo de los métodos de razonamiento automático, tanto antes como después de la introducción de la resolución de Robinson. Esto se refleja en nuestra referencia al «universo de Herbrand» más que al «universo de Skolem», aun pensando que fue realmente Skolem el que inventó el concepto. Herbrand también puede ser considerado como el inventor de la unificación. Gödel (1930) trabajó con base en las ideas de Skolem y Herbrand para demostrar que la lógica de primer orden tiene un procedimiento de demostración completo. Alan Turing (1936) y Alonzo Church (1936) demostraron, al mismo tiempo y utilizando diferentes demostraciones, que la validez en lógica de primer orden no era decidible. El excelente texto de Enderton (1972) explica todos estos resultados de un modo riguroso aunque de manera bastante comprensible.

Aunque McCarthy (1958) había sugerido el uso de la lógica de primer orden para la representación y el razonamiento en la IA, el primero de este tipo de sistemas fue desarrollado por los lógicos interesados en la demostración de teoremas matemáticos. Fue Abraham Robinson quien propuso el uso de la proposicionalización y del teorema de Herbrand, y Gilmore (1960) fue quien escribió el primer programa basado en este enfoque. Davis y Putnam (1960) utilizaron la forma clausular y desarrollaron un programa que intentaba encontrar refutaciones mediante la sustitución de las variables de los miembros del universo de Herbrand para generar cláusulas base y entonces averiguar si había inconsistencias entre las cláusulas base. Prawitz (1960) desarrolló la idea clave para permitir que el análisis de la inconsistencia proposicional dirija el proceso de búsqueda, y generar términos del universo de Herbrand sólo cuando es necesario hacerlo y así poder establecer la inconsistencia proposicional. Después de un largo desarrollo hecho por otros investigadores, esta idea le llevó a J. A. Robinson (no hay relación) a desarrollar el método de la resolución (Robinson, 1965). El denominado método inverso desarrollado por el investigador soviético S. Maslov (1964, 1967), más o menos en la misma época, y basado en unos principios algo diferentes, ofrece ventajas computacionales similares sobre la proposicionalización. El **método de conexión** de Wolfgang Bibel (1981) se puede ver como una extensión de este enfoque.

Después del desarrollo de la resolución, el trabajo sobre la inferencia en lógica de primer orden se produjo en varias direcciones diferentes. En la IA, la resolución se adoptó en los sistemas de respuesta a peticiones por Cordell Green y Bertram Raphael (1968).

Un enfoque algo menos formal se tomó por Carl Hewitt (1969). Su lenguaje PLANNER, aunque nunca implementado en su totalidad, fue un precursor de la programación lógica e incluía directivas para el encadenamiento hacia delante y hacia atrás, y para la negación como fallo. Un subconjunto conocido como MICRO-PLANNER (Sussman y Winograd, 1970) se implementó y utilizó en el sistema de comprensión del lenguaje natural SHRDLU (Winograd, 1972). Las implementaciones más recientes en IA ponen una buena cantidad de esfuerzo en las estructuras de datos que permitirían una recuperación eficiente de los hechos; este trabajo está cubierto en los textos de programación en IA (Charniak *et al.*, 1987; Norvig, 1992; Forbus y de Kleer, 1993).

A principios de los 70, el **encadenamiento hacia delante** estaba bien establecido en la IA como una alternativa comprensible a la resolución. Se utilizó en una amplia variedad de sistemas, desde el demostrador de teoremas de geometría de Nevins (Nevins, 1975) al sistema experto R1 para la configuración de Vax (McDermott, 1982). Las aplicaciones en IA por lo general manejaban grandes cantidades de reglas, así que era importante desarrollar una tecnología eficiente para el emparejamiento de reglas, en concreto para las actualizaciones incrementales. La tecnología para los **sistemas de producción** se desarrolló para apoyar este tipo de aplicaciones. El lenguaje de sistema de producción OPS-5 (Forgy, 1981; Brownston *et al.*, 1985) se utilizó para el R1 y para la arquitectura cognitiva SOAR (Laird *et al.*, 1987). El OPS-5 incorporaba el proceso de emparejamiento rete (Forgy, 1982). El SOAR, que genera reglas nuevas para depositar los resultados de cálculos previos, puede crear conjuntos de reglas muy grandes, acerca de 1.000.000 de reglas en el caso del sistema TACAIR-SOAR para controlar un avión de caza simulado (Jones *et al.*, 1998). CLIPS (Wygant, 1989) fue un lenguaje basado en C para sistemas de producción desarrollado en la NASA que permitió una mejor integración con otros paquetes de *software*, *hardware*, y sistemas de sensores, y se utilizó en la automatización de naves espaciales y diversas aplicaciones militares.

El área de investigación conocida como **bases de datos deductivas** también ha contribuido bastante a nuestro entendimiento de la inferencia hacia delante. Comenzó en una sesión de trabajo en Toulouse en 1977, organizada por Jack Minker, en la que juntó a expertos en inferencia lógica y en sistemas de bases de datos (Gallaire y Minker, 1978). Una revisión histórica reciente (Ramakrishnan y Ullman, 1995) dice, «los sistemas de [bases de datos] deductivas son un intento de adaptar el Prolog, que tiene un punto de vista del mundo de “datos pequeños”, a un mundo de “datos grandes”». De este modo, pretende fundir la tecnología de bases de datos relacionales, que está diseñada para recuperar grandes *conjuntos* de hechos, con la tecnología de inferencia del Prolog, que por lo general recupera un hecho cada vez. Algunos de los textos sobre bases de datos deductivas son el de Ullman (1989) y el de Ceri *et al.*, (1990).

Los trabajos influyentes de Chandra y Harel (1980) y el de Ullman (1985) condujeron a la adopción del Datalog como un lenguaje estándar para las bases de datos deductivas. La inferencia «de abajo arriba», o el encadenamiento hacia delante, también se convirtieron en el estándar, en parte porque evitan los problemas con la no terminación y el cálculo redundante que se presentan en el encadenamiento hacia atrás, y en parte porque tiene una implementación mucho más natural en términos de las operaciones de las bases de datos relacionales. El desarrollo de la técnica de los **conjuntos mágicos**

para la reescritura de las reglas de Bancilhon *et al.*, (1986) le permitió al encadenamiento hacia delante coger prestado la ventaja de la orientación al objetivo del encadenamiento hacia atrás. Igualándolo a la carrera armamentística, los métodos de programación lógica tableados (véase Apartado 9.6) permitieron la ventaja de la programación lógica desde el encadenamiento hacia delante.

Gran parte de nuestro entendimiento de la complejidad de la inferencia lógica nos ha venido de la comunidad de las bases de datos deductivas. Chandra y Merlin (1977) fueron los primeros en demostrar que el emparejamiento de una única regla no recursiva (una **petición conjuntiva** en la terminología de las bases de datos) puede ser NP-duro. Kuper y Vardi (1993) propusieron la **complejidad de datos** (es decir, la complejidad como una función del tamaño de la base de datos, tratando el tamaño de la regla como constante) como una medida útil en la respuesta a una petición. Gottlob *et al.*, (1999b) discuten la conexión entre las peticiones conjuntivas y la satisfacción de restricciones, mostrando cómo la descomposición de hiperárboles puede optimizar el proceso de emparejamiento.

Tal como hemos comentado anteriormente, el **encadenamiento hacia atrás** en la inferencia lógica apareció en el lenguaje PLANNER de Hewitt (1969). La programación lógica *per se* se desarrolló independientemente a este trabajo. Una forma restringida de la resolución lineal, denominada **resolución-SL** fue desarrollada por Kowalski y Kuehner (1971), construida a partir de la técnica de **eliminación de modelos** de Loveland (1968); cuando se aplicó a las cláusulas positivas, pasó a ser la **resolución-SLD**, que se presta a la interpretación de las cláusulas positivas como programas (Kowalski, 1974, 1979a, 1979b). Mientras tanto, en 1972, el investigador francés Alain Colmerauer había desarrollado e implementado **Prolog** con el propósito de analizar el lenguaje natural; las cláusulas de Prolog se pensaron inicialmente como reglas de gramáticas libres de contexto (Roussel, 1975; Colmerauer *et al.*, 1973). Gran parte de la base teórica de la programación lógica se desarrolló por Kowalski, trabajando con Colmerauer. La definición semántica mediante máximos puntos fijos es de Van Emden y Kowalski (1976). Kowalski (1988) y Cohen (1988) nos proporcionan unos buenos análisis históricos de los orígenes del Prolog. *Foundations on Logic Programming* (Lloyd, 1987) es un análisis teórico de los fundamentos del Prolog y otros lenguajes de programación lógica.

Los compiladores eficientes para Prolog están basados, por lo general, en el modelo de cálculo de la *Warren Abstract Machine* (WAM) de David H. D. Warren (1983). Van Roy (1990) que mostró la aplicación de técnicas de compilación adicionales, como el tipo de inferencia, hizo programas en Prolog que eran competitivos con programas en C en lo referente a la velocidad. El proyecto *Japanese Fifth Generation*, un esfuerzo de investigación de 10 años, que comenzó en 1982, estaba totalmente basado en el Prolog para desarrollar sistemas inteligentes.

Los métodos para evitar los bucles innecesarios en los programas lógicos recursivos fueron desarrollados independientemente por Smith *et al.*, (1986) y por Tamaki y Sato (1986). El último trabajo también incluía la memorización en los programas lógicos, un método desarrollado extensivamente como la **programación lógica tableada** por David S. Warren. Swift y Warren (1994) muestran cómo ampliar la WAM para manejar el tableado, permitiendo a los programas en Datalog ejecutarse en un orden de magnitud

PETICIÓN CONJUNTIVA

COMPLEJIDAD DE DATOS

RESOLUCIÓN-SL

RESOLUCIÓN-SLD

más rápida que los sistemas de bases de datos deductivas mediante el encadenamiento hacia delante.

El trabajo teórico inicial sobre la programación lógica con restricciones fue de Jaffar y Lassez (1987). Jaffar *et al.*, (1992a) desarrollaron el sistema CLP(R) para manejar restricciones con valores reales. Jaffar *et al.*, (1992b) generalizaron la WAM para generar la CLAM (*Constraint Logic Abstract Machine*) para especificar las implementaciones del CLP. Ait-Kaci y Podelski (1993) describen un lenguaje sofisticado denominado LIFE, que combina CLP con la programación funcional y con el razonamiento sobre herencia. Kohn (1991) describe un proyecto ambicioso para utilizar la programación lógica con restricciones como el fundamento de una arquitectura de control en tiempo real, con aplicaciones de pilotos totalmente automatizados.

Hay un buen número de libros acerca de la programación lógica y el Prolog. *Logic for Problem Solving* (Kowalski, 1979b) es uno de los textos iniciales sobre la programación lógica en general. Entre los textos sobre el Prolog encontramos el de Clocksin y Mellish (1994), Shoham (1994) y Bratko (2001). Marriott y Stuckey (1998) proporcionan una excelente cobertura de la PLR. Hasta su cierre en 2000, la *Journal of Logic Programming* era la revista de partida; ahora ha sido reemplazada por *Theory and Practice of Logic Programming*. Entre las conferencias acerca de la programación lógica encontramos la *International Conference on Logic Programming* (ICLP) y el *International Logic Programming Symposium* (ILPS).

La investigación en **demostración de teoremas matemáticos** comenzó aún antes de los primeros sistemas de primer orden completos. El demostrador de teoremas de geometría de Herbert Gelernter (Gelernter, 1959) utilizaba los métodos de búsqueda heurística combinados con los diagramas de poda de subobjetivos falsos y era capaz de demostrar algunos resultados bastante intrincados de la geometría Euclidiana. Sin embargo, desde entonces, no ha habido mucha interacción entre la demostración de teoremas y la IA.

Los trabajos iniciales se concentraban en la completitud. Siguiendo al trabajo inicial de Robinson, las reglas de demodulación y paramodulación para el razonamiento con igualdades fueron introducidas por Wos *et al.*, (1967) y Wos y Robinson (1968), respectivamente. Estas reglas también fueron desarrolladas independientemente en el contexto de los sistemas de reescritura de términos (Knuth y Bendix, 1970). La incorporación del razonamiento con igualdad en el algoritmo de resolución se debe a Gordon Plotkin (1972); también era una característica del QLISP (Sacerdoti *et al.*, 1976). Jouannaud y Kirchner (1991) hacen un repaso a la unificación ecuacional desde la perspectiva de la reescritura de términos. Los algoritmos eficientes de la unificación estándar fueron desarrollados por Martelli y Montanari (1976) y Paterson y Wegman (1978).

Junto al razonamiento con igualdad, los demostradores de teoremas han incorporado una variedad de procedimientos de propósito específico. Nelson y Oppen (1979) propusieron un esquema muy influyente para la integración de dichos procedimientos en un sistema de razonamiento general, entre otros métodos tenemos la «resolución de teoría» de Stickel (1985) y las «relaciones especiales» de Manna y Waldinger (1986).

Se han propuesto un buen número de estrategias para la resolución, comenzando con la estrategia de preferencia unitaria (Wos *et al.*, 1964). La estrategia del conjunto soporte

fue propuesta por Wos *et al.*, (1965), para proporcionar cierto grado de orientación al objetivo en la resolución. La resolución lineal apareció por primera vez en Loveland (1970). Genesereth y Nilsson (1987, Capítulo 5) proporcionan un breve, aunque cuidadoso, análisis de una gran variedad de estrategias de control.

Guard *et al.*, (1969) describen el demostrador de teoremas SAM inicial, que ayudó a resolver un problema abierto acerca de la teoría de retículos. Wos y Winker (1983) dan un repaso a las contribuciones del demostrador de teoremas AURA a través de la resolución de problemas abiertos en diversas áreas de las matemáticas y la lógica. McCune (1992) continúa en esta dirección, hablando de las realizaciones del sucesor del AURA, el OTTER, en la resolución de problemas abiertos. Weidenbach (2001) describe el SPASS, uno de los demostradores de teoremas actuales más potentes. A *Computational Logic* (Boyer y Moore, 1979) es la referencia básica del demostrador de teoremas de Boyer-Moore. Stickel (1988) cubre la *Prolog Technology Theorem Prover* (PTTP), que combina las ventajas de la compilación del Prolog con la completitud de la eliminación de modelos (Loveland, 1968). SETHEO (Letz *et al.*, 1992) es otro demostrador de teoremas ampliamente utilizado basado en este enfoque; puede realizar varios millones de inferencias por segundo en estaciones del modelo-2000. LEANTAP (Beckert y Posegga, 1995) es un demostrador de teoremas eficiente implementado tan sólo en 25 líneas de Prolog.

El trabajo inicial sobre síntesis automática de programas fue realizado por Simon (1963), Green (1969a) y Manna y Waldinger (1971). El sistema transformacional de Burstall y Darlington (1977) utilizaba el razonamiento ecuacional para la síntesis de programas recursivos. KIDS (Smith, 1990, 1996) es uno de los sistemas modernos más potentes; opera como un asistente experto. Manna y Waldinger (1992) dan una introducción de tutorial al estado del arte actual, con énfasis en su propio enfoque deductivo. *Automating Software Design* (Lowry y McCartney, 1991) recopila un buen número de trabajos en el área. El uso de la lógica en el diseño del *hardware* se trata en Kern y Greenstreet (1999); Clarke *et al.*, (1999) cubre la comprobación de modelos para la verificación del *hardware*.

Computability and Logic (Boolos y Jeffrey, 1989) es una buena referencia sobre la completitud y la indecidibilidad. Muchos trabajos iniciales sobre lógica matemática se encuentran en *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). La revista de partida en este campo de la lógica matemática pura (opuesta a la deducción automática) es *The Journal of Symbolic Logic*. Entre los libros engranados en la deducción automática encontramos el clásico *Symbolic Logic and Mechanical Theorem Proving* (Chang y Lee, 1973), así como otros trabajos más recientes como el de Wos *et al.*, (1992), Bibel (1993) y Kaufmann *et al.*, (2000). La antología *Automation of Reasoning* (Siekmann y Wrightson, 1983) dispone de trabajos iniciales muy importantes sobre la deducción automática. Otras revisiones históricas han sido escritas por Loveland (1984) y Bundy (1999). La revista principal en este campo de la demostración de teoremas es la *Journal of Automated Reasoning*; la conferencia más importante es la *Conference on Automated Deduction* (CADE), que es anual. La investigación en la demostración de teoremas también está estrechamente relacionada con el uso de la lógica en el análisis de programas y los lenguajes de programación; y cuya conferencia más importante es la *Logic in Computer Science*.



EJERCICIOS

9.1 Demuestre a partir de los principios básicos que la Especificación Universal es sólida y que la Especificación Existencial genera una base de conocimiento inferencialmente equivalente.

9.2 De *Gusta(Jerry, Helado)* parece razonable inferir $\exists x \text{Gusta}(x, \text{Helado})$. Apunte una regla de inferencia general, **Introducción del Existencial**, que sancione esta inferencia. Establezca cuidadosamente las condiciones que deben satisfacer las variables y los términos involucrados.

9.3 Suponga que una base de conocimiento contiene tan sólo una sentencia, $\exists x \text{TanAltoComo}(x, \text{Everest})$. ¿Cuál de los siguientes resultados son legítimos al aplicar la Especificación Existencial?

- a) *TanAltoComo(Everest, Everest)*.
- b) *TanAltoComo(Kilimanjaro, Everest)*.
- c) *TanAltoComo(Kilimanjaro, Everest) \wedge TanAltoComo(BenNevis, Everest)* (después de dos aplicaciones).

9.4 Dé el unificador más general, si existe, de cada una de las siguientes parejas de sentencias atómicas:

- a) *P(A, B, B), P(x, y, z)*.
- b) *Q(y, G(A, B)), Q(G(x, x), y)*.
- c) *Mayor(Padre(y), y), Mayor(Padre(x), John)*.
- d) *Conoce(Padre(y), y), Conoce(x, x)*.

9.5 Teniendo en cuenta los retículos de subsunción de la Figura 9.2.

- a) Construya el retículo de la sentencia *Contrata(Madre(John), Padre(John))*.
- b) Construya el retículo de la sentencia *Contrata(IBM, y)* («Cada uno trabaja en IBM»). Recuerde incluir cada tipo de petición que se unifica con la sentencia.
- c) Asuma que ALMACENAR indexa cada sentencia bajo cada nodo de su retículo de subsunción. Explique cómo BUSCAR trabajaría cuando alguna de estas sentencias contiene variables; utilice como ejemplos las sentencias de (a) y la petición *Contrata(x, Padre(x))*.

9.6 Suponga que introducimos en una base de datos lógica un segmento de los datos del censo de Estados Unidos listando la edad, la ciudad de residencia, la fecha de nacimiento, y la madre de cada persona, utilizando los números de la seguridad social como las constantes de identificador de cada persona. De este modo, la edad de George sería *Edad(443-65-1282, 56)*. ¿Cuál de los siguientes criterios de indexación C1-C5 permite una solución eficiente para cada una de las peticiones P1-P4 (asumiendo que se utiliza el encadenamiento hacia atrás estándar)?

- C1: un índice para cada átomo en cada posición.
- C2: un índice para cada primer argumento.
- C3: un índice para cada predicado atómico.
- C4: un índice para cada *combinación* de predicado y su primer argumento.

- **C5:** un índice para cada *combinación* de predicado y su segundo argumento y un índice para cada primer argumento (no estándar).
- **P1:** *Edad(443-44-4321, x)*
- **P2:** *Residen(x, Houston)*
- **P3:** *Madre(x, y)*
- **P4:** *Edad(x, 34) \wedge Residen(x, TinyTownUSA)*

9.7 Uno podría pensar que se puede evitar el conflicto entre variables en la unificación durante el encadenamiento hacia atrás estandarizando todas las sentencias de la base de conocimiento de una vez por todas. Demuestre que para algunas sentencias este enfoque no puede aplicarse. (*Pista:* tenga en cuenta una sentencia, y una parte de ella que se unique con otra.)

9.8 Explique cómo escribir cualquier problema de SAT-3 de tamaño arbitrario utilizando una única cláusula positiva de primer orden y no más de 30 hechos base.

9.9 Escriba representaciones lógicas para las siguientes sentencias, que sean adecuadas para utilizarse con el Modus Ponens Generalizado.

- a) Los caballos, las vacas y los cerdos son mamíferos.
- b) El descendiente de un caballo es un caballo.
- c) Barba Azul es un caballo.
- d) Barba Azul es padre de Charlie.
- e) Descendiente y padre son relaciones inversas.
- f) Cada mamífero tiene un parente.

9.10 En este ejercicio utilizaremos las sentencias que escribió en el Ejercicio 9.9 para responder a una pregunta mediante el algoritmo de encadenamiento hacia atrás.

- a) Dibuje el árbol de demostración generado mediante un algoritmo de encadenamiento hacia atrás para la petición $\exists c \text{ Caballo}(c)$, donde las cláusulas se emparejan en el orden dado.
- b) ¿Qué nota en este dominio?
- c) ¿Cuántas soluciones para c realmente se siguen de sus sentencias?
- d) ¿Puede pensar en algún mecanismo para encontrarlas todas? (*Pista:* Podría querer consultar Smith *et al.*, (1986).)

9.11 Un acertijo infantil muy popular dice «No tengo hermanos ni hermanas, pero el padre de ese hombre es el hijo de mi padre». Utilice las reglas del dominio de la familia (Capítulo 8) para demostrar quién es ese hombre. Puede aplicar cualquiera de los métodos descritos en este capítulo. ¿Por qué piensa que este acertijo es difícil?

9.12 Trace la ejecución del algoritmo de encadenamiento hacia atrás de la Figura 9.6 cuando se aplica para resolver el problema del crimen. Muestre la secuencia de los valores que se toman en la variable *objetivos*, y ordénelos en un árbol.

9.13 El siguiente código de Prolog define un predicado *P*:

```
P(X, [X|Y]) .  
P(X, [Y|Z]) :- P(X, Z) .
```

- a)** Muestre los árboles de demostración y las soluciones para las peticiones $P([1, 2, 3])$ y $P(2, [1, A, 3])$.
- b)** ¿Qué operación estándar sobre listas representa P ?

9.14 En este ejercicio veremos la ordenación mediante el Prolog.

- a)** Escriba cláusulas en Prolog que definan el predicado `ordenado(L)`, que es verdadero si y sólo si la lista `L` está ordenada en orden ascendente.
- b)** Escriba una definición en Prolog del predicado `perm(L, M)`, que es verdadero si y sólo si `L` es una permutación de `M`.
- c)** Defina `ordena(L, M)` (`M` es la versión ordenada de `L`) utilizando `perm` y `ordenado`.
- d)** Ejecute `ordena` sobre listas más y más largas hasta que pierda la paciencia. ¿Cuál es la complejidad temporal de su programa?
- e)** Escriba un algoritmo de ordenación más rápido, como la ordenación por inserción o la ordenación rápida, en Prolog.

9.15 En este ejercicio veremos la aplicación recursiva de las reglas de reescritura mediante la programación lógica. Una regla de reescritura (o **demodulador** en la terminología del OTTER) es una ecuación con una dirección especificada. Por ejemplo, la regla de reescritura $x + 0 \rightarrow x$ sugiere reemplazar cualquier expresión que empareje con $x + 0$ por la expresión x . La aplicación de las reglas de reescritura es una parte central de los sistemas de razonamiento cuasiconjuntivos. Utilizaremos el predicado `rescribir(X; Y)` para representar las reglas de reescritura. Por ejemplo, la regla de reescritura anterior se escribe `rescribir(X + 0, X)`. Algunos términos son *primitivos* y por tanto no se pueden simplificar más; por tanto, escribiremos `primitivo(0)` para decir que 0 es un término primitivo.

- a)** Escriba una definición de un predicado `simplificar(X, Y)`, que es verdadero cuando `Y` es una versión simplificada de `X`, es decir, cuando no hay más reglas de reescritura aplicables a cualquier subexpresión de `Y`.
- b)** Escriba un conjunto de reglas para la simplificación de expresiones que tengan operadores aritméticos, y aplique su algoritmo de simplificación a algunas de las expresiones de muestra.
- c)** Escriba un conjunto de reglas de reescritura para la diferenciación simbólica, y utilícelas entre sus reglas de simplificación para diferenciar y simplificar expresiones que tengan expresiones aritméticas, incluyendo la exponentiación.

9.16 En este ejercicio vamos a tener en cuenta la implementación de los algoritmos de búsqueda en Prolog. Suponga que `sucesor(X, Y)` es verdadero cuando `Y` es el sucesor del estado `X`; y que `objetivo(X)` es verdadero cuando `X` es el estado objetivo. Escriba una definición para `resolver(X, C)`, que significa que `C` es un camino (lista de estados) comenzando por `X` y acabando en el estado objetivo, y que consiste en una secuencia de pasos legales tal como se definen mediante el predicado `sucesor`. Encuentra que la búsqueda del primero en profundidad es la forma más fácil de hacerlo. ¿Cómo de fácil sería añadir una heurística para el control de la búsqueda?

9.17 ¿Cómo se puede utilizar la resolución para demostrar que una sentencia es válida?, ¿e insatisfacible?

9.18 De «Los caballos son animales», se sigue que «La cabeza de un caballo es la cabeza de un animal». Demuestre que esta inferencia es válida llevando a cabo los siguientes pasos:

- a) Traduzca la premisa y la conclusión al lenguaje de la lógica de primer orden. Utilice tres predicados: *CabezaDe(c, x)* (que significa que «c es la cabeza de x»), *Caballo(x)* y *Animal(x)*.
- b) Niegue la conclusión y transforme la premisa y la conclusión negada a la forma normal conjuntiva.
- c) Utilice la resolución para demostrar que la conclusión se sigue de la premisa.

9.19 Aquí tenemos dos sentencias en el lenguaje de la lógica de primer orden:

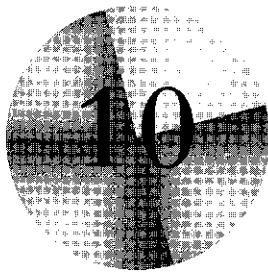
$$(A): \forall x \exists y (x \geq y)$$

$$(B): \exists y \forall x (x \geq y)$$

- a) Asuma que el rango de las variables está sobre todos los números naturales 0, 1, 2, ..., ∞ y que el predicado « \geq » significa «es mayor o igual a». Bajo esta interpretación, transforme (A) y (B) al lenguaje natural.
- b) ¿Es (A) verdadero bajo esta interpretación?
- c) ¿Es (B) verdadero bajo esta interpretación?
- d) ¿(A) implica lógicamente (B)?
- e) ¿(B) implica lógicamente (A)?
- f) Utilizando la resolución, intente demostrar que (A) se sigue de (B). Hágalo aunque piense que (B) no implica lógicamente (A); continúe hasta que la demostración se interrumpa y no pueda proceder (si se ha interrumpido). Muestre que unificación y sustitución en cada paso de la resolución. Si la demostración falla, explique concretamente dónde, cómo y por qué ha fallado.
- g) Ahora intente demostrar que (B) se sigue de (A).

9.20 La resolución puede generar demostraciones no constructivas a partir de peticiones con variables, así que teníamos que introducir mecanismos especiales para extraer las respuestas positivas. Explique por qué este problema no aparece en las bases de datos que sólo contienen cláusulas positivas.

9.21 En este capítulo hemos comentado que la resolución no se puede utilizar para generar todas las consecuencias lógicas de un conjunto de sentencias. ¿Algún algoritmo lo puede hacer?



Representación del conocimiento

Donde se muestra cómo utilizar la lógica de primer orden para representar los aspectos más importantes del mundo real como las acciones, el espacio, el tiempo, los eventos mentales y el hecho de ir de compras.

En los últimos tres capítulos se han descrito las tecnologías en las que se sustentan los agentes basados en el conocimiento: la sintaxis, la semántica y las demostraciones teóricas de la lógica proposicional y de primer orden, así como la implementación de los agentes que utilizan este tipo de lógica. En este capítulo se abordará la cuestión de qué *contenido* incorporar a la base de conocimiento de los agentes (cómo representar hechos acerca del mundo).

La Sección 10.1 introduce la idea general de ontología, que organiza todo lo existente en el mundo en una jerarquía de categorías. La Sección 10.3 aborda la representación de las acciones, aspecto fundamental en la construcción de agentes basados en conocimiento. La Sección 10.2 cubre las categorías básicas de objetos y sustancias, y la Sección 10.3 explica el concepto más general de **eventos**, o segmentos espacio-temporales. La Sección 10.4 habla sobre el conocimiento acerca de las creencias y la Sección 10.5 proporciona todo el conocimiento de forma conjunta en el contexto de una tienda que vende a través de Internet. Los Apartados 10.6 y 10.7 cubren los sistemas de razonamiento especializados para representar incertezza y conocimiento cambiante.

10.1 Ingeniería ontológica

En dominios de «juguete», el problema de la representación no es importante y es fácil encontrar un vocabulario consistente. Por otro lado, los dominios complejos como son

la compra utilizando Internet o el control de un robot en un entorno físico cambiante, requieren representaciones más generales y flexibles. Este capítulo muestra cómo crear estas representaciones, concentrándose en conceptos generales (como las *acciones*, el *tiempo*, los *objetos físicos* y las *creencias*) que ocurren en dominios muy diferentes. La representación de estos conceptos abstractos se suele denominar **ingeniería ontológica** (relacionada con el proceso de la **ingeniería del conocimiento** descrito en la Sección 8.4, aunque opera a gran escala).

La posibilidad de representarlo *todo* en el mundo, es una tarea de enormes proporciones. Por supuesto, no se va a realizar una descripción completa de todo (eso sería demasiado hasta para un libro de 1.000 páginas), pero se dejarán moldes donde se pueda incorporar nuevo conocimiento, sea cual sea el dominio. Por ejemplo, se definirá lo que significa un objeto físico, y los detalles de diferentes tipos de objetos (robots, televisores o cualquier otra cosa), que pueda ser rellenado con posterioridad. El marco de trabajo general para los conceptos se llama **ontología superior**, debido a la convención general de representar en los grafos los conceptos generales en la parte superior y los conceptos más específicos debajo de ellos, como en la Figura 10.1.

Antes de considerar la ontología en más profundidad, se planteará una advertencia importante. Se ha seleccionado el uso de la lógica de primer orden para tratar el contenido y la organización del conocimiento. Ciertos aspectos del mundo real son difíciles de capturar en LPO (lógica de primer orden). La principal dificultad es que casi todas las generalizaciones tienen excepciones, o son ciertas sólo en un determinado grado. Por ejemplo, aunque «los tomates son rojos» es una regla útil, algunos tomates son verdes, amarillos o naranjas. Se pueden encontrar excepciones similares para casi todas las afirmaciones hechas en este capítulo. La habilidad para manejar excepciones e incertezas es extremadamente importante, pero es ortogonal con la tarea de comprender la ontología general. Por esta razón, se retrasará el tratamiento de excepciones

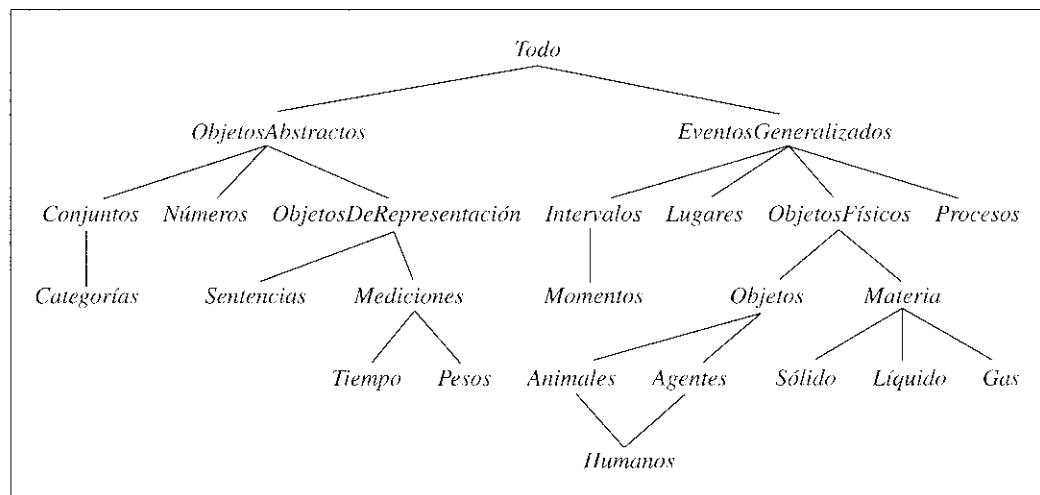


Figura 10.1 La ontología superior del mundo, en donde se indican los temas que más adelante se abordan en el capítulo. Cada arco indica que el concepto inferior es una especialización del concepto superior.

hasta la Sección 10.6, y los aspectos más generales sobre información incierta hasta el Capítulo 13.

¿Qué se utiliza en una ontología superior? Considérese de nuevo la ontología para circuitos de la Sección 8.4. Se realiza un gran número de asunciones para su simplificación. Por ejemplo, el tiempo se omite por completo. Las señales son fijas y no se propagan. La estructura de un circuito se mantiene constante. Si se quiere hacer más general, se considerarán las señales a distintos intervalos de tiempo y se incluirá la longitud de los cables y los retrasos de propagación. Esto permitiría simular las propiedades del tiempo en el circuito, y de hecho esas simulaciones son a menudo llevadas a cabo por diseñadores de circuitos. También se podrían introducir clases de puertas más interesantes, por ejemplo especificando la tecnología (TTL, MOS, CMOS, etc.), así como las especificaciones de entrada/salida. Si se quisiera considerar la fiabilidad del diagnóstico, se introduciría la posibilidad de que la estructura del circuito o las propiedades de las puertas pudieran cambiar de forma espontánea. Para tener en cuenta las capacitaciones perdidas, se debería ir desde una representación puramente topológica hacia una descripción más realista de las propiedades geométricas.

Al centrarse en el mundo de los *wumpus*, se aplican consideraciones similares. Aunque se incluye el tiempo, tiene una estructura muy simple: nada sucede excepto cuando el agente actúa y todos los cambios son instantáneos. Una ontología más general, que se adapte mejor para el mundo real, debería permitir cambios simultáneos que se extienden en el tiempo. Se utiliza el predicado *Hoyo* para especificar qué cuadrados tienen hoyos. Se podrían haber permitido diferentes clases de hoyos teniendo varios individuos que pertenecieran a dichas clases, cada uno de ellos con diferentes propiedades. De igual modo, se podría querer permitir otros animales además de *wumpuses*. No sería posible identificar las especies exactas a partir de las percepciones, por lo tanto sería necesario construir una taxonomía biológica del mundo de los *wumpus* para ayudar al agente a predecir el comportamiento partiendo de pistas escasas.

Para una ontología de propósito específico, es posible hacer cambios como estos para moverse hacia una mayor generalidad. Entonces surge una pregunta obvia: ¿convergerán todas estas ontologías en una ontología de propósito general? Después de siglos de investigación filosófica y computacional, la respuesta es «posiblemente». En esta sección, se propondrá una versión, que representa una síntesis de las ideas de todos estos siglos. Hay dos características principales en las ontologías de propósito general que las distinguen de la colección de ontologías de propósito específico:

- Una ontología de propósito general debe ser aplicable en mayor o menor medida a cualquier dominio de propósito específico (con la inclusión de axiomas específicos del dominio). Esto significa que, en tanto como sea posible, no se deben refinrar aspectos de representación ni ser ignorados.
- En un dominio dispar, las diferentes áreas de conocimiento deben ser *unificadas*, puesto que el razonamiento y la resolución de problemas podría involucrar varias áreas simultáneamente. Un sistema de reparación de circuitos para robots, por ejemplo, necesitar razonar acerca de los circuitos en términos de conectividad eléctrica y disposición física, y sobre el tiempo para realizar el análisis de tiempos y estimar el costo de la obra. Las sentencias que describen el tiempo deben ser ca-

paces de poder ser combinadas con aquellas que describen disposición espacial, y deben trabajar igualmente bien con nanosegundos y minutos o utilizando angstroms y metros.

Después de presentar la ontología general, se utilizará para describir el dominio de compra por Internet. Este dominio es más que adecuado para ejercitarse la ontología propuesta, y proporciona el alcance suficiente para que el lector realice alguna representación creativa de conocimiento por su cuenta. Considérese por ejemplo, que el agente de compra por Internet debe conocer millares de títulos y autores para comprar libros en Amazon.com, todas las clases de comidas para comprar provisiones en Peapod.com y todo lo que cualquiera puede encontrarse en una estación de servicio para buscar ganancias en Ebay.com¹.

10.2 Categoría y objetos

CATEGORÍAS



La organización de objetos en **categorías** es una parte vital de la representación del conocimiento. Aunque la interacción con el mundo tiene lugar a nivel de objetos individuales, *la mayoría del proceso de razonamiento tiene lugar en el nivel de categorías*. Por ejemplo, un comprador puede tener el objetivo de comprar un balón de baloncesto, en lugar de un balón de baloncesto *concreto* como *BB*₀. Las categorías también sirven para hacer predicciones sobre los objetos una vez que están clasificados. Se puede inferir la presencia de ciertos objetos a través de la percepción, inferir la categoría a la que pertenece utilizando las propiedades del objeto percibidas y entonces usar la información sobre categorías para realizar predicciones sobre los objetos. Por ejemplo, a partir de las características verde, cáscara moteada, tamaño grande y con forma ovalada, uno puede inferir que un objeto es una sandía; a partir de esto, uno puede inferir que puede ser útil para una ensalada de frutas.

Existen dos opciones para representar categorías en lógica de primer orden: predicados y objetos. Es decir, se puede usar el predicado *Balón_de_baloncesto(b)*, o se puede **reformular** la categoría como un objeto, *Balones_de_baloncesto*. Entonces se puede decir *Miembro(b, Balones_de_Baloncesto)* (que se puede abbreviar como $b \in \text{Balones_de_baloncesto}$) para decir que *b* es un miembro de la categoría de balones de baloncesto. Se utiliza *Subconjunto(Balones_de_baloncesto, Balones)* (abreviado como *Balones_de_baloncesto ⊂ Balones*) para indicar que los balones de baloncesto son una subcategoría, o subconjunto, de los *Balones*. Entonces se puede pensar que una categoría es un conjunto que agrupa a sus miembros, o se puede pensar que es un objeto más complejo que surge cuando tienen sentido las relaciones de *Miembro* y *Subconjunto* definidas para él.

HERENCIA

Las categorías sirven para organizar y simplificar el conocimiento base, a través de la **herencia**. Se dice que todos los objetos de la categoría *Alimentos* son comestibles, y

¹ Discúlpese si debido a circunstancias fuera de nuestro control, algunas de las tiendas *on-line* no se encuentran funcionando en el momento que el lector lea estas líneas.

se afirma que *Fruta* es una subclase de *Alimentos* y que *Manzanas* en una subclase de *Fruta*, entonces se sabe que cualquier manzana es comestible. Se dice que las manzanas individuales **heredan** la propiedad comestible, es este caso por su función de pertenencia a la categoría *Alimentos*.

TAXONOMÍA

Las relaciones de subclasificación organizan categorías en **taxonomías**, o **relaciones taxonómicas**. Las taxonomías se han utilizado explícitamente desde hace siglos en campos técnicos. Por ejemplo, la biología sistemática intenta proporcionar una taxonomía para todas las especies vivas y extinguidas; la bibliografía de ciencias ha desarrollado una taxonomía de todos los campos del conocimiento, codificado como sistema Decimal de Dewey, mientras que los departamentos de impuestos y otros departamentos gubernamentales han desarrollado grandes taxonomías sobre ocupaciones y productos comerciales. Las taxonomías son también un aspecto importante en el conocimiento general del sentido común.

La lógica de primer orden hace sencillo realizar afirmaciones sobre categorías, ya sea relacionando objetos con categorías o cuantificando sus miembros:

- Un objeto es un miembro de una categoría. Por ejemplo:

$$BB_9 \in Balones_de_baloncesto$$

- Una categoría es subclase de otra categoría. Por ejemplo:

$$Balones_de_baloncesto \subset Balones$$

- Todos los miembros de una categoría tienen algunas propiedades. Por ejemplo:

$$x \in Balones_de_baloncesto \Rightarrow Redondo(x)$$

- Miembros de una categoría se pueden reorganizar por algunas propiedades. Por ejemplo:

$$\begin{aligned} Naranja(x) \wedge Redondo(x) \wedge Diámetro(x) = 9.5'' \wedge x \in Balones \Rightarrow x \in \\ \in Balones_de_baloncesto \end{aligned}$$

- Una categoría como conjunto tiene algunas propiedades. Por ejemplo:

$$Perros \in EspeciesDomesticadas$$

Nótese que debido a que *Perros* es una categoría y es un miembro de *EspeciesDomesticadas*, esta última debe ser una categoría de categorías. Se podrían tener incluso categorías de categorías de categorías, pero no son de mucha utilidad.

Aunque las relaciones de subclasificación y miembro son las más importantes para las categorías, también se quiere ser capaz de modelar relaciones entre categorías que no son subclase unas de otras. Por ejemplo, si se dice que *Machos* y *Hembras* son subclases de *Animales*, entonces no se afirma que *Machos* no sean *Hembras*. Se dice que dos o más categorías son **disjuntas**, si no tienen miembros en común. Incluso si se conoce que machos y hembras son disjuntos, no se sabe que un animal que no es un macho, debe ser una hembra, a menos que se explice que machos y hembras constituyen una **descomposición exhaustiva** de los animales. Una descomposición exhaustiva disjunta se conoce como una **partición**. Los siguientes ejemplos ilustran estos tres conceptos:

$$Disjunto(\{Animales, Vegetales\})$$

$$DescomposiciónExhaustiva(\{Americanos, Canadienses, Mexicanos\},$$

$$Norteamericanos)$$

$$Partición(\{Machos, Hembras\}, Animales)$$

DISJUNTAS

DESCOMPOSICIÓN EXHAUSTIVA

PARTICIÓN

(Nótese que la *DescomposiciónExhaustiva* de *Norteamericanos* no es una *Partición*, porque alguna gente tiene doble nacionalidad.) Los tres predicados se definen a continuación:

$$\begin{aligned} \text{Disjunto}(s) &\Leftrightarrow (\forall c_1, c_2 \in s \wedge c_1 \neq c_2 \Rightarrow \text{Intersección}(c_1, c_2) = \{\}) \\ \text{DescomposiciónExhaustiva}(s, c) &\Leftrightarrow (\forall i \in c \Leftrightarrow \exists c_2 \in s \wedge i \in c_2) \\ \text{Partición}(s, c) &\Leftrightarrow \text{Disjunto}(s) \wedge \text{DescomposiciónExhaustiva}(s, c) \end{aligned}$$

Las categorías también se pueden definir proporcionando las condiciones necesarias y suficientes para la función de pertenencia. Por ejemplo, soltero es un macho adulto no casado:

$$x \in \text{Solteros} \Leftrightarrow \text{NoCasado}(x) \wedge x \in \text{Adultos} \wedge x \in \text{Machos}$$

Como se comenta en el recuadro correspondiente al género natural, no siempre es posible formular definiciones lógicas rigurosas de las categorías, ni en todos los casos es necesario.

Objetos compuestos

La idea de que un objeto puede ser parte de otro es familiar. La nariz forma parte de la cabeza, Rumanía es parte de Europa y este capítulo es parte de este libro. En general, se utiliza la relación *ParteDe* para decir que algo forma parte de otra cosa. Los objetos se pueden agrupar dentro de jerarquías *ParteDe*, reminiscencia de la jerarquía *Subconjunto*:

$$\begin{aligned} &\text{ParteDe(Bucarest, Rumanía)} \\ &\text{ParteDe(Rumanía, EuropaDelEste)} \\ &\text{ParteDe(EuropaDelEste, Europa)} \\ &\text{ParteDe(Europa, Tierra)} \end{aligned}$$

La relación *ParteDe* es transitiva y reflexiva: es decir,

$$\begin{aligned} \text{ParteDe}(x, y) \wedge \text{ParteDe}(y, z) &\Rightarrow \text{ParteDe}(x, z) \\ \text{ParteDe}(x, x) \end{aligned}$$

Por lo tanto, se puede concluir *ParteDe(Bucarest, Tierra)*.

Las categorías de **objetos compuestos** se caracterizan a menudo por relaciones estructurales entre las partes. Por ejemplo, un bípedo tiene dos piernas unidas a su cuerpo:

$$\begin{aligned} \text{Bipedo}(a) \Rightarrow & \exists l_1, l_2, b \text{ Pierna}(l_1) \wedge \text{Pierna}(l_2) \wedge \text{Cuerpo}(b) \wedge \\ & \text{ParteDe}(l_1, a) \wedge \text{ParteDe}(l_2, a) \wedge \text{ParteDe}(b, a) \wedge \\ & \text{UnidaA}(l_1, b) \wedge \text{UnidaA}(l_2, b) \wedge \\ & l_1 \neq l_2 \wedge [\forall l_3 \text{ Pierna}(l_3) \wedge \text{ParteDe}(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)] \end{aligned}$$

La notación de «exactamente dos» no es la más adecuada. Esto fuerza a decir que hay dos piernas, que no son las mismas, y que si alguien propone una tercera pierna, deberá ser la misma que una de las otras dos. En la Sección 10.6, se verá cómo un formalismo denominado descripción lógica hace mucho más fácil representar restricciones como «exactamente dos».

Se puede definir una *ParticiónDePartes* como análoga a la relación de *Partición* correspondiente a las categorías (véase Ejercicio 10.6). Los objetos están constituidos por las partes de su *ParticiónDePartes*, y se puede considerar que algunas de sus propiedades se derivan de tales partes. Por ejemplo, la masa de un objeto compuesto, es la suma de cada una de sus partes. Conviene advertir que no sucede lo mismo con las categorías, las cuales no tienen masa, aunque sus elementos sí puedan tenerla.

Es conveniente definir los objetos compuestos mediante partes bien definidas, aunque sin una estructura determinada. Por ejemplo, se podría afirmar: «Las manzanas de esta bolsa pesan kilo y medio». La tentación sería adscribir este peso al conjunto de manzanas en la bolsa, pero esto es un error, porque el conjunto es un concepto matemático abstracto que tiene elementos pero no tiene peso. En su lugar, se precisa de un nuevo concepto, el cual se llamará **montón**. Por ejemplo, si las manzanas son *Manzana₁*, *Manzana₂* y *Manzana₃*, entonces

$$\text{MontónDe}(\{\text{Manzana}_1, \text{Manzana}_2, \text{Manzana}_3\})$$

denota al objeto compuesto cuyas partes son las tres manzanas (no elementos). Se puede utilizar el concepto de montón como un objeto normal, aunque no estructurado. Nótese que *MontónDe(Manzanas)* es un objeto compuesto formado por todas las manzanas (que no debe ser confundido con *Manzanas*, la categoría o conjunto de todas las manzanas).

Se puede definir *MontónDe* en términos de la relación *ParteDe*. Obviamente, cada elemento de *s* es parte de *MontónDe(s)*:

$$\forall x \ x \in s \Rightarrow \text{ParteDe}(x, \text{MontónDe}(s))$$

Además, *MontónDe(s)* es el objeto más pequeño que satisface esta condición. En otras palabras, *MontónDe(s)* debe ser parte de cualquier objeto que tiene todos los elementos de *s* como partes:

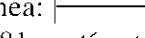
$$\forall y [\forall x \ x \in s \Rightarrow \text{ParteDe}(x, y)] \Rightarrow \text{ParteDe}(\text{MontónDe}(s), y)$$

MIMIZACIÓN LÓGICA

Estos axiomas son un ejemplo de una técnica general llamada **minimización lógica**, que define a un objeto como el más pequeño que satisface ciertas condiciones.

Medidas

MEDIDAS

Tanto en las teorías científicas del mundo como en las que apelan al sentido común, los objetos poseen peso, masa, costo, etc. Los valores que se asignan a estas propiedades se conocen como **medidas**. Es muy fácil representar medidas cuantitativas. Se puede pensar que en el universo existen «objetos de medida» abstractos tales como la *longitud*, que es la longitud de este segmento de línea: . A la longitud anterior se puede llamar 1,5 pulgadas o 3,81 centímetros. Es decir, la misma longitud puede denominarse de diferentes formas en el lenguaje. Lógicamente, esto se realiza combinando una **función de unidades** con un número (un esquema alternativo se explora en el Ejercicio 10.8). Si se denota al segmento de línea como *L₁*, se puede escribir

$$\text{Longitud}(L_1) = \text{Pulgadas}(1,5) = \text{Centímetros}(3,81)$$

FUNCIÓN DE UNIDADES

LOS GÉNEROS NATURALES

Algunas categorías se definen de manera rigurosa: un objeto se considera como triángulo si y sólo si es un polígono de tres lados. Por el contrario, la mayor parte de las categorías del mundo real que no tienen una definición precisa se llaman de **género natural**. Por ejemplo, los tomates en general tienen un color escarlata tenue, son más o menos esféricos y tienen una pequeña depresión en la parte superior, que es donde está el tallo, tienen un diámetro que puede ir desde 7.5 hasta 10 cm, tienen piel delgada pero resistente y en el interior tienen pulpa, semillas y jugo. Aunque también hay variaciones: algunos tomates son de color naranja, los tomates que no están todavía maduros son verdes, algunos son más grandes y otros más pequeños que el promedio, mientras que los tomates pequeños de ensalada son todos igual de pequeños. En vez de una definición completa de los tomates, se tiene un conjunto de características que permiten identificar esos objetos, que evidentemente son tomates típicos. Características que no permitirán tomar decisiones en el caso de otros objetos. ¿Acaso puede haber tomates con piel de melocotón?

Lo anterior plantea un problema al agente lógico. Éste no puede estar seguro de que aquello que ha percibido sea un tomate, incluso estando seguro de ello; no tendría la certeza de qué propiedades de un tomate típico tiene ese tomate en particular. El problema anterior es consecuencia inevitable de operar en entornos parcialmente observables.

La idea clave consiste en separar lo que es válido para todos los casos concretos de una categoría, de aquello que sólo se cumple en los casos típicos de esa categoría. Por ejemplo, además de la categoría *Tomates*, también se dispone de la categoría *Típicos(Tomates)*. En este caso, *Típico* es una función que correlaciona una categoría con la subclase de tal categoría en la que se encuentran sólo los casos típicos:

$$\text{Típico}(c) \subseteq c$$

De hecho, gran parte del conocimiento sobre los géneros naturales se refiere a los casos típicos:

$$x \in \text{Típico}(\text{Tomates}) \Rightarrow \text{Rojo}(x) \wedge \text{Redondo}(x)$$

De esta forma, se pueden poner por escrito hechos útiles acerca de la categorías, sin tener que ofrecer definiciones exactas.

Wittgenstein (1953), en su libro *Philosophical Investigations*, explicó lo difícil que es ofrecer definiciones exactas para la mayoría de las categorías naturales. Empleó el ejemplo de los juegos, para mostrar que los miembros de una categoría lo que tenían en común eran «parecidos familiares», más que características necesarias y suficientes.

Quine (1953), desafió también la utilidad de la noción de definición rigurosa. Comentó que, incluso una definición como la anterior de «soltero», deja que desear. Por ejemplo, cuestionó afirmaciones como la de que «el Papa es soltero». Aunque esta afirmación no es estrictamente falsa, su uso es desafortunado porque induce a inferencias no deseadas por parte del receptor. El conflicto podría ser resuelto distinguiendo entre definiciones lógicas adecuadas para representación de conocimiento interno, y aquellos criterios más matizados para su correcto uso lingüístico. Estos últimos podrían ser alcanzados filtrando las aserciones derivadas de las primeras. También puede ser posible que los fallos derivados del uso lingüístico sirvan como retroalimentación para modificar definiciones internas, filtrando aquellas que no sean necesarias.

La conversión entre una unidad y la otra se realiza igualando múltiplos de una unidad respecto a la otra:

$$\text{Centímetros}(2,54 \times d) = \text{Pulgadas}(d)$$

Se pueden escribir axiomas similares para libras y kilogramos, segundos y días, así como dólares y centavos. Las medidas se pueden usar para describir objetos, como por ejemplo:

$$\text{Diámetro}(\text{Balón_de_baloncesto}_{12}) = \text{Pulgadas}(9.5)$$

$$\text{Precio}(\text{Balón_de_baloncesto}_{12}) = \$19)$$

$$d \in \text{Días} \Rightarrow \text{Duración}(d) = \text{Horas}(24)$$

Es conveniente resaltar que $\$(1)$ no es un billete de un dólar. Uno puede tener dos billetes de dólar, pero hay sólo un objeto denominado $\$(1)$. Resaltar también que mientras $\text{Pulgadas}(0)$ y $\text{Centímetros}(0)$ se refieren al mismo valor cero de longitud, no son equivalentes a otras medidas de cero como $\text{Segundos}(0)$.

Es fácil representar las medidas sencillas y cuantitativas. Existe otro tipo de medidas que son más difíciles, pues no se dispone de una escala de valores bien definida. Los ejercicios tienen dificultad, los postres tienen delicia, los poemas tienen belleza, sin embargo a ninguna de estas cualidades se le puede asignar un número. Se podría estar tentado, en un afán por tener en cuenta sólo aquello que es contable, a descartar las propiedades anteriores al considerarlas inútiles para el razonamiento lógico, o lo que es peor, imponer una escala numérica a la belleza. Lo anterior sería un grave error, puesto que no es necesario. El aspecto más importante de las medidas no reside en los valores numéricos particulares en sí, sino en el hecho de que las medidas permiten una *ordenación*.

Aun cuando las medias no estén representadas por números, es posible compararlas entre sí mediante signos de ordenación como $>$. Por ejemplo, hay quienes consideran que los ejercicios de Norvig son más difíciles que los propuestos por Russell, y es más difícil obtener una buena calificación en éstos:

$$e_1 \in \text{Ejercicios} \wedge e_2 \in \text{Ejercicios} \wedge \text{Escribió}(\text{Norvig}, e_1) \wedge \text{Escribió}(\text{Russell}, e_2) \Rightarrow \\ \text{Dificultad}(e_1) > \text{Dificultad}(e_2)$$

$$e_1 \in \text{Ejercicios} \wedge e_2 \in \text{Ejercicios} \wedge \text{Dificultad}(e_1) > \text{Dificultad}(e_2) \Rightarrow \\ \text{Calificación Esperada}(e_1) < \text{Calificación Esperada}(e_2)$$

Lo anterior bastará para decidir qué ejercicios realizar, aunque no existan valores numéricos para tomar tal decisión (lo que sí sería necesario es saber quién escribió cada ejercicio). Este tipo de relaciones monotónicas que guardan entre sí las medidas, constituye la base del campo conocido como **física cualitativa**, un subcampo de la IA que investiga cómo razonar acerca de los sistemas físicos, sin tener que enfatizarse en la elaboración de minuciosas ecuaciones y simulaciones numéricas. En la sección de notas históricas se habla sobre la física cualitativa.

Sustancias y objetos

Quizás al mundo real se le podría considerar constituido por objetos primitivos (partículas) y por objetos compuestos construidos con estas partículas. Al razonar en el nivel

INDIVIDUALIZACIÓN

MATERIA O SUSTANCIA

SUSTANTIVOS CONTABLES

SUSTANTIVOS NO CONTABLES

INTRÍNSECAS

de objetos grandes como manzanas y coches, se elimina la complejidad de tratar por separado con una inmensa cantidad de objetos primitivos. Existe sin embargo, una importante porción de realidad que parecería desafiar todo tipo de **individualización**: separación en objetos distintos. A esta porción se le conoce genéricamente como **materia o sustancia**. Por ejemplo, supóngase que ante mí tengo un oso hormiguero y mantequillas. Si bien se puede afirmar que hay un oso hormiguero, no es obvio cómo se puede cuantificar la mantequilla, no es evidente qué cantidad de «objetos-mantequilla» hay, puesto que cualquier parte de un objeto-mantequilla, es también otro objeto-mantequilla, por lo menos hasta llegar a partes realmente diminutas. Lo anterior constituye la diferencia fundamental entre una sustancia y las cosas. Si se parte por la mitad al oso hormiguero, no resultan dos osos hormigueros, desafortunadamente.

Nótese que en español, como en otros idiomas, se distingue claramente entre sustancias y cosas. Se dice, por ejemplo, «un oso hormiguero», en cambio (excepto en algunos presuntuosos restaurantes californianos) nunca se pediría «una mantequilla». Los lingüistas establecen una diferencia entre **sustantivos contables** como osos hormigueros, orificios y teoremas y **sustantivos no contables** como mantequilla, agua y energía. Varias ontologías competentes reclaman el manejo de esta distinción. Aquí se describirá sólo una, las restantes se tratan en la sección de notas históricas.

Para representar correctamente a las sustancias, se tiene que empezar por lo que es obvio. En la ontología presentada, se tendrán que incluir por lo menos el grueso del «paquete» de aquellas sustancias con las que se interactúa. Por ejemplo, se podría considerar que la mantequilla es la misma que la que se dejó sobre la mesa la noche anterior, se puede coger, pesarla, venderla o cualquier otra cosa. En este sentido, la mantequilla es un objeto igual que el oso hormiguero. Llámese *Mantequilla*₃. Se definirá también la categoría *Mantequilla*. De manera no formal, sus elementos serán todas aquellas cosas de las que se puede afirmar «Es mantequilla», incluida *Mantequilla*₃. Aclarando que por ahora se omitirán algunas partes muy pequeñas, toda parte de un objeto-mantequilla es también un objeto-mantequilla:

$$x \in \text{Mantequilla} \wedge \text{ParteDe}(y, x) \Rightarrow y \in \text{Mantequilla}$$

Se puede decir ahora, que la mantequilla se derrite aproximadamente a los 30 °C:

$$x \in \text{Mantequilla} \Rightarrow \text{PuntoDeFusión}(x, \text{Centígrados}(30))$$

La mantequilla es amarilla, menos densa que el agua, se reblandece a temperatura ambiente, tiene alto contenido de grasas, etc. Por otra parte, la mantequilla no tiene tamaño, peso, ni forma específicos. Lo que sí se puede es definir categorías más especializadas para la mantequilla, por ejemplo *MantequillaSinSal*, que también es un tipo de sustancia, puesto que una parte de un objeto-mantequilla-sin-sal es también un objeto-mantequilla-sin-sal. Por otra parte, si se define una categoría *KiloDeMantequilla*, cuyos miembros sean todos los objetos-mantequilla que pesen un kilo, ¡ya se tiene una sustancia! Si se parte un kilo de mantequilla por la mitad, muy a pesar nuestro, el resultado no serán dos kilos de mantequilla.

Lo que realmente ha sucedido es lo siguiente: hay propiedades que son **intrínsecas**, pertenecen a la misma sustancia del objeto más que el objeto como un todo. Cuando se divide algo en dos, ambas partes conservan el mismo conjunto de propiedades

EXTRÍNSECAS

intrínsecas (cosas como la densidad, el punto de ebullición, el sabor, el color, la propiedad, etc.). Por otra parte, las propiedades **extrínsecas** son lo contrario: propiedades tales como peso, longitud, forma, función, etc., que después de dividir algo no se conservan.

Aquella clase de objetos que en su definición incorpore sólo propiedades *intrínsecas* es una sustancia, o un sustantivo no contable, mientras que una clase que incorpore *cualquier* propiedad extrínseca en su definición es un sustantivo contable. La categoría *Sustancia* es la categoría más general de las sustancias, en la que no se especifica ninguna propiedad intrínseca. La categoría *Objeto*, es la categoría de objetos discretos más general, en la que no se especifica ninguna propiedad extrínseca.

10.3 Acciones, situaciones y eventos

El razonamiento sobre los resultados de las acciones es fundamental para el funcionamiento de un agente basado en conocimiento. El Capítulo 7 proporciona ejemplos de sentencias proposicionales que describen cómo las acciones afectan al mundo de *wumpus* (por ejemplo, la Ecuación (7.3) en el Apartado 7.7, describe cómo la posición del agente cambia debido a un movimiento delantero). Una desventaja de la lógica proposicional, es la necesidad de tener diferentes copias de la descripción de la acción para cada intervalo de tiempo en la cual la acción se podría llevar a cabo. Esta sección describe un método de representación que utiliza lógica de primer orden para resolver este problema.

La ontología del cálculo de situaciones

Una forma obvia de resolver la necesidad de disponer de múltiples copias de los axiomas es simplemente cuantificar en el tiempo ($\forall t$, tal que es el resultado en $t + 1$ de realizar la acción en t). En vez de tratar con intervalos de tiempo explícitos como $t + 1$, esta sección utiliza *situaciones*, que denotan los estados resultantes de ejecutar acciones. Esta aproximación se denomina **cálculo de situaciones** y utiliza las siguientes ontologías:

CÁLCULO DE SITUACIONES

SITUACIONES

FLUJOS

- Como en el Capítulo 8, las acciones son términos lógicos como *HaciaDelante* y *Girar(derecha)*. Por ahora, se asumirá que el entorno contiene sólo un agente (si existe más de uno, se debe insertar un argumento adicional para decir qué agente está realizando las acciones).
- Las **situaciones** son términos lógicos que consisten en una situación inicial (normalmente denominada S_0), y todas las situaciones que son generadas mediante la aplicación de una acción a una situación. La función *Resultado(a, s)* (en ocasiones denominada *Do*), da nombre a la situación resultante de ejecutar una acción *a* en una situación *s*. La Figura 10.2 ilustra esta idea.
- Los **flujos** son funciones y predicados que varían de una situación a la siguiente, como la posición de un agente o la vitalidad de *wumpus*. El diccionario dice que flujo representa algo que fluye, como un líquido. Utilizando este concepto, se quie-

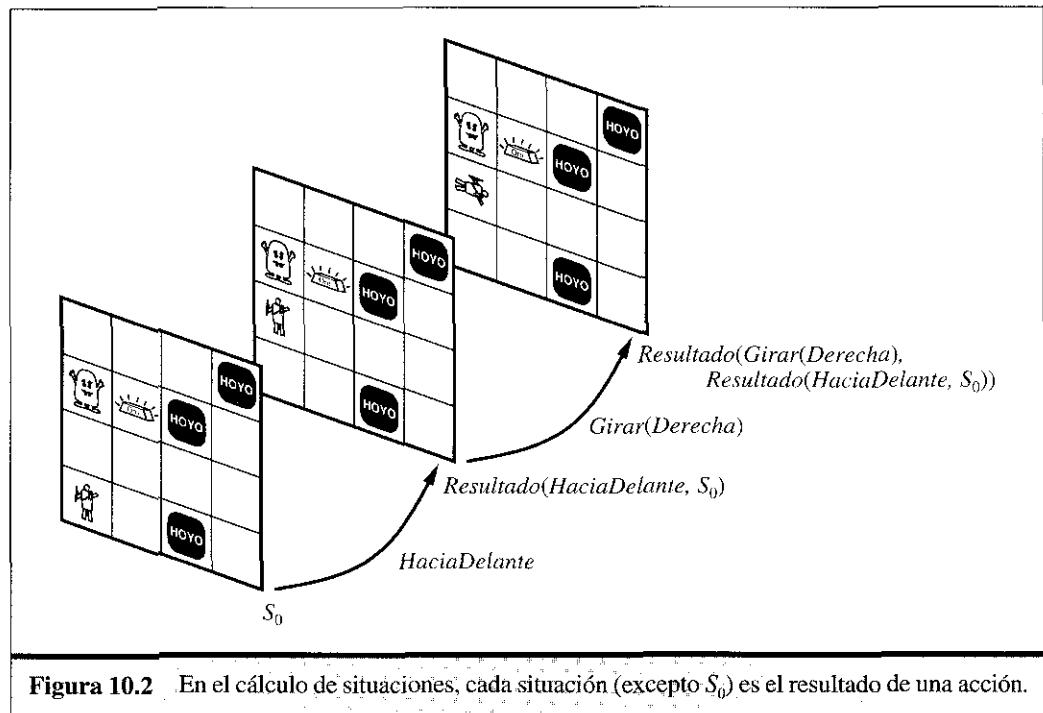


Figura 10.2 En el cálculo de situaciones, cada situación (excepto S_0) es el resultado de una acción.

re significar el flujo o cambio a través de las situaciones. Por convención, la situación es siempre el último argumento de un flujo. Por ejemplo, $\neg Sostener(G_1, S_0)$ indica que el agente no está sosteniendo el lingote de oro G_1 en la situación inicial S_0 . $Edad(Wumpus, S_0)$ se refiere a la edad de *wumpus* en S_0 .

- Las funciones o predicados **atemporales** o **eternos** también se permiten. Ejemplos son el predicado *Lingote_de_oro(G₁)* y la función *PiernaIzquierdaDe(Wumpus)*.

Como complemento a las acciones simples, también es útil razonar sobre las secuencias de acciones. Se pueden definir los resultados de las secuencias en términos de los resultados de acciones individuales. Primero, se establecerá que ejecutar una secuencia vacía, deja la situación inalterada

$$\text{Resultado}([], s) = s$$

Ejecutar una secuencia no vacía es lo mismo que ejecutar la primera acción, y entonces ejecutar el resto sobre la situación resultante:

$$\text{Resultado}([a] \text{ seq}, s) = \text{Resultado}(\text{seq}, \text{Resultado}(a, s))$$

Un agente de cálculo de situaciones debería ser capaz de deducir el resultado de una secuencia dada de acciones. Esta es la tarea de la **proyección**. Con un algoritmo adecuado de inferencia constructiva, debería ser capaz de encontrar una secuencia que logre el efecto deseado. Esto es lo que se denomina tarea de **planificación**.

Se utilizará un ejemplo de una versión modificada del mundo de *wumpus*, en la que no se tiene en cuenta la orientación del agente y a dónde puede *Ir*, dada una localiza-

ción y su localización adyacente. Supóngase que el agente está en [1, 1] y que el lingote de oro está en [1, 2]. El objetivo es tener el lingote de oro en [1, 1]. Los predicados de flujo son $En(o, x, s)$ y $Sosteniendo(o, s)$. Entonces la base de conocimiento inicial podría incluir la siguiente descripción:

$$En(Agente, [1, 1], S_0) \wedge En(GL, [1, 2], S_0)$$

De todas formas esto no es suficiente, porque no especifica qué no es cierto en S_0 (para una discusión más en profundidad sobre este punto, véase el Apartado 10.7). La descripción completa sería como sigue:

$$\begin{aligned} En(o, x, S_0) &\Leftrightarrow [(o = Agente \wedge x = [1, 1]) \vee (o = GL \wedge x = [1, 2])] \\ &\neg Sosteniendo(o, S_0) \end{aligned}$$

También se necesita decir que GL es un lingote de oro y que [1, 1] y [1, 2] son adyacentes:

$$Lingote_de_oro(GL) \wedge Adyacente([1, 1], [1, 2]) \wedge Adyacente([1, 2], [1, 1])$$

Uno desearía ser capaz de probar que el agente consigue su objetivo desplazándose a [1, 2], cogiendo el lingote de oro y volviendo a [1, 1]. Es decir:

$$En(GL, [1, 1], Resultado([Ir([1, 1], [1, 2]), Tomar(GL), Ir([1, 2], [1, 1])], S_0))$$

Una posibilidad más interesante es la de construir un plan para tomar el lingote de oro respondiendo a la pregunta «¿qué secuencia de acciones tienen como resultado que el lingote de oro esté al final en la posición [1, 1]?».

$$\exists seq \ En(GL, [1, 1], Resultado(seq, S_0))$$

Se verá qué debe existir en la base de conocimiento para que preguntas como ésta se puedan responder.

Descripción de acciones en el cálculo de situaciones

AXIOMA DE POSIBILIDAD

AXIOMA DE EFECTO

En la versión más simple del cálculo de situaciones, cada acción se describe por dos axiomas: un **axioma de posibilidad** que especifica cuándo es posible ejecutar una acción, y un **axioma de efecto** que determina qué sucede cuando se ejecuta una acción posible. Se utilizará $Possible(a, s)$ para expresar que es posible la ejecución de la acción a en la situación s . Los axiomas tienen la siguiente forma:

AXIOMA DE POSIBILIDAD: $Precondiciones \Rightarrow Possible(a, s)$.

AXIOMA DE EFECTO: $Possible(a, s) \Rightarrow Cambios\ que\ son\ el\ resultado\ de\ ejecutar\ una\ acción$.

Se presentan estos axiomas para el mundo modificado de *wumpus*. Para hacer más cortas las sentencias, se omitirán los cuantificadores universales cuyo ámbito sea la sentencia entera. Se asumirá que la variable s representa situaciones, a representa acciones, o representa objetos (incluyendo agentes), g representa lingotes de oro y x e y representan localizaciones.

Los axiomas de posibilidad para este mundo establecen que un agente puede moverse entre localizaciones adyacentes, tomar un lingote de oro en la posición actual y soltar un lingote de oro que está sosteniendo:

$$\begin{array}{ll} En(\text{Agente}, x, s) \wedge \text{Adyacente}(x, y) & \Rightarrow \text{Possible}(\text{Ir}(x, y), s) \\ \text{Lingote_de_oro}(g) \wedge En(\text{Agente}, x, s) \wedge En(g, x, s) & \Rightarrow \text{Possible}(\text{Tomar}(g), s) \\ \text{Sosteniendo}(g, s) & \Rightarrow \text{Possible}(\text{Soltar}(g), s) \end{array}$$

Los axiomas de efecto establecen que, si una acción es posible, entonces ciertas propiedades (flujos) tendrán lugar en la situación resultante de ejecutar la acción. Ir desde x a y supone estar en y , tomar el lingote de oro conlleva sostenerlo y soltar el lingote de oro supone no sostenerlo:

$$\begin{array}{ll} \text{Possible}(\text{Ir}(x, y), s) & \Rightarrow En(\text{Agente}, y, \text{Resultado}(\text{Ir}(x, y), s)) \\ \text{Possible}(\text{Tomar}(g), s) & \Rightarrow \text{Sosteniendo}(g, \text{Resultado}(\text{Tomar}(g), s)) \\ \text{Possible}(\text{Soltar}(g), s) & \Rightarrow \neg \text{Sosteniendo}(g, \text{Resultado}(\text{Soltar}(g), s)) \end{array}$$

Al haber propuesto estos axiomas, ¿se puede probar que este pequeño plan alcanzará el objetivo?, ¡desafortunadamente no! Al principio todo funciona correctamente: $\text{Ir}([1, 1], [1, 2])$ es ciertamente posible en S_0 y el axioma de efecto para Ir permite concluir que el agente alcanza $[1, 2]$:

$$En(\text{Agente}, [1, 2], \text{Resultado}(\text{Ir}([1, 1], [1, 2]), S_0))$$

Ahora se considerará la acción $\text{Tomar}(G)$. Se debe mostrar que en la nueva situación es posible, es decir,

$$En(G_1, [1, 2], \text{Resultado}(\text{Ir}([1, 1], [1, 2]), S_0))$$

Pero desgraciadamente nada en la base de conocimiento justifica esta conclusión. Intuitivamente, se entiende que la acción del agente Ir no debería de tener efecto en la colocación del lingote de oro, por lo tanto, éste debería estar en la posición $[1, 2]$, donde estaba en la situación S_0 . *El problema es que el axioma de efecto dice lo que cambia, pero no lo que permanece igual.*

Representar todas las cosas que permanecen inalterables es lo que se conoce con el nombre del **problema del marco**². Se debe encontrar una solución eficiente al problema del marco porque en el mundo real, casi todo permanece inalterable todo el tiempo. Cada acción afecta sólo a una pequeña fracción de todo lo que fluye.

Una aproximación es escribir **axiomas marco** explícitos, que lo que hagan sea especificar qué permanece inalterable. Por ejemplo, los movimientos del agente dejan otros objetos en la misma posición a menos que sean tomados:

$$En(o, x, s) \wedge (o \neq \text{Agente}) \wedge \neg \text{Sosteniendo}(o, s) \Rightarrow En(o, x, \text{Resultado}(\text{Ir}(y, z), s))$$

Si hay F predicados de flujo y A acciones, entonces se necesitarán $O(AF)$ axiomas marco. Por otro lado, si cada acción tiene como mucho E efectos, donde normalmente E es mucho menor que F , entonces se podría representar lo que sucede con una base de co-

² El nombre de «problema del marco» procede del concepto físico de «marco de referencia» (el fondo que se asume fijo respecto al cual se mide la acción). También tiene relación con el fondo de una película, en el cual se producen pocos cambios de una imagen a otra.



PROBLEMA
DEL MARCO

AXIOMAS MARCO

PROBLEMA DE LA REPRESENTACIÓN DEL MARCO

PROBLEMA DE LA INFERNICIA DEL MARCO

AXIOMAS ESTADO-SUCESOR

nocimiento mucho menor de tamaño $O(AE)$. Este es el **problema de la representación del marco**. El problema cercano relacionado es el **problema de la inferencia del marco**, consistente en proyectar los resultados de una secuencia de acciones de t fases en el instante de tiempo $O(Et)$, en lugar de en el instante $O(Ft)$ o $O(AEt)$. Se abordará cada problema por separado.

Resolver el problema de la representación del marco

La solución al problema de la representación del marco implica un ligero cambio en el punto de vista utilizado para escribir los axiomas. En lugar de especificar los efectos de cada acción, se considerará la forma en que cada predicado de flujo evoluciona en el tiempo³. Se denominará a los axiomas a utilizar **axiomas estado-sucesor**. Estos axiomas tienen la siguiente forma:

AXIOMA ESTADO-SUCESOR:

Acción es Posible \Rightarrow

*(Flujo es cierto en el estado resultante \Leftrightarrow Los efectos de las acciones se produjeron
v Eran ciertos antes y la acción los dejó igual)*

Después de la salvedad de que no se consideran acciones imposibles, nótese que la definición utiliza \Leftrightarrow , no \Rightarrow . Esto significa que el axioma especifica que el flujo será cierto si y sólo si la parte derecha es cierta. Dicho de otra forma, se especifica que el valor de verdad de cada flujo en el siguiente estado es una función de la acción y del valor de verdad en el estado actual. Esto significa que el siguiente estado viene especificado de forma completa por el estado actual, y por lo tanto, no se necesitan axiomas marco adicionales.

El axioma estado-sucesor para la localización del agente dice que el agente está en y después de ejecutar una acción, bien porque la acción es posible y consiste en el movimiento a y, o bien porque el agente se encontraba en y y la acción no es un movimiento a ningún lado:

Possible(a, s) \Rightarrow
*(En(Agente, y, Resultado(a, s)) \Leftrightarrow a = Ir(x, y)
v (En(Agente, y, s) \wedge a \neq Ir(y, z)))*

El axioma para *Sosteniendo* dice que el agente está sosteniendo g después de ejecutar una acción, siempre que la acción fuera tomar aplicado a g suponiendo que la acción tomar es posible, o bien si el agente ya estaba sosteniendo g y la acción no supone soltarlo:

Possible(a, s) \Rightarrow
*(Sosteniendo(g, Resultado(a, s)) \Leftrightarrow a = Tomar(g)
v (Sosteniendo(g, s) \wedge a \neq Soltar(g)))*

³ Esta es esencialmente la aproximación que utilizamos en la construcción del agente basado en circuito booleano del Capítulo 7. De hecho, los axiomas como las Ecuaciones (7.4) y (7.5) pueden ser vistos como axiomas estado-sucesor.



EFECTO IMPLÍCITO

PROBLEMA DE LA RAMIFICACIÓN

AXIOMAS DE NOMBRE ÚNICO

Los axiomas estado-sucesor solucionan el problema de la representación del marco, porque el tamaño total de axiomas es $O(AE)$ literales: cada uno de los E efectos de cada acción A , se menciona exactamente una vez. Los literales se aplican sobre los F diferentes axiomas, por lo que los axiomas tienen un tamaño medio de AE/F .

El lector inteligente habrá notado que los axiomas manejan el flujo En para el agente, pero no para el lingote de oro. Por lo tanto, no se puede demostrar todavía que el plan de tres fases consiga el objetivo de tener el lingote de oro en $[1, 1]$. Se necesita exponer que un **efecto implícito** de que un agente se mueva desde una posición x a una posición y , es que cualquier lingote de oro que porte, se moverá también (así como cualquier hormiga que estuviera en el lingote, cualquier bacteria en la hormiga, etc.). El tratar con efectos implícitos se conoce como el **problema de la ramificación**. Se discutirá el problema en general posteriormente, pero para este dominio específico se puede resolver escribiendo un axioma estado-sucesor más general para En . El nuevo axioma que engloba la versión anterior establece que un objeto o está en la posición y si el agente fue a y o o es el agente o algo que el agente estaba sosteniendo, o si o se encontraba ya en la posición y y el agente no fue a ningún otro sitio, siendo o el agente o algo que el agente estaba sosteniendo.

$$\text{Posible}(a, s) \Rightarrow$$

$$(En(o, y, Resultado(a, s)) \Leftrightarrow (a = Ir(x, y) \wedge (o = \text{Agente} \vee Sosteniendo(o, s))) \\ \vee (En(o, y, s) \wedge \neg(\exists z \ y \neq z \wedge a = Ir(y, z) \wedge \\ (o = \text{Agente} \vee Sosteniendo(o, s))))).$$

Existe un tecnicismo más: un proceso de inferencia que use estos axiomas debe ser capaz de evaluar desigualdades. El tipo de desigualdad más sencilla es entre constantes (por ejemplo, $\text{Agente} \neq G_1$). La semántica general de la lógica de primer orden permite distinguir constantes para referirse al mismo objeto, por lo tanto, la base de conocimiento debe incluir un axioma para prevenir esto. Los **axiomas de nombre único** establecen una desigualdad para cada par de constantes en la base de conocimiento. Cuando esto se asume por el demostrador de teoremas en vez de ser especificado en la base de conocimiento, se denomina **asunción de nombres únicos**. También se necesitan especificar desigualdades entre los términos de las acciones: $Ir([1, 1], [1, 2])$ es una acción diferente a $Ir([1, 2], [1, 1])$ o $Tomar(G_1)$. Primero, se establece que cada tipo de acción es distinta (que la acción Ir no es una acción $Tomar$). Para cada par de nombres de acción A y B , se tendrá que

$$A(x_1, \dots, x_m) \neq B(y_1, \dots, y_n)$$

A continuación, se establece que dos términos de acción con el mismo nombre de acción se refieren a la misma acción, sólo si las acciones afectan a los mismos objetos:

$$A(x_1, \dots, x_m) = A(y_1, \dots, y_n) \Leftrightarrow x_1 = y_1 \wedge \dots \wedge x_m = y_n$$

AXIOMAS DE ACCIÓN ÚNICA

Todo esto se denomina conjuntamente **axiomas de acción única**. La combinación de la descripción del estado inicial, axiomas estado-sucesor, nombres de axioma únicos y axiomas de acción única, es suficiente para demostrar que el plan propuesto consigue el objetivo.

Resolver el problema de la inferencia del marco

Los axiomas estado-sucesor resuelven el problema de la representación del marco, pero no el problema de la inferencia del marco. Considérese un plan p de t fases en el que $S_t = \text{Resultado}(p, S_0)$. Para decidir qué flujos son ciertos en S_t , se necesita considerar cada uno de los axiomas marco F en cada uno de las fases de tiempo t . Debido a que los axiomas tienen un tamaño medio de AE/F , esto supone un trabajo de inferencia de $O(AEt)$. La mayor parte del trabajo corresponde a copiar flujos que no cambian de una situación a la siguiente.

Para resolver el problema de la inferencia del marco, existen dos posibilidades. Primero, se podría descartar el cálculo de las situaciones e inventar un nuevo formalismo para escribir axiomas. Esto ha sido llevado a cabo por formalismos como el **cálculo de flujos**. En segundo lugar, se podría alterar el mecanismo de inferencia para manejar axiomas marco de forma más eficiente. Un detalle que debería ser posible es que la aproximación más simple fuera $O(AEt)$. ¿Por qué debería depender del número de acciones, A , cuando se conoce que se ejecuta exactamente una acción en cada instante de tiempo? Para ver cómo se pueden mejorar las cosas, primero se presentará el formato de los axiomas marco:

$$\begin{aligned} Possible(a, s) \Rightarrow \\ F_i(Resultado(a, s)) &\Leftrightarrow (a = A_1 \vee a = A_2 \dots) \\ &\quad \vee F_i(s) \wedge (a \neq A_3) \wedge (a \neq A_4) \wedge \dots \end{aligned}$$

Es decir, cada axioma menciona varias acciones que pueden hacer el flujo cierto y varias acciones que pueden hacerlo falso. Esto se puede formalizar introduciendo el predicado $EfectoPos(a, F_i)$, que significa que una acción a hace que F_i sea cierto, y $EfectoNeg(a, F_i)$ que significa que a hace que F_i sea falso. Entonces se puede rescribir el esquema de acciones anterior como:

$Possible(a, s) \Rightarrow$
 $F_i(Resultado(a, s)) \Leftrightarrow EfectoPos(a, F_i) \vee [F_i(s) \wedge \neg EfectoNeg(a, F_i)]$
 $EfectoPos(A_1, F_i)$
 $EfectoPos(A_2, F_i)$
 $EfectoNeg(A_3, F_i)$
 $EfectoNeg(A_4, F_i)$

Que esto se pueda hacer automáticamente, depende del formato exacto de los axiomas marco. Para llevar a cabo un procedimiento de inferencia eficiente usando axiomas como éste, se necesita realizar lo siguiente:

1. Indexar los predicados *EfectoPos* y *EfectoNeg* por su primer argumento, de forma que cuando se tenga una acción que ocurre en el instante de tiempo t , se pueda encontrar su efecto en un tiempo de $O(1)$.
 2. Indexar los axiomas de tal forma que cuando se conozca que F_i es un efecto de una acción, se pueda encontrar el axioma para F_i en un tiempo de $O(1)$. Por lo tanto, no es necesario considerar los axiomas para flujos que no son un efecto de una acción.

3. Representar cada situación como una situación previa más un incremento. De este modo, si no cambia nada desde un paso al siguiente, no se necesita realizar trabajo alguno. En la aproximación antigua, se necesitaría hacer un trabajo de $O(F)$ para generar una sentencia para cada flujo $F_i(\text{Resultado}(a, s))$ de la sentencia $F_i(s)$ precedente.

Por lo tanto, en cada instante de tiempo, es necesario centrarse en la acción actual, buscando sus efectos y actualizando el conjunto de flujos ciertos. Cada instante de tiempo tendrá una media E de estas actualizaciones, con una complejidad total de $O(Et)$. Esto constituye una solución al problema de la inferencia del marco.

El tiempo y el cálculo de eventos

CÁLCULO DE EVENTOS

El cálculo de situaciones funciona bien cuando existe un agente simple realizando acciones discretas e instantáneas. Cuando las acciones tienen una duración y se pueden solapar unas con otras, el cálculo de situaciones se convierte en engorroso. Por lo tanto, estos temas se tratarán con un formalismo alternativo conocido como el **cálculo de eventos**, basado en puntos en el tiempo en vez de en situaciones. Los términos «evento» y «acción» se pueden intercambiar. Informalmente, un «evento» se corresponde con un conjunto amplio de acciones, incluyendo aquellas sin un agente explícito. Son más sencillas de manejar en el cálculo de eventos que en el cálculo de situaciones.

En el cálculo de eventos, los flujos tienen lugar en puntos en el tiempo en vez de en situaciones. El axioma de cálculo de eventos dice que un flujo es cierto en un punto concreto en el tiempo, si el flujo fue iniciado por un evento en un instante de tiempo anterior y no fue finalizado por la intervención de algún otro evento. Las relaciones *Inicio* y *Terminación* representan un papel similar al de la relación *Resultado* en el cálculo de situaciones. *Inicio*(e, f, t) significa que la ocurrencia del evento e en el tiempo t causa que el flujo f sea cierto, mientras que *Terminación*(w, f, t) significa que f deja de ser cierto. Se utilizará *Sucede*(e, t) para reflejar que el evento e sucede en el tiempo t , y se utilizará *Interrumpido*(f, t, t_2) para expresar que f ha finalizado por algún evento en algún instante entre t y t_2 . Formalmente, el axioma es:

AXIOMA DE CÁLCULO DE EVENTOS:

$$\begin{aligned} T(f, t_2) &\Leftrightarrow \exists e, t \quad \text{Sucede}(e, t) \wedge \text{Inicio}(e, f, t) \wedge (t < t_2) \wedge \neg \text{Interrumpido}(f, t, t_2) \\ \text{Interrumpido}(f, t, t_2) &\Leftrightarrow \exists e, t_1 \quad \text{Sucede}(e, t_1) \wedge \text{Terminación}(e, f, t_1) \\ &\quad \wedge (t < t_1) \wedge (t_1 < t_2) \end{aligned}$$

Esto proporciona una funcionalidad similar al cálculo de situaciones, pero con la habilidad de poder hablar de puntos en el tiempo e intervalos; por lo tanto, se puede afirmar *Sucede*(Apagar(Conmutador₁), 1:00) para decir que una llave de luz será apagada exactamente a la 1:00 h.

Se han hecho varias extensiones al cálculo de eventos (algunas con más éxito que otras), para solucionar los problemas derivados de tener que representar eventos con duración, eventos concurrentes, eventos que cambian continuamente y otras complicaciones.

El cálculo de eventos puede ser extendido para manejar efectos indirectos, cambios continuos, efectos no deterministas, restricciones causales y otras situaciones. Se retomarán algunos de estos temas en el Apartado 10.3.

Eventos generalizados

Hasta ahora se han revisado dos conceptos principales: acciones y objetos. Ahora es el momento de ver cómo encajan en una ontología global, en la cual las acciones y los objetos pueden ser vistos como aspectos de un universo físico. Se utilizará un universo particular compuesto por dimensiones espacial y temporal. En el mundo de *wumpus*, el componente espacial se corresponde con una rejilla bidimensional y el tiempo es discreto. El mundo real tiene tres dimensiones en el espacio y una dimensión en el tiempo⁴, todas continuas. Un **evento generalizado** se compone de aspectos de alguna pieza espacio-temporal (un segmento del universo espacio-temporal de múltiples dimensiones). Esta abstracción generaliza la mayoría de los conceptos que se han visto hasta ahora, incluyendo las acciones, localizaciones, tiempo, flujos y objetos físicos. La Figura 10.3 da una idea general. A partir de ahora, se utilizará el término simple «evento» para referirse a eventos generalizados.

Por ejemplo, La Segunda Guerra Mundial es un evento que tuvo lugar en varios puntos en el espacio-tiempo, como representa la zona sombreada de la figura. Puede ser dividida en **subeventos**⁵:

SubEvento(BatallaDeBretaña, SegundaGuerraMundial)

De modo similar, la Segunda Guerra Mundial es un subevento del siglo xx:

SubEvento(SegundaGuerraMundial, SigloXX)

El siglo xx es un *intervalo* de tiempo. Los intervalos son trozos de espacio-tiempo que incluyen todo el espacio entre dos puntos de tiempo. La función *Período(e)* denota el intervalo más pequeño que encierra al evento *e*. *Duración(i)* es la longitud del tiempo que ocupa un intervalo, por lo tanto se puede decir *Duración(Período(SegundaGuerraMundial)) > Años(5)*.

Australia es un lugar. Un trozo con unos bordes delimitados en el espacio. Los bordes pueden variar en el tiempo, debido a cambios geológicos o políticos. Se utiliza el predicado *En* para denotar la relación de subevento que tiene lugar cuando la proyección espacial de un evento es *ParteDe* otro:

En(Sydney, Australia)

La función *Localización(e)* denota el lugar más pequeño que encierra al evento *e*.

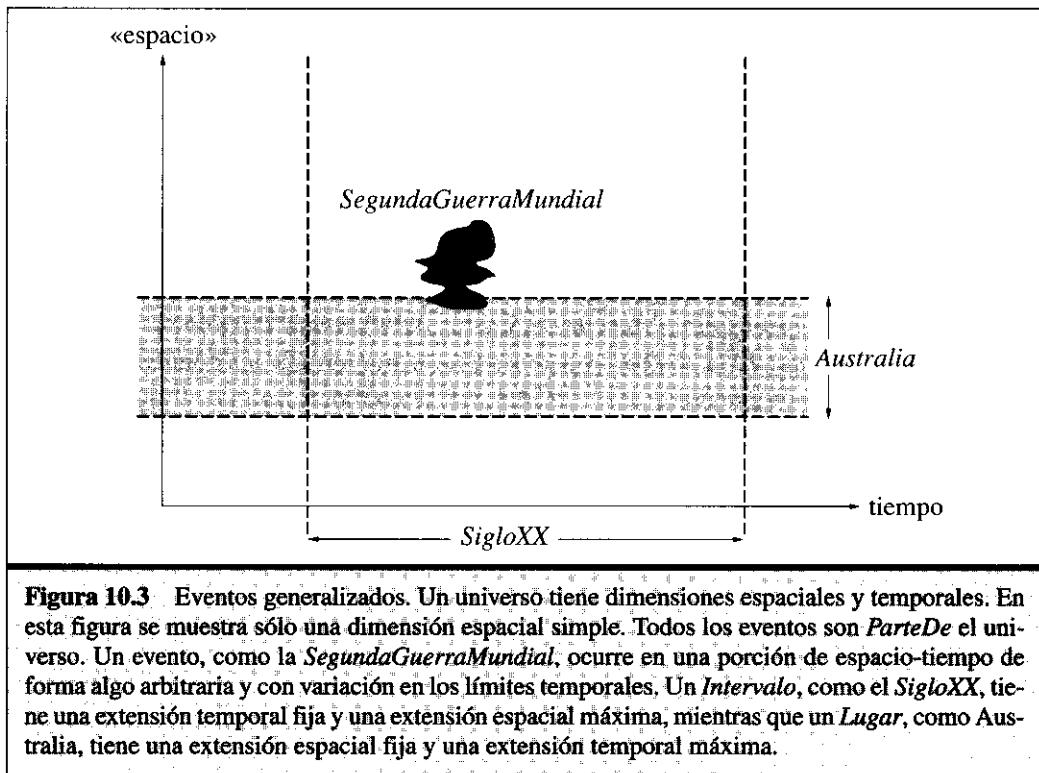
Como cualquier otro tipo de objetos, los eventos se pueden agrupar en categorías. Por ejemplo, *SegundaGuerraMundial* pertenece a la categoría de *Guerras*. Para decir que una guerra civil ocurrió en Inglaterra en 1640, se podría decir:

$\exists w \ w \in \text{GuerrasCiviles} \wedge \text{SubEvento}(w, 1640) \wedge \text{En}(\text{Localización}(w), \text{Inglaterra})$

La noción de categoría de eventos responde a una pregunta que se obvió cuando se describieron los efectos de los axiomas en el Apartado 10.3: ¿A qué se refieren los térmi-

⁴ Algunos físicos que estudian la teoría de la cadena, hablan de 10 dimensiones o más, y algunos hablan de un mundo discreto, pero una representación espacio-temporal continua de cuatro dimensiones es adecuada para el propósito de hacer razonamientos basados en el sentido común.

⁵ Nótese que *SubEvento* es un caso especial de la relación *ParteDe*, siendo también transitiva y reflexiva.



nos lógicos como $Ir([1, 1], [1, 2])$? ¿son eventos? La respuesta, posiblemente sorprendente, es *no*. Se puede entender esto considerando un plan con dos acciones «idénticas» como

$$[Ir([1, 1], [1, 2]), Ir([1, 2], [1, 1]), Ir([1, 1], [1, 2])]$$

En este plan, $Ir([1, 1], [1, 2])$ no puede ser el nombre de un evento porque hay *dos eventos diferentes* que ocurren en tiempos diferentes. En su lugar, $Ir([1, 1], [1, 2])$ es el nombre de una *categoría* de eventos (representando todos los eventos mediante los cuales el agente va desde $[1, 1]$ a $[1, 2]$). El plan de tres fases establece que ocurrirán instancias de estos tres eventos.

Nótese que esta es la primera vez que se han visto nombres de categorías formados por términos complejos en vez de por constantes. Esto no presenta nuevas dificultades. De hecho, se puede usar la estructura de los argumentos como una ventaja. Mediante la eliminación de argumentos se crea una categoría más general.

$$Ir(x, y) \subseteq IrA(y) \quad Ir(x, y) \subseteq IrDesde(x)$$

De modo similar, se pueden añadir argumentos para crear categorías más específicas. Por ejemplo, para describir acciones de otros agentes, se puede añadir un argumento que represente al agente. Por lo tanto, el decir que Shankar voló ayer desde Nueva York a Nueva Delhi, se podría escribir:

$$\exists e \ e \in Volar(Shankar, NuevaYork, NuevaDelhi) \wedge SubEvento(e, Ayer)$$

La forma de esta fórmula es tan común que se creará una abreviación para ella: $E(c, i)$ significará que un elemento de una categoría de eventos c es un subevento del evento o intervalo i :

$$E(c, i) \Leftrightarrow \exists e \ e \in c \wedge SubEvento(e, i)$$

Por lo tanto tenemos que,

$$E(Volar(Shankar, NuevaYork, NuevaDelhi), Ayer)$$

Procesos

EVENTOS DISCRETOS

Los eventos que se han visto hasta ahora son los que se denominan **eventos discretos** (con una estructura definida). El viaje de Shankar tiene un comienzo, un punto intermedio y un final. Si se interrumpe a medio camino, el evento será diferente (no será un viaje desde Nueva York a Nueva Delhi, sino un viaje desde Nueva York a algún lugar en Europa). Por otro lado, la categoría de eventos representados por *Volando(Shankar)* tiene una cualidad diferente. Si se considera un pequeño intervalo en el vuelo de Shankar, por ejemplo el tercer segmento de 20 minutos (mientras que él espera ansiosamente por una segunda maleta de cacahuates), este evento forma parte todavía de *Volando(Shankar)*. De hecho, esto es cierto para cualquier subintervalo.

PROCESOS

EVENTOS LÍQUIDOS

Las categorías de eventos que cumplen esta propiedad se llaman categorías de **procesos** o categorías de **eventos líquidos**. Todos los subintervalos de un proceso son también miembros de la misma categoría de procesos. Mediante la misma notación que se empleó en los eventos discretos, se puede afirmar que, por ejemplo, Shankar iba volando en algún momento del día de ayer:

$$E(Volando(Shankar), Ayer)$$

Frecuentemente se necesitará expresar que algún proceso se realizó *durante* cierto intervalo, en vez de que sólo se realizó durante cierto subintervalo. Para ello, utilizamos el predicado T :

$$T(Trabajando(Stuart), HoyHoraDelAlmuerzo)$$

$T(c, i)$ significa que cierto evento de tipo c se produjo exactamente durante el intervalo i , es decir, el evento comienza y termina al mismo tiempo que el intervalo.

SUSTANCIAS TEMPORALES

SUSTANCIAS ESPACIALES

ESTADOS

La diferencia entre eventos líquidos y no líquidos es análoga a la diferencia entre sustancias o materia, y objetos individuales. De hecho, algunos han llamado **sustancias temporales** a los eventos líquidos, mientras que cosas como la mantequilla son **sustancias espaciales**.

De la misma forma que se describen los procesos de cambio continuo, los eventos líquidos puede describir procesos de cambio no continuo. Son llamados **estados**. Por ejemplo, «Encontrándose Shankar en Nueva York» es una categoría de estados que se denotan como *En(Shankar, NuevaYork)*. Para decir que él estuvo en Nueva York todo el día, se escribiría

$$T(En(Shankar, NuevaYork), Hoy)$$

CÁLCULO DE FLUJOS

Se pueden formar eventos y estados más complejos combinando las primitivas. Esta aproximación recibe el nombre de **cálculo de flujos**. El cálculo de flujos se refiere a la combinación de flujos, no a flujos individuales. Ya se ha visto una forma de representar el evento correspondiente al momento en que dos cosas suceden al mismo tiempo, denominado *Ambos*(e_1, e_2). En cálculo de flujos, esto se abrevia normalmente con la notación infija $e_1 \circ e_2$. Por ejemplo, para decir que alguien caminó y masticaba chicle al mismo tiempo, se puede escribir

$$\exists p, i \ (p \in Gente) \wedge T(\text{Caminar}(p) \circ \text{MasticarChicle}(p), i)$$

La función « \circ » es conmutativa y asociativa, como la conjunción lógica. Se pueden definir funciones análogas a las de disyunción y negación, pero se debe tener cuidado (hay dos formas razonables de interpretar la disyunción). Cuando se dice «el agente o estaba caminando o estaba masticando chicle durante los dos últimos minutos» se puede querer decir que el agente estaba haciendo una de las dos acciones durante todo el intervalo de tiempo, o quizás que estaba alternando entre las dos acciones. Se utilizará *UnaDe* y *Cualquiera* para indicar estas dos posibilidades. La Figura 10.4 representa los eventos complejos.

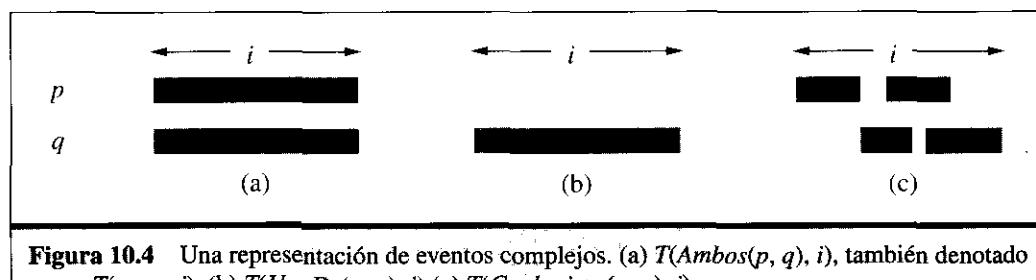


Figura 10.4 Una representación de eventos complejos. (a) $T(\text{Ambos}(p, q), i)$, también denotado como $T(p \circ q, i)$, (b) $T(\text{UnoDe}(p, q), i)$ (c) $T(\text{Cualquiera}(p, q), i)$.

Intervalos

El tiempo es importante para cualquier agente que realice acciones, y se ha realizado mucho trabajo para la representación de intervalos de tiempo. Aquí se consideran dos clases: momentos e intervalos extendidos. La diferencia es que sólo los *momentos tienen* duración cero:

$$\begin{aligned} &\text{Partición}(\{\text{Momentos}, \text{IntervalosExtendidos}\}, \text{Intervalos}) \\ &i \in \text{Momentos} \Leftrightarrow \text{Duración}(i) = \text{Segundos}(0) \end{aligned}$$

A continuación se creará una escala de tiempo y se asociarán puntos con momentos en esa escala, de lo cual se obtiene el concepto de tiempo absoluto. La escala de tiempo es arbitraria. Se medirá en segundos y se dirá que el momento de medianoche (GMT) del 1 de enero de 1900 tiene un valor de tiempo igual a cero. Las funciones *Comienzo* y *Fin* seleccionan los momentos más tempranos y tardíos en un intervalo, y la función *Tiempo* determina el punto en la escala de tiempo para un momento. La función *Duración* devuelve la diferencia entre el momento final y el inicial.

$\text{Intervalo}(i) \Rightarrow \text{Duración}(i) = (\text{Tiempo}(\text{Fin}(i)) - \text{Tiempo}(\text{Comienzo}(i)))$
 $\text{Tiempo}(\text{Comienzo}(1900DC)) = \text{Segundos}(0)$
 $\text{Tiempo}(\text{Comienzo}(2001DC)) = \text{Segundos}(3187324800)$
 $\text{Tiempo}(\text{Fin}(2001DC)) = \text{Segundos}(3218860800)$
 $\text{Duración}(2001DC) = \text{Segundos}(31536000)$

Para facilitar la lectura de los números, también se introduce una función *Fecha*, la cual recibe seis argumentos (horas, minutos, segundos, día, mes y año) y devuelve un punto en el tiempo:

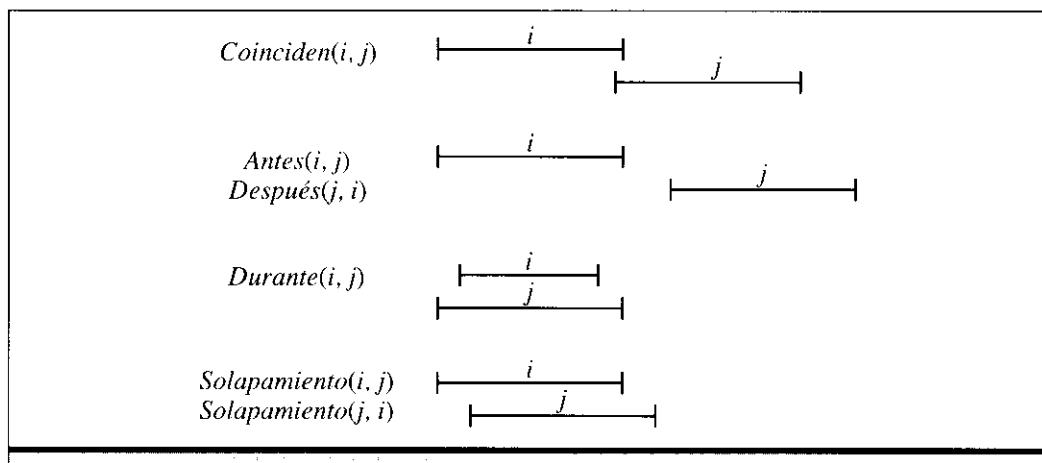
$\text{Tiempo}(\text{Fecha}(2001DC)) = \text{Fecha}(0, 0, 0, 1, \text{Ene}, 2001)$
 $\text{Fecha}(0, 20, 21, 24, 1, 1995) = \text{Segundos}(3000000000)$

Dos intervalos *Coincidan* si el momento final del primero es igual al momento de inicio del segundo. También es posible definir predicados como *Antes*, *Después*, *Durante* y *Solapamiento* únicamente en términos de *Coincidan*, pero es más intuitivo definirlos en términos de puntos en la escala de tiempo (véase la Figura 10.5 para una representación gráfica).

$$\begin{aligned}
\text{Coinciden}(i, j) &\Leftrightarrow \text{Tiempo}(\text{Fin}(i)) = \text{Tiempo}(\text{Inicio}(j)) \\
\text{Antes}(i, j) &\Leftrightarrow \text{Tiempo}(\text{Fin}(i)) < \text{Tiempo}(\text{Inicio}(j)) \\
\text{Después}(j, i) &\Leftrightarrow \text{Antes}(i, j) \\
\text{Durante}(i, j) &\Leftrightarrow \text{Tiempo}(\text{Inicio}(j)) \leq \text{Tiempo}(\text{Inicio}(i)) \\
&\quad \wedge \text{Tiempo}(\text{Fin}(i)) \leq \text{Tiempo}(\text{Fin}(j)) \\
\text{Solapamiento}(i, j) &\Leftrightarrow \exists k \text{ Durante}(k, i) \wedge \text{Durante}(k, j)
\end{aligned}$$

Por ejemplo, para decir que el reinado de Isabel II siguió al de Jorge VI, y el reinado de Elvis se solapó con la década de los 50, se puede escribir lo siguiente:

$\text{Antes}(\text{ReinadoDe}(\text{IsabelII}), \text{ReinadoDe}(\text{JorgeVI}))$
 $\text{Solapamiento}(\text{AñosCincuenta}, \text{ReinadoDe}(\text{Elvis}))$
 $\text{Inicio}(\text{AñosCincuenta}) = \text{Inicio}(1950DC)$
 $\text{Fin}(\text{AñosCincuenta}) = \text{Fin}(1959DC)$

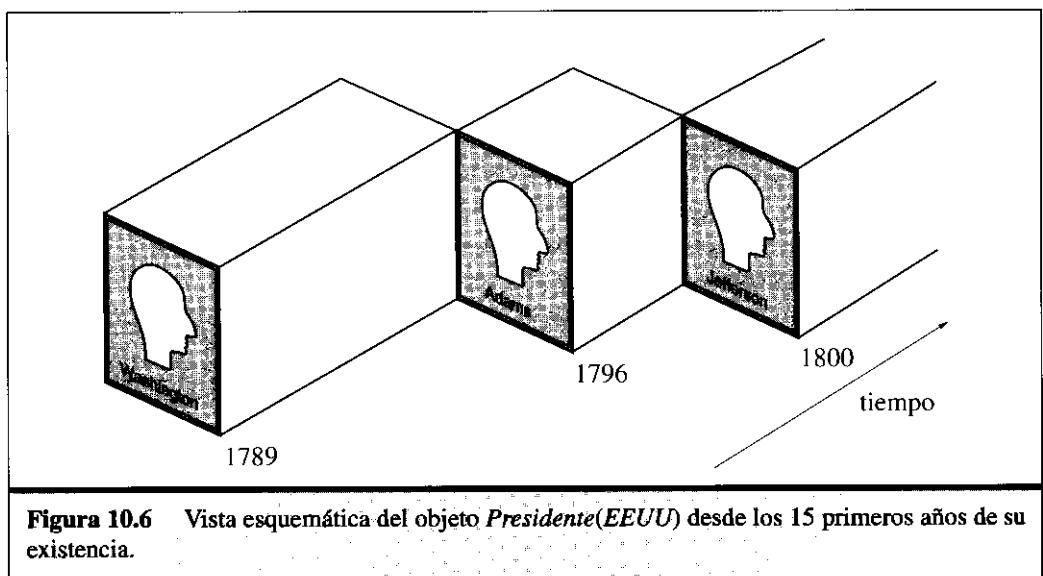


Flujos y objetos

Se ha mencionado que los objetos físicos pueden ser vistos como eventos generalizados, en el sentido de que un objeto físico es un trozo de espacio-tiempo. Por ejemplo, Estados Unidos puede ser visto como un evento que empezó, digamos que en 1776 con la unión de los 13 estados, y que se encuentra hoy en día en progreso como unión de 50 estados. Se puede describir el cambio en las propiedades de Estados Unidos usando estados de flujo. Por ejemplo, se puede decir que en un momento determinado de 1999, la población ascendía a 271 millones:

$$E(\text{Población}(EEUU, 271000000), 1999DC)$$

Otra propiedad de Estados Unidos que cambia cada cuatro u ocho años, a menos que no suceda algún contratiempo, es el presidente. Se podría proponer que el término lógico *Presidente(EEUU)* denotara un objeto diferente para distintos momentos en el tiempo. Desgraciadamente esto no es posible, porque un término denota exactamente a un objeto en una estructura de modelos determinada (el término *Presidente(EEUU)* no puede denotar diferentes objetos dependiendo del valor de t , puesto que la ontología utilizada mantiene los índices de tiempo separados de los flujos). La única posibilidad es que *Presidente(EEUU)* denote un objeto simple que consista en gente diferente en diferentes momentos en el tiempo. Es el objeto que representa a George Washington desde 1789 hasta 1796, John Adams desde 1796 hasta 1800 y así sucesivamente, como muestra la Figura 10.6.



Para expresar que George Washington fue presidente durante 1790, se puede escribir

$$T(\text{Presidente}(EEUU)) = \text{George Washington}, 1790DC$$

Sin embargo, se necesita ser cuidadoso. En la sentencia anterior, el símbolo igual «=» debe ser un símbolo de función en lugar de un operador lógico estándar. La interpreta-

ción no es que *George Washington* y *Presidente(EEUU)* son idénticos desde un punto de vista lógico en 1790. La identidad lógica no es algo que pueda cambiar a lo largo del tiempo. La identidad lógica existe entre los subeventos de cada objeto que están definidos por el período 1790.

No confundir el objeto físico *George Washington* con una colección de átomos. *George Washington* no es idéntico desde un punto de vista lógico a *ninguna* colección específica de átomos, porque el conjunto de átomos que lo forman varía considerablemente en el tiempo. Él tiene un tiempo de vida corto y cada átomo tiene un tiempo de vida muy corto. En conjunto, todos se cruzan durante algún período de tiempo, durante el cual, el tiempo de vida del átomo es *ParteDe* George y a partir de ahí evolucionan por separado.

10.4 Eventos mentales y objetos mentales

Los agentes que se han construido hasta ahora tienen creencias y pueden deducir nuevas creencias. Pero por ahora ninguno de ellos tiene conocimiento *sobre* creencias o *sobre* deducción. En dominios de un solo agente simple, el conocimiento sobre el propio conocimiento y los procesos de razonamiento son útiles para controlar la inferencia. Por ejemplo, si uno es consciente de que no puede conocer nada acerca de la geografía rumana, entonces no empleará un enorme esfuerzo computacional para tratar de calcular el camino más corto desde Arad a Bucarest. Uno también puede razonar sobre su propio conocimiento, para construir planes que permitan cambiarlo (por ejemplo comprando un mapa de Rumanía). En dominios multiagente, es importante para un agente razonar acerca de los estados mentales de los otros agentes. Por ejemplo, un oficial de policía rumano posiblemente conocerá la mejor forma de llegar a Bucarest, por lo que el agente podría pedirle ayuda.

En esencia, lo que se precisa es un modelo de los objetos mentales que existen en la cabeza (o en la base de conocimiento) de alguien y los procesos mentales para manipular esos objetos mentales. El modelo debe ser fiel a la realidad, pero no tiene por qué ser detallado. No se necesita ser capaz de predecir cuántos milisegundos le llevará a un agente en concreto realizar una deducción, ni tampoco será necesario predecir qué neuronas se dispararán cuando a un animal se le presenta un determinado estímulo visual. Llegará con concluir que el oficial de policía rumano podría informar de cómo llegar a Bucarest si conoce el camino y se da cuenta de que alguien está perdido.

Una teoría formal de creencias

Se comenzó trabajando con las relaciones existentes entre agentes y «objetos mentales» (relaciones como *Cree*, *Conoce* y *Desea*). Las relaciones de este tipo se denominan **actitudes de proposición**, porque describen una actitud que un agente puede tomar hacia una proposición. Supóngase que Lois cree algo, es decir, *Cree(Lois, x)*. ¿Qué tipo de cosa es *x*? Obviamente *x* no puede ser una sentencia lógica. Si *Vuela(Supermán)* es una sentencia lógica, no se puede decir *Cree(Lois, Vuela(Supermán))*, porque sólo los términos (no las sentencias) pueden ser argumentos de los predicados. Pero si *Vuela* es una fun-

REFICACIÓN

ción, entonces *Vuela(Supermán)* es un candidato para ser un objeto mental, y *Cree* puede ser una relación entre una gente y un flujo proposicional. Convertir una proposición en un objeto, se conoce con el nombre de **reificación**⁶.

Esto parece proporcionar lo que se necesita: la capacidad para que un agente razonne sobre las creencias de los agentes. Desafortunadamente, hay un problema con esta aproximación: si Clark y Supermán son uno al mismo tiempo (es decir, *Clark = Supermán*) entonces los vuelos de Clark y Supermán son el mismo y pertenecen a la misma categoría de eventos, es decir, *Vuela(Clark) = Vuela(Supermán)*. Por lo tanto, se debe concluir que si Lois cree que Supermán puede volar, él también cree que Clark puede volar, *incluso si ella no cree que Clark es Supermán*. Es decir,

$$(Supermán = Clark) \models (Cree(Lois, Vuela(Supermán)) \Leftrightarrow Cree(Lois, Vuela(Clark)))$$

Una de las interpretaciones de lo anterior es válida: Lois cree que cierta persona, que a veces se llama Clark, puede volar. Pero existe otra interpretación que es errónea: si se le pregunta a Lois ¿puede volar Clark?, indudablemente contestaría que no. Los objetos reificados y los eventos funcionan bien para la primera interpretación de *Cree*, pero para la segunda interpretación sería necesario reificar las *descripciones* de tales objetos y eventos, de manera que Clark y Supermán puedan ser descripciones distintas (aunque se refieran al mismo objeto).

Desde un punto de vista técnico, a la propiedad de sustituir libremente un término por otro igual se le denomina **transparencia referencial**. En lógica de primer orden, todas las relaciones tienen transparencia referencial. Convendría definir *Cree* (y las otras actitudes de proposición) como relaciones cuyo segundo argumento es **opaco** referencialmente, es decir, no es posible sustituir el segundo argumento por un término igual sin cambiar el significado.

Existen dos formas de lograr esto. La primera es usar una lógica diferente, denominada **lógica modal**, en la cual las actitudes de proposición como *Cree* y *Sabe* son **operadores modales** que son referencialmente opacos. Esta aproximación se trata en la sección de notas históricas. La segunda aproximación que se desea lograr, es conseguir una opacidad efectiva con un lenguaje transparente referencialmente usando una **teoría sintáctica** de objetos mentales. Esto significa que los objetos mentales serán representados por **cadenas**. El resultado es un modelo rudimentario de la base de conocimiento de un agente, que consiste en cadenas que representan sentencias que son creídas por el agente. Una cadena no es más que un término complejo denotado por una lista de símbolos, así, el evento *Vuela(Clark)*, puede ser representado por la lista de caracteres $[V, u, e, l, a, (, C, l, a, r, k,)]$, que se abreviará como «*Vuela(Clark)*». La teoría sintáctica incluye un **axioma de cadena única** que establece que las cadenas son idénticas si y sólo si están formadas por caracteres idénticos. Por lo tanto, incluso si *Clark = Supermán*, se seguirá teniendo que «*Clark*» \neq «*Supermán*».

Ahora sólo falta proponer sintaxis, semántica y teoría de la demostración correspondientes al lenguaje de representación de la cadena, como en el caso del Capítulo 7.

⁶ El término «reificación» viene de la palabra Latina *res*, o cosa. John McCarthy propuso el término «cosificación», pero nunca llegó a ser popular.

La diferencia es que ahora hay que definirlas a todas mediante lógica de primer orden. Se comenzará por definir *Den* como aquella función que correlaciona una cadena con el objeto que denota ésta, y a *Nombrar* como la función que correlaciona un objeto con una cadena que es el nombre de una constante que denota al objeto. Por ejemplo la denotación de «*Clark*» como «*Supermán*» es el objeto referido mediante el símbolo constante *HombreDeAcero*, y el nombre de tal objeto puede ser tanto «*Supermán*» como «*Clark*», o alguna otra constante, por ejemplo, « X_{11} ».

$$\begin{aligned} \text{Den}(\text{«Clark»}) &= \text{HombreDeAcero} \wedge \text{Den}(\text{«Supermán»}) = \text{HombreDeAcero} \\ \text{Nombrar}(\text{HombreDeAcero}) &= \text{«}X_{11}\text{»} \end{aligned}$$

El siguiente paso consiste en definir las reglas de inferencia de los agentes lógicos. Por ejemplo, digamos que se desea que el agente lógico sea capaz de efectuar un *Modus Ponens*: si cree p y también cree $p \Rightarrow q$, entonces también cree q . El primer ensayo de cómo escribir este axioma es:

$$\text{AgenteLógico}(a) \wedge \text{Cree}(a, p) \wedge \text{Cree}(a, p \Rightarrow q) \Rightarrow \text{Cree}(a, q)$$

Pero es incorrecto, puesto que aunque la cadena « $p \Rightarrow q$ » contiene las letras « p » y « q », no tiene nada que ver con las cadenas que corresponden a los valores de las variables p y q . La formulación correcta es

$$\text{AgenteLógico}(a) \wedge \text{Cree}(a, p) \wedge \text{Cree}(a, \text{Concatenar}(p, \text{«} \Rightarrow \text{»}, q)) \Rightarrow \text{Cree}(a, q)$$

donde *Concatenar* es una función de las cadenas que concatena entre sí a sus elementos. Se abreviará *Concatenar*(p , « \Rightarrow », q) como « $p \Rightarrow q$ ». Es decir, la aparición de x dentro de una cadena es **no acotada**, con el significado de que se debe sustituir la variable x por su valor. Los programadores de Lisp identificarán lo anterior como un operador de cita reversiva, y los programadores de Perl lo reconocerán como una interpolación de una variable de tipo \$.

Una vez añadidas las otras reglas de inferencia además de Modus Ponens, se podrá responder a preguntas como «si un agente lógico conoce estas premisas, ¿podrá llegar a esa conclusión?». Además de las reglas de inferencia normales, también son necesarias algunas reglas específicas de creencia. Por ejemplo, la siguiente regla afirma que si un agente lógico cree algo, entonces también cree que lo cree:

$$\text{AgenteLógico}(a) \wedge \text{Cree}(a, p) \Rightarrow \text{Cree}(a, \text{«} \text{Cree}(\text{Nombrar}(a), p) \text{»})$$

A partir de ahora, de acuerdo con el axioma, un agente puede deducir cualquier consecuencia de su creencia de un modo infalible. Esto se llama **omnisciencia lógica**. Se han realizado numerosos intentos para tratar de definir agentes racionales limitados, los cuales pueden realizar un número limitado de deducciones en un tiempo también limitado. Nada es completamente satisfactorio, pero esta formulación permite un rango muy restringido de predicciones sobre agentes limitados.

NO ACOTADA

OMNISCIENCIA LÓGICA

Conocimiento y creencia

Las relaciones entre creencia y conocimiento se han estudiado de forma exhaustiva en Filosofía. Es conocida la afirmación en el sentido de que el conocimiento no es sino una creencia válida demostrada. Por ejemplo, si usted cree algo, y si esto último realmente

es cierto, y tiene la prueba de ello, entonces usted lo sabe. La demostración es indispensable, pues evitará que usted afirme «Sé que el resultado de lanzar esta moneda al aire será una cara» y que sólo sea cierto la mitad de las veces.

Considérese que *Sabe(a, p)* significa que el agente *a* *sabe que* la preposición *p* es cierta. También es posible definir otras clases de conocimiento. Por ejemplo, a continuación se establece una definición de «saber si»:

$$\text{SabeSi}(a, p) \Leftrightarrow \text{Sabe}(a, p) \vee \text{Sabe}(a, \langle\!\langle \neg p \rangle\!\rangle)$$

Continuando con el ejemplo, Lois sabe si Clark puede volar tanto si ella sabe que Clark puede volar, como si no.

El concepto de «saber que» es más complicado. Uno puede sentirse tentado a aceptar que un agente sabe que el número de teléfono de Bob es una *x* para la cual *x* = *Número Teléfono(Bob)*. Pero esto no es suficiente, porque el agente podría conocer que Alice y Bob tienen el mismo número de teléfono (es decir, *NúmeroTeléfono(Bob)* = *NúmeroTeléfono(Alice)*), pero si el número de teléfono de Alice es desconocido, esto no sería de mucha ayuda. Una definición mejor de «saber que» sería que el agente debe saber cierta *x* que es una cadena de dígitos y que es el número de Bob:

$$\begin{aligned} \text{SabeQue}(a, \langle\!\langle \text{NúmeroTeléfono}(b) \rangle\!\rangle) &\Leftrightarrow \\ \exists x \text{ Sabe}(a, \langle\!\langle \underline{x} = \text{NúmeroTeléfono}(b) \rangle\!\rangle) \wedge x \in \text{SecuenciaDeDígitos} \end{aligned}$$

Desde luego que para otro tipo de preguntas los criterios serán disjuntos, dependiendo de lo que se considere como una respuesta aceptable. En el caso de la pregunta «¿cuál es la capital de Nueva York?», una respuesta aceptable sería un nombre propio, «Albani», no algo como «la ciudad en donde está la cámara legislativa del estado». Para ello, se hará que *SaberQue* sea una relación de tres partes: empleará un agente, un término y un predicado que integren la respuesta válida. Por ejemplo, se podría tener lo siguiente:

$$\begin{aligned} \text{SabeQue}(\text{Agente}, \langle\!\langle \text{Capital(NuevaYork)} \rangle\!\rangle, \text{NombrePropio}) \\ \text{SabeQue}(\text{Agente}, \langle\!\langle \text{NúmeroTeléfono}(Bob) \rangle\!\rangle, \text{SecuenciaDeDígitos}) \end{aligned}$$

Conocimiento, tiempo y acción

En la mayoría de las situaciones reales, un agente tratará con creencias (las suyas propias o las de otros agentes) que cambian en el tiempo. El agente también tendrá que hacer planes que involucren cambios de sus propias creencias, como comprar un mapa para averiguar cómo llegar a Bucarest. Como con otros predicados, se podrá reificar *Cree* y hablar sobre creencias que ocurren en un período de tiempo. Por ejemplo, para decir que Lois cree hoy que Supermán puede volar, se escribirá

$$T(\text{Cree}(Lois, \langle\!\langle \text{Volar(Supermán)} \rangle\!\rangle), Hoy)$$

Si el objeto de creencia es una proposición que puede cambiar en el tiempo, entonces también se puede describir utilizando el operador *T* dentro de la cadena. Por ejemplo, Lois podría creer hoy que Supermán pudo volar ayer:

$$T(\text{Cree}(Lois, \langle\!\langle T(\text{Volar(Supermán)}, Ayer) \rangle\!\rangle), Hoy)$$

PRECONDICIONES DE CONOCIMIENTO**EFFECTOS DE CONOCIMIENTO****VARIABLES DE EJECUCIÓN**

Dada una forma de describir las creencias en el tiempo, se puede usar la maquinaria del cálculo de eventos para realizar planes que incluyan creencias. Acciones que pueden tener **precondiciones de conocimiento** y **efectos de conocimiento**. Por ejemplo, la acción de marcar un número de teléfono tiene la precondición de conocer dicho número, y la acción de buscar el número tiene el efecto de conocer el número. Se puede describir esta última acción a través de la maquinaria del cálculo de eventos:

$$\begin{aligned} &\text{Iniciar}(\text{Buscar}(a, \langle\!\langle \text{NúmeroTeléfono}(b) \rangle\!\rangle), \\ &\quad \text{SabeQue}(a, \langle\!\langle \text{NúmeroTeléfono}(b) \rangle\!\rangle, \text{SecuenciaDeDígitos}), t) \end{aligned}$$

Las **variables de ejecución** se utilizan frecuentemente como notación abreviada para representar planes que unen y utilizan información. Esto es similar a la convención de variable no acotada vista con anterioridad. Por ejemplo, el plan para buscar el número de teléfono de Bob y llamarlo a continuación puede ser escrito como

$$[\text{Buscar}(\text{Agente}, \langle\!\langle \text{NúmeroTeléfono}(Bob) \rangle\!\rangle, \underline{n}), \text{Lamar}(\underline{n})]$$

Aquí, \underline{n} es una variable de ejecución cuyo valor será establecido por la acción *Buscar* y puede ser usado por la acción *Lamar*. Los planes de este tipo ocurren frecuentemente en dominios parcialmente observables. Se verán ejemplos en la sección siguiente y en el Capítulo 12.

10.5 El mundo de la compra por Internet

En esta sección se codificará algún conocimiento relacionado con la compra a través de Internet. Se creará un agente que investigue compras y ayude al comprador a encontrar ofertas de productos en Internet. El comprador da al agente de compra una descripción de un producto, y el agente debe producir un listado de páginas web que oferten ese producto. En algunos casos, la descripción del producto que da el comprador será precisa, como para la *cámara digital Coolpix 995*, y la tarea será encontrar aquellas tiendas que tengan una mejor oferta. En otros casos la descripción será sólo parcialmente especificada, como *una cámara digital por un precio inferior a 300 dólares*, y el agente tendrá que comparar productos diferentes.

El entorno del agente de compra es toda la WWW (*Word Wide Web*), no un entorno de juguete simulado, sino un entorno complejo que evoluciona constantemente y que es usado por millones de personas cada día. La percepción del agente serán las páginas web, pero mientras un usuario de la Web percibe las páginas como vector de puntos en pantalla, el agente de compra recibirá la página como una cadena de caracteres, que consiste en un conjunto de palabras ordinarias intercaladas con comandos de formato en lenguaje HTML. La Figura 10.7 muestra una página web y su cadena de caracteres HTML correspondiente. El problema de la percepción para el agente de compra abarca la extracción de la información útil de percepciones de este tipo.

Claramente, la percepción de páginas web es más fácil que, por ejemplo, la percepción mientras se conduce un taxi en El Cairo. Sin embargo, existen complicaciones en la tarea de percepción en Internet. La página web de la Figura 10.7 es muy simple com-

Almacén Genérico online

Seleccionar una de nuestras trabajadas líneas de productos:

- Computadores
 - Cámaras
 - Libros
 - Vídeos
 - Música
-

```
<h1>Almacén Genérico on-line</h1>
<i>Seleccionar</i> una de nuestras trabajadas líneas de productos:
<ul>
<li> <a href=<http://gen-store.com/compu>>Computadores</a>
<li> <a href=<http://gen-store.com/cama>>Cámaras</a>
<li> <a href=<http://gen-store.com/libr>>Libros</a>
<li> <a href=<http://gen-store.com/vide>>Vídeos</a>
<li> <a href=<http://gen-store.com/musi>>Música</a>
</ul>
```

Figura 10.7 Una página web procedente de un almacén genérico *online* en la forma percibida por una persona (arriba), y la correspondiente cadena HTML percibida por el navegador web o el agente de compra (abajo). En HTML, los caracteres entre *<y>* son etiquetas que especifican cómo se muestra la página. Por ejemplo, la cadena *<i>Seleccionar</i>* implica establecer una fuente en cursiva, mostrar la palabra *Seleccionar* y finalizar con el uso de la fuente cursiva. Un identificador de página como *http://gen-store.com/libr* se denomina un **localizador de recursos uniforme o URL**. La etiqueta *ancla* implica la creación de un hipervínculo a la *url* con el **texto del enlace ancla**.

parada con webs reales dedicadas a la compra de artículos, que incluyen cookies, Java, Javascript, Flash, protocolos de exclusión de robots, código HTML con incorrecciones, ficheros de sonido, clips de películas y texto que aparece sólo como imágenes JPEG. Un agente que sea capaz de tratar con *todo* Internet es casi tan complejo como un robot que pueda mover un objeto en el mundo real. El planteamiento se centrará en un agente que ignore la mayoría de estas complicaciones.

La primera tarea del agente es encontrar ofertas relevantes de productos (se verá posteriormente cómo seleccionar la mejor oferta de las relevantes). Sea *peticIÓN* la descripción del producto que el usuario teclea (por ejemplo, «portátiles»). Entonces una página es una oferta relevante para *peticIÓN* si la página es relevante y la página es efectivamente una oferta. También se seguirá la pista de la URL asociada con la página:

$$\begin{aligned} \text{OfertaRelevante}(\text{página}, \text{url}, \text{petición}) \Leftrightarrow \\ \text{Relevante}(\text{página}, \text{url}, \text{petición}) \wedge \text{Oferta}(\text{página}) \end{aligned}$$

Una página con una comparativa de los portátiles más modernos debería ser relevante, pero si no proporciona un mecanismo para comprar, no es una oferta. Por ahora, se dirá que

una página es una oferta si contiene la palabra «compra» o «precio» en un enlace HTML o en un formulario. En otras palabras, si la página contiene una cadena de la forma: «[...compra...](#)» entonces es una oferta. También podría aparecer «precio» en vez de «compra» o usar la etiqueta «form» en vez de la etiqueta «a». Se puede escribir un axioma para esto:

$$\begin{aligned}
 Oferta(página) &\Leftrightarrow EnEtiqueta(\langle a \rangle, str, página) \vee \\
 &\quad EnEtiqueta(\langle form \rangle, str, página) \\
 &\quad \wedge (En(\langle comprar \rangle, str) \\
 &\quad \vee (En(\langle precio \rangle, str)). \\
 EnEtiqueta(etiqueta, cadena, página) &\Leftrightarrow En(\langle < \rangle + etiqueta + str + \\
 &\quad \langle < / \rangle + etiqueta, página). \\
 En(subcadena, cadena) &\Leftrightarrow \exists i \ str[i : i + Longitud(subcadena)] \\
 &\quad = subcadena.
 \end{aligned}$$

Ahora se necesita encontrar páginas relevantes. La estrategia es empezar en la página de inicio de un almacén general y considerar todas las páginas a las que se puede llegar siguiendo los enlaces relevantes⁷. El agente tendrá conocimiento acerca de varios almacenes, por ejemplo:

$$\begin{aligned}
 Amazon &\in AlmacenesOnLine \wedge PáginaDeInicio(Amazon, \langle amazon.com \rangle). \\
 Ebay &\in AlmacenesOnLine \wedge PáginaDeInicio(Ebay, \langle ebay.com \rangle). \\
 GenStore &\in AlmacenesOnLine \wedge PáginaDeInicio(GenStore, \langle gen-store.com \rangle)
 \end{aligned}$$

Los almacenes clasifican sus productos en categorías de productos, y proporcionan enlaces a las categorías principales desde sus páginas de inicio. Las categorías descendientes se pueden localizar siguiendo los enlaces relevantes, y finalmente se accederá a las ofertas. En otras palabras, una página es relevante para la petición si se puede localizar a través de enlaces relevantes de categorías que parten de la página principal del almacén, y entonces se seguirán uno o más enlaces a ofertas de productos:

$$\begin{aligned}
 Relevante(página, url, petición) &\Leftrightarrow \\
 &\exists \text{almacén, } \text{inicio} \text{ } \text{almacén} \in AlmacenesOnLine \wedge \\
 &PáginaDeInicio(almacén, inicio) \\
 &\wedge \exists url_2 \text{ } EnlacesRelevante(inicio, url_2, petición) \wedge Enlace(url_2, url) \\
 &\wedge página = ObtenerPágina(url)
 \end{aligned}$$

Aquí el predicado *Enlace(de, a)* significa que existe un hipervínculo desde la URL *de* hasta la URL *a* (véase Ejercicio 10.13). Para definir qué se tiene en cuenta en *Enlaces-Relevante*, no se podrá seguir cualquier hipervínculo, sino aquellos cuyo texto indique que es relevante para la petición del producto. Para ello, se utilizarán *TextoDeEnlace(de, a, texto)* para significar que hay un enlace entre *de* hacia *a* con el texto especificado por *texto*. Una cadena de enlaces entre dos URL's, *inicio* y *fin*, es relevante para una descripción *d* si el texto del hipervínculo de cada enlace contiene un nombre de categoría

⁷ Una alternativa a la estrategia de seguir los enlaces es la de usar los resultados de un motor de búsqueda de Internet. La tecnología que hay detrás de un buscador de Internet, recuperación de información, será tratada en la Sección 23.2.

relevante para d . La existencia de la cadena de enlaces en sí misma se determina por una definición recursiva con el enlace vacío ($\text{inicio} = \text{fin}$) como caso base:

$$\begin{aligned} \text{EnlacesRelevantes}(\text{inicio}, \text{fin}, \text{petición}) &\Leftrightarrow (\text{inicio} = \text{fin}) \\ &\vee (\exists u, \text{texto} \text{ } \text{TextodeEnlace}(\text{inicio}, u, \text{texto}) \\ &\quad \wedge \text{NombreCategoríaRelevante}(\text{petición}, \text{texto}) \\ &\quad \wedge \text{EnlacesRelevante}(u, \text{fin}, \text{petición})) \end{aligned}$$

Ahora se puede definir lo que debe contener texto para que sea un $\text{NombreCategoríaRelevante}$ para petición . Esto se realiza utilizando el predicado $\text{Nombre}(s, c)$, que dice que la cadena s es un nombre de categoría c (por ejemplo, se puede afirmar que $\text{Nombre}(\langle\text{portátiles}\rangle, \text{ComputadoresPortátiles})$). Algunos ejemplos más del predicado Nombre aparecen en la Figura 10.8(b). A continuación, se definirá el concepto de relevancia. Supóngase que la petición es «portátiles». Entonces $\text{NombreCategoríaRelevante}(\text{petición}, \text{texto})$ es cierto cuando una de las siguientes afirmaciones es cierta:

- El texto y la petición hacen referencia a la misma categoría (por ejemplo, «computadores portátiles» y «portátiles»).
- El texto hace referencia a una supercategoría como «computadores».
- El texto hace referencia a una subcategoría como «notebooks ultraligeros».

La definición lógica de $\text{NombreCategoríaRelevante}$ es la siguiente:

$$\begin{aligned} \text{NombreCategoríaRelevante}(\text{petición}, \text{texto}) &\Leftrightarrow \\ \exists c_1, c_2 \text{ } \text{Nombre}(\text{petición}, c_1) \wedge \text{Nombre}(\text{texto}, c_2) \wedge (c_1 \sqsubseteq c_2 \vee c_2 \sqsubseteq c_1) & (10.1) \end{aligned}$$

A parte de esto, el texto del hipervínculo es irrelevante porque se refiere a una categoría fuera de esta línea, como «computadores mainframe» o «césped & jardín».

Para seguir enlaces relevantes, es esencial tener una jerarquía rica en categorías de productos. La parte superior de esta jerarquía debería ser parecida a la mostrada en la Figura 10.8(a). No será viable realizar un listado de todas las categorías de compras posibles, porque a un comprador le podrían surgir nuevos deseos y los fabricantes crearían nuevos productos para satisfacer a los clientes (¿calentadores de rótula eléctricos?). Sin

$\text{Libros} \subset \text{Productos}$	$\text{Nombre}(\langle\text{libros}\rangle, \text{Libros})$
$\text{GrabacionesMusicales} \subset \text{Productos}$	$\text{Nombre}(\langle\text{música}\rangle, \text{GrabacionesMusicales})$
$\text{CDsMusicales} \subset \text{GrabacionesMusicales}$	$\text{Nombre}(\langle\text{CDs}\rangle, \text{CDsMusicales})$
$\text{CintasMúsica} \subset \text{GrabacionesMusicales}$	$\text{Nombre}(\langle\text{cintas}\rangle, \text{CintasMúsica})$
$\text{Electrónica} \subset \text{Productos}$	$\text{Nombre}(\langle\text{electrónica}\rangle, \text{Electrónica})$
$\text{CámarasDigitales} \subset \text{Electrónica}$	$\text{Nombre}(\langle\text{cámaras digitales}\rangle, \text{CámarasDigitales})$
$\text{EquipamientoEstéreo} \subset \text{Electrónica}$	$\text{Nombre}(\langle\text{estéreos}\rangle, \text{EquipamientoEstéreo})$
$\text{Computadores} \subset \text{Electrónica}$	$\text{Nombre}(\langle\text{computadores}\rangle, \text{Computadores})$
$\text{ComputadoresPortátiles} \subset \text{Computadores}$	$\text{Nombre}(\langle\text{portátiles}\rangle, \text{ComputadoresPortátiles})$
$\text{ComputadoresSobreMesa} \subset \text{Computadores}$	$\text{Nombre}(\langle\text{sobremesa}\rangle, \text{ComputadoresSobreMesa})$
...	...
(a)	(b)

Figura 10.8 (a) Taxonomía de categorías de productos. (b) Palabras usadas para esas categorías.

embargo, una ontología de aproximadamente 1.000 categorías sería una herramienta muy útil para la mayoría de los compradores.

Además de la jerarquía de productos en sí misma, también es necesario tener un vocabulario rico para los nombres de las categorías. La vida sería mucho más sencilla si hubiera una correspondencia uno a uno entre categorías y cadenas de caracteres para nombrarlas. Ya se ha comentado el problema de la **sinonimia** (dos nombres para la misma categoría, como «computadores portátiles» y «portátiles»). También existe el problema de la **ambigüedad** (un nombre para dos o más categorías diferentes). Por ejemplo, si se añade la sentencia

Nombre(«CDs», CertificadosDeDepósito)

a la base de conocimiento de la Figura 10.8(b), entonces «CDs» nombrará a dos categorías diferentes.

Sinonimia y ambigüedad pueden causar un incremento significativo en el número de caminos que el agente debe seguir, y puede algunas veces hacer difícil el determinar si una página dada es realmente relevante. Un problema mucho más serio es la existencia de un amplio rango de descripciones que un usuario puede teclear, así como nombres de categorías que un almacén puede emplear. Por ejemplo, un enlace podría decir «portátil» cuando la base de conocimiento tiene sólo «portátiles», o el usuario podría preguntar por «un computador que pueda encajar en una mesa portátil de un asiento de clase económica en un Boeing 737». Resulta imposible enumerar a priori todas las formas de nombrar a una categoría, por lo que el agente tendrá que ser capaz de realizar un razonamiento adicional para determinar si la relación *Nombre* es cierta. En el peor caso, esto requiere una comprensión completa del lenguaje natural, un tema que se postergará hasta el Capítulo 22. En la práctica, unas pocas reglas simples (como el permitir que «portátil» case con una categoría llamada «portátiles») sigue un largo camino. El Ejercicio 10.15 propone el desarrollo de un conjunto de reglas después de hacer algunas investigaciones en los almacenes *online*.

Dadas las definiciones lógicas de los párrafos precedentes y las bases de conocimiento apropiadas sobre categorías de producto y convenciones de nombres, ¿se está preparado para aplicar un algoritmo de inferencia y obtener un conjunto de ofertas relevantes para la petición?, ¡no exactamente! El elemento que falta es la función *ObtenerPágina(url)*, que se refiere a la página HTML de una URL determinada. El agente no tiene el contenido de cada URL en su base de conocimiento, ni tampoco reglas explícitas para deducir qué tipo de contenidos encontrará. En su lugar, se puede hacer que el procedimiento HTTP correcto se ejecute siempre que un subobjetivo necesite la función *ObtenerPágina*. De esta forma, se hace creer al motor de inferencia que la Web entera se encuentra dentro de la base de conocimiento. Esto es un ejemplo de la técnica general denominada **acoplamiento funcional**, según el cual predicados y funciones particulares pueden ser manejadas por métodos de propósito específico.

Comparación de ofertas

Asúmase que el proceso de razonamiento de la sección precedente ha producido un conjunto de páginas con ofertas para la petición sobre «portátiles». Para comparar esas ofer-

ENVOLTORIO

tas, el agente debe extraer la información relevante (precio, velocidad, tamaño del disco duro, peso, etc.) de las páginas recuperadas. Esto puede ser una tarea difícil en páginas web reales por las razones que se comentaron con anterioridad. Un modo común de abordar este problema es usar algunos programas de tipo **envoltorio** para extraer información de la página. La tecnología para la extracción de información se trata en la Sección 23.3. Por ahora, se asumirá que los programas de tipo envoltorio existen, y cuando se les proporciona una página determinada y una base de conocimiento, añaden aserciones a la base de conocimiento. Normalmente se aplicará una jerarquía de programas de tipo envoltorio a una página: uno muy general para extraer fechas y precios, uno más específico para extraer atributos de productos relacionados con los computadores y, si es necesario, uno específico para el sitio web particular y que conozca el formato concreto del almacén. Dada una página en el sitio de gen-store.com con el texto

YVM ThinkBook 970. Nuestro Precio: 1449.00\$

seguido de varias especificaciones técnicas, se deseará un programa envoltorio para extraer información como la siguiente:

$$\exists lc, oferta \quad lc \in \text{ComputadoresPortátiles} \wedge oferta \in \text{OfertaDeProductos} \wedge \\ TamañoPantalla(lc, \text{Pulgadas}(14)) \wedge \text{TipoPantalla}(lc, \text{ColorLCD}) \wedge \\ TamañoMemoria(lc, \text{Megabytes}(512)) \wedge \text{VelocidadCPU}(lc, \text{GHz}(2.4)) \wedge \\ \text{ProductoOfrecido}(oferta, lc) \wedge \text{Almacén}(oferta, \text{GenStore}) \wedge \\ URL(oferta, \langle\!\langle \text{genstore.com/comps/34356.html} \rangle\!\rangle) \wedge \\ \text{Precio}(oferta, \$449) \wedge \text{Fecha}(oferta, Hoy)$$

Este ejemplo ilustra varios temas que surgen cuando se toma en serio la tarea de aplicar ingeniería de conocimiento a las transacciones comerciales. Por ejemplo, destacar que el precio es un atributo de una *oferta*, no del producto en sí mismo. Esto es importante porque la oferta en un almacén determinado puede cambiar día a día, incluso para el mismo portátil. Para algunas categorías (como casas y pinturas) el mismo objeto individual puede ser ofrecido simultáneamente por diferentes intermediarios a diferentes precios. Todavía hay más complicaciones que aún no se han manejado, como la posibilidad de que el precio dependa del método de pago y de la clasificación del comprador para ciertos descuentos. Sea como sea, todavía queda un montón de trabajo interesante por hacer.

La tarea final es comparar las ofertas que han sido extraídas. Por ejemplo, considérense estas tres ofertas:

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\$.

C es **dominada** por A, es decir, A es más barato y más rápido, y todas las demás características son iguales. En general, X domina a Y si X tiene un valor mejor en al menos un atributo, y no es peor en los atributos restantes. Pero ni A ni B dominan la una a la otra. Para decidir cuál es mejor, se necesita conocer cómo sopesa el comprador la velocidad de CPU y el precio, frente a la memoria y la capacidad en disco. El tema general de preferencias entre múltiples atributos se aborda en la Sección 16.4. Por ahora, el agen-

te de compra simplemente devolverá una lista con todas las ofertas no dominantes que cumplen con la descripción del comprador. En este ejemplo, tanto *A* como *B* no son dominantes. Nótese que esta salida se basa en la asunción de que todo el mundo prefiere precios más baratos, procesadores más rápidos y más capacidad de almacenamiento. Algunos atributos, como el tamaño de pantalla en un portátil, dependen de las preferencias particulares del usuario (portabilidad vs. visibilidad). Para éstas, el agente de compra preguntará al usuario.

El agente de compra que se ha descrito aquí es muy simple y se pueden realizar muchos refinamientos. De todos modos, tiene suficiente capacidad para ser utilizado como asistente para la compra, siempre y cuando se disponga de conocimiento correcto acerca del domino específico. Debido a su construcción declarativa, es fácilmente extensible para aplicaciones más complejas. El punto principal en esta sección es mostrar que es necesaria una representación del conocimiento (en particular de la jerarquía de productos), para la construcción de un agente de este tipo, y que una vez que se dispone de conocimiento en esta forma, no es muy complicado diseñar un agente basado en conocimiento que haga el resto.

10.6 Sistemas de razonamiento para categorías

Se han visto las categorías como bloques de construcción primarios para cualquier esquema de representación del conocimiento a gran escala. Existen dos familias de sistemas íntimamente relacionadas: (i) las **redes semánticas** proporcionan una ayuda gráfica para visualizar una base de conocimiento, así como algoritmos eficientes para inferir propiedades de un objeto con base en su pertenencia a una categoría; (ii) la **lógica descriptiva** proporciona un lenguaje formal para construir y combinar definiciones de categorías, así como algoritmos eficientes para decidir las relaciones de subconjunto y superconjunto entre categorías.

Redes semánticas

GRAFOS EXISTENCIALES

En 1909, Charles Peirce propuso una notación gráfica de nodos y arcos denominada **grafos existenciales** que él denominó «la lógica del futuro». Entonces empezó un debate de larga duración entre los defensores de la «lógica» y los defensores de las «redes semánticas». Desafortunadamente, el debate oscureció el hecho de que las redes semánticas (por lo menos aquellas con un concepto de semántica bien definido) son una forma de lógica. La notación que proporcionan las redes semánticas para cierta clase de sentencias es a menudo más conveniente, pero si se deja de lado la «interfaz humana», los conceptos base (objetos, relaciones, cuantificación, etc.) son los mismos.

Existen diversas variantes de las redes semánticas, pero todas son capaces de representar objetos individuales, categorías de objetos y relaciones entre objetos. Una notación gráfica común visualiza objetos o nombres de categorías en óvalos o cajas, y los conecta con arcos etiquetados. Por ejemplo, la Figura 10.9 tiene un enlace *MiembroDe* entre *Mary* y *PersonaFemenina*, correspondiente a la aserción lógica *Mary* ∈ *Persona*.

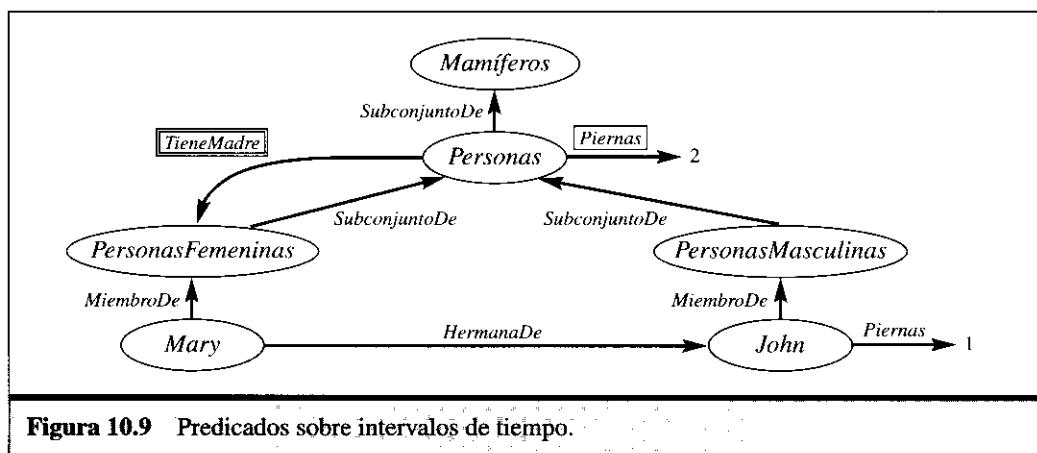


Figura 10.9 Predicados sobre intervalos de tiempo.

Femenina. De igual modo, el enlace *HermanaDe* entre *Mary* y *John* corresponde a la aserción *HermanaDe(Mary, John)*. Se pueden conectar categorías usando enlaces *SubconjuntoDe*. Es tan divertido dibujar burbujas y flechas que uno puede dejarse llevar. Por ejemplo, se conoce que todas las personas tienen una madre de sexo femenino, por lo tanto ¿se puede dibujar un enlace *TieneMadre* desde *Personas* hasta *PersonasFemeninas*? La respuesta es no, porque *TieneMadre* es una relación entre una persona y su madre, y las categorías no tienen madre⁸. Por esta razón, se ha usado una notación especial (en enlace con caja doble) en la Figura 10.9. Este enlace expresa que

$$\forall x \ x \in \text{Personas} \Rightarrow [\forall y \ \text{TieneMadre}(x, y) \Rightarrow y \in \text{PersonasFemeninas}]$$

También se podría querer afirmar que las personas tienen dos piernas, por lo que,

$$\forall x \ x \in \text{Personas} \Rightarrow \text{Piernas}(x, 2)$$

Como antes, se necesita ser cuidadoso para no afirmar que una categoría tiene piernas. La caja con línea simple de la Figura 10.9 se usa para especificar propiedades de cada miembro de una categoría.

La notación de la red semántica hace muy conveniente la utilización de razonamiento basado en **herencia** del tipo del introducido en la Sección 10.2. Por ejemplo, por el hecho de ser persona, *Mary* hereda la propiedad de tener dos piernas. Por lo tanto, para saber cuántas piernas tiene *Mary*, el algoritmo de herencia sigue el enlace *MiembroDe* desde *Mary* hasta la categoría a la cual pertenece y entonces continúa por el enlace *SubconjuntoDe* hasta encontrar la categoría en la cual existe un enlace etiquetado con el recuadro *Piernas* (en este caso, la categoría *Personas*). La simplicidad y eficiencia de este mecanismo de inferencia, comparado con el teorema de la prueba lógica, ha sido uno de los más atractivos para las redes semánticas.

⁸ Varios sistemas desarrollados hace tiempo fallaron en distinguir entre propiedades de miembros de una categoría y propiedades de la categoría como un todo. Esto puede llevar directamente a inconsistencias, como apuntó Drew McDermott (1976) en su artículo *Artificial Intelligence Meets Natural Stupidity*. Otro problema común fue el uso del enlace *EsUn* para los subconjuntos y las relaciones de pertenencia, en correspondencia con el uso inglés: «un gato es un mamífero» y «Fifi es un gato». Véase el Ejercicio 10.25 para más información sobre este punto.

HERENCIA MÚLTIPLE

La herencia se complica cuando un objeto puede pertenecer a más de una categoría, o cuando una categoría puede ser un subconjunto de varias categorías. Esto se conoce con el nombre de **herencia múltiple**. En estos casos, el algoritmo de herencia puede encontrar dos o más valores en conflicto que resuelvan la pregunta. Por esta razón, no se permite herencia múltiple en algunos lenguajes de **programación orientada a objetos** (POO), como Java, que usa la herencia para la jerarquía de clases. La herencia múltiple es permitida comúnmente en las redes semánticas, pero se pospondrá esta discusión hasta la Sección 10.7.

ENLACES INVERSOS

Otra forma común de herencia es el uso de **enlaces inversos**. Por ejemplo, *TieneHermana* es el inverso de *HermanaDe*, lo que significa que

$$\forall p, s \text{ } TieneHermana(p, s) \Leftrightarrow HermanaDe(s, p)$$

Esta sentencia puede ser expresada en una red semántica si el enlace es **reificado** (es decir, transformado en objetos). Por ejemplo, se podría tener el objeto *HermanaDe*, conectado con un enlace *inverso* mediante *TieneHermana*. Dada la pregunta acerca de quién es *HermanaDe* John, el algoritmo de inferencia puede descubrir que *TieneHermana* es el inverso de *HermanaDe* y por lo tanto, puede responder a la pregunta siguiendo el enlace *TieneHermana* desde *John* hasta *Mary*. Sin la información inversa, sería necesario comprobar cada persona de sexo femenino para ver si esa persona tiene un enlace *HermanaDe* hacia John. Esto es así debido a que las redes semánticas proporcionan indexación directa sólo para objetos, categorías y los enlaces que salen de ellos. Utilizando el vocabulario de la lógica de primer orden, es como si la base de conocimiento estuviera indexada sólo por el primer argumento de cada predicado.

El lector puede haberse dado cuenta de un inconveniente obvio de la notación de las redes semánticas, en comparación con la lógica de primer orden: el hecho de que los enlaces entre burbujas representan sólo relaciones *binarias*. Por ejemplo, la sentencia *Vuela(Shankar, NuevaYork, NuevaDelhi, Ayer)* no se puede expresar directamente en una red semántica. Sin embargo, se *puede* conseguir el efecto de las relaciones *n*-arias reificando la proposición como si fuera un evento (véase la Sección 10.3) perteneciente a una categoría de eventos apropiada. La Figura 10.10 muestra la estructura de la red semántica para este evento particular. Nótese que la restricción de las relaciones binarias fuerza a la creación de una ontología rica de conceptos reificados. De hecho, mucha de la ontología desarrollada en este capítulo se originó en sistemas de redes semánticas.

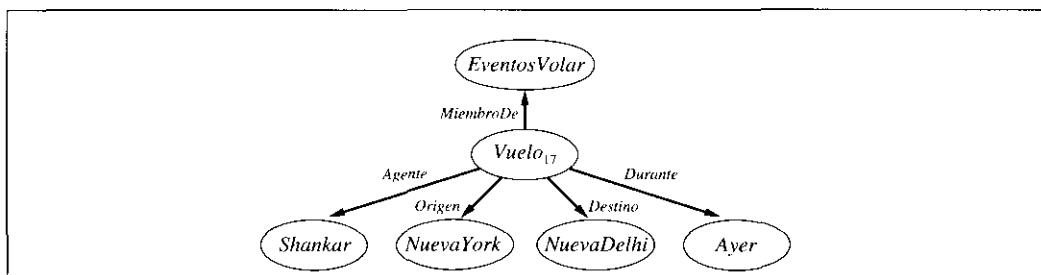


Figura 10.10 Una fragmento de una red semántica que muestra la representación de la aserción lógica *Vuela(Shankar, NuevaYork, NuevaDelhi, Ayer)*.

La reificación de proposiciones hace posible representar cualquier terreno, sentencia atómica con función libre expresada en lógica de primer orden, utilizando la notación de las redes semánticas. Ciertas clases de sentencias universalmente cuantificadas se pueden expresar usando enlaces inversos y las fechas con cajas de borde simple o doble aplicadas a las categorías, pero esto aún deja un largo camino plagado de lógica de primer orden. La negación, disyunción, símbolos de función anidados y cuantificación existencial aún no se pueden representar. Ahora es *possible* extender la notación para hacerla equivalente a la lógica de primer orden (como en los grafos existenciales de Peirce o las redes semánticas particionadas de Hendrix (1975)), pero haciendo esto se pierde una de las principales ventajas de las redes semánticas, como es la simplicidad y la transparencia del proceso de inferencia. Los diseñadores pueden construir una red grande y seguir teniendo una buena idea sobre qué preguntas serán eficientes, porque (a) es fácil visualizar los pasos que el procedimiento de inferencia seguirá y (b) en algunos casos, el lenguaje de consulta es tan simple que las consultas difíciles no se pueden representar. En aquellos casos donde el poder expresivo sea demasiado limitado, algunas redes semánticas proporcionan **acoplamiento procedural** para llenar el vacío. El acoplamiento procedural es una técnica por la cual una consulta sobre (o a veces una afirmación de) una cierta relación, implica una llamada a un procedimiento especial diseñado para esa relación, en lugar de la utilización del algoritmo de inferencia general.

**VALORES
POR DEFECTO**

SOBREESCRITURA

Uno de los aspectos más importantes de las redes semánticas es su habilidad para representar **valores por defecto** para las categorías. Examinando la Figura 10.9 con detenimiento, uno se da cuenta de que John tiene una pierna en lugar de tener dos, puesto que él es una persona y todas las personas tienen dos piernas. En una base de conocimiento estrictamente lógica esto sería una contradicción, pero en una red semántica la afirmación de que todas las personas tienen dos piernas tiene sólo el sentido de valor por defecto. Es decir, se asume que una persona tiene dos piernas a menos que sea contradicho por información más específica. La semántica por defecto se refuerza de forma natural por el algoritmo de inferencia, porque utiliza enlaces hacia arriba desde el objeto en sí mismo (John en este caso), y para tan pronto como encuentra un valor. Se dice que el valor por defecto es **sobrescrito** por un valor más específico. Nótese que se podría haber sobrescrito el valor por defecto para el número de piernas a través de la creación de una categoría de *PersonasConUnaPierna*, un subconjunto de *Personas* de la cual John es miembro.

Se puede mantener la semántica lógica de modo estricto en la red si se dice que la aserción *Personas* posee una excepción para John:

$$\forall x \ x \in \text{Personas} \wedge x \neq \text{John} \Rightarrow \text{Piernas}(x, 2)$$

Para una red *constante*, lo anterior es semánticamente adecuado, pero será mucho menos concisa que la notación de la red si existen un montón de excepciones. Sin embargo, esta aproximación falla para una red que será actualizada con más aserciones (realmente se quiere decir que cualquier persona con una sola pierna es también una excepción). La Sección 10.7 profundiza en este punto y en el razonamiento por defecto general.

Lógica descriptiva

LÓGICA DESCRIPTIVA

SUBSUNCIÓN

CLASIFICACIÓN

La sintaxis de la lógica de primer orden está diseñada para hacer más fácil el afirmar cosas sobre objetos. La **lógica descriptiva** se basa en notaciones que están diseñadas para hacer más fácil describir definiciones y propiedades de categorías. Los sistemas de lógica descriptiva han evolucionado desde las redes semánticas, en respuesta a las presiones para formalizar el significado de la red y retener al mismo tiempo, el énfasis en la estructura taxonómica como principio organizacional.

Las principales tareas de inferencia para la lógica descriptiva son la **subsunción** (comprobar si una categoría es un subconjunto de otra a través de la comparación de sus definiciones) y la **clasificación** (comprobar si un objeto pertenece a una categoría). Algunos sistemas también incluyen **consistencia** de la definición de una categoría (si el criterio de pertenencia puede ser satisfecho lógicamente).

El lenguaje CLASSIC (Borgida *et al.*, 1989) es un ejemplo típico de lógica descriptiva. En la Figura 10.11 se muestra la sintaxis de las descripciones de CLASSIC⁹. Por ejemplo, para expresar que los solteros son adultos que no están casados, se escribiría

$$\text{Soltero} = Y(\text{NoCasado}, \text{Adulto}, \text{Masculino})$$

El equivalente de lo anterior en lógica de primer orden sería

$$\text{Soltero}(x) \leftrightarrow \text{NoCasado}(x) \wedge \text{Adulto}(x) \wedge \text{Masculino}(x)$$

Obsérvese que la lógica descriptiva realmente permite efectuar operaciones lógicas directas en los predicados, en vez de tener que crear primero oraciones que se unen mediante conectores. Toda descripción en CLASSIC se puede expresar mediante lógica de primer orden, pero algunas descripciones resultan más directas expresadas en CLASSIC. Por ejemplo, para describir el conjunto de hombres que por lo menos tengan tres hijos, estén desempleados y cuya esposa es doctora, y que además tengan como máximo dos hijas, ambas profesoras de Física o Matemáticas se escribiría

$$\begin{aligned} &Y(\text{Hombre}, \text{AlMenos}(3, \text{Hijos}), \text{AlMenos}(2, \text{Hijas}), \\ &\quad \text{Todos}(\text{Hijos}, Y(\text{Desempleado}, \text{Casado}, \text{Todos}(\text{Esposa}, \text{Doctor}))), \\ &\quad \text{Todos}(\text{Hijas}, Y(\text{Profesor}, \text{Satisfice}(\text{Departamento}, \text{Física}, \text{Matemáticas})))) \end{aligned}$$

Se deja como ejercicio traducir lo anterior a lógica de primer orden.

Quizás el aspecto más importante de la lógica descriptiva sea el énfasis que se pone en la maleabilidad de la inferencia. Los problemas se resuelven mediante su descripción y cuestionando si pueden ser subsumidos mediante una de las varias categorías posibles de solución. En los sistemas promedio de lógica de primer orden, muchas veces es imposible predecir cuál va a ser el tiempo necesario para hallar la solución. Frecuentemente, se deja al usuario diseñar la representación que permita evitar aquellos conjuntos de oraciones, que probablemente sean las causantes de que el sistema emplee varias semanas en resolver un problema. El énfasis en la lógica descriptiva, por otra parte, es el de ga-

⁹ Obsérvese que el lenguaje *no* permite limitarse a afirmar que un concepto o una categorías es subconjunto de otro. Lo anterior es deliberativo: la subsunción entre categorías debe obtenerse a partir de ciertos aspectos de las descripciones de las categorías. De no ser así, seguramente algo está faltando en las descripciones.

<i>Concepto</i>	\rightarrow	Objeto <i>NombreDelConcepto</i> <i>Y(Concepto, ...)</i> <i>Todo(NombreRol, Concepto)</i> <i>AlMenos(Entero, NombreRol)</i> <i>Casi(Entero, NombreRol)</i> <i>Cumple(NombreRol, NombreIndividual, ...)</i> <i>IgualQue(Ruta, Ruta)</i> <i>UnoDe(NombreIndividual, ...)</i> <i>[NombreDelRol, ...]</i>
<i>Ruta</i>	\rightarrow	

Figura 10.11 La sintaxis de descripciones en un subconjunto del lenguaje CLASSIC.

rantizar que la prueba de subsunción pueda ser resuelta en un tiempo que sea una función polinómica del tamaño de la descripción del problema¹⁰.

En principio, lo anterior parecería maravilloso hasta que uno se da cuenta que entraña dos consecuencias: los problemas difíciles no se pueden formular, o ¡requieren de descripciones de vastas extensiones exponenciales! A pesar de lo anterior, la docilidad de los resultados obtenidos arrojan luz sobre qué tipos de estructuras causan problemas, y esto ayuda al usuario a comprender el comportamiento de las diversas representaciones.

Por ejemplo, la lógica descriptiva por lo general carece de *negación* y *disyunción*. Ambas fuerzan a los sistemas lógicos de primer orden a producir análisis de caso de tipo exponencial, si es que se desea garantizar la completitud. Por la misma razón se les ha excluido de Prolog. CLASSIC acepta sólo una variante limitada de la disyunción, las estructuras *SatisfaceQue* y *UnaDe*, que permiten aplicar la disyunción en individuos específicamente designados, pero no en las descripciones. En el caso de las descripciones disyuntivas, las definiciones anidadas dan lugar fácilmente a una cantidad exponencial de rutas alternativas, en las que una categoría puede subsumir a otra.

10.7 Razonamiento con información por defecto

En la sección precedente, se ha visto un ejemplo simple de una asercción con un valor por defecto: las personas tienen dos piernas. Este valor por defecto puede ser sobrescrito con información más específica, como que Long John Silver tiene una pierna. Se ha visto que el mecanismo de herencia en las redes semánticas implementa la sobrescritura de valores por defecto de forma simple y natural. En esta sección, se estudiarán los valores por defecto de forma más general, con la vista puesta hacia la comprensión de la *semántica* de los valores por defecto, en vez de proporcionar tan sólo un mecanismo procedimental.

¹⁰ CLASSIC proporciona una prueba de subsunción eficiente en la práctica, pero el peor caso de ejecución es exponencial.

Mundos abiertos y cerrados

Suponga que estaba mirando en un tablón de noticias en el Departamento de Informática de una Universidad y vio una nota que decía, «Se ofrecerán los siguientes cursos: CS 101, CS 102, CS 106, EE 101». ¿Cuántos cursos se ofrecerán?, si su respuesta es «cuatro», estará en consonancia con un sistema de base de datos típico. Dada una base de datos relacional con el equivalente de las cuatro aserciones

$$\text{Curso}(\text{CS}, 101), \text{Curso}(\text{CS}, 102), \text{Curso}(\text{CS}, 106), \text{Curso}(\text{EE}, 101), \quad (10.2)$$

La consulta SQL `count * from Curso` devuelve 4. Por otro lado, un sistema basado en lógica de primer orden respondería «algo entre uno e infinito», no «cuatro». La razón es que las aserciones *Curso* no niegan la posibilidad de que se impartan otros cursos no mencionados, ni que los cursos que se mencionan sean diferentes los unos de los otros.

Este ejemplo muestra que los sistemas de bases de datos y las convenciones de los humanos para la comunicación, difieren de la lógica de primer orden en al menos dos puntos. En primer lugar, las bases de datos (y la gente) asume que la información proporcionada es *completa*, por lo que las sentencias atómicas para las que no se dispone de una aserción que diga que son ciertas, se consideran falsas. Esto es lo que se conoce con el nombre de **asunción del mundo cerrado**, o CWA (*Closed-World Assumption*). En segundo lugar, normalmente se asume que nombres distintos hacen referencia a objetos distintos. Esto se conoce con el nombre de **asunción de nombres únicos**, o UNA (*Unique Names Assumption*), introducido primero en el contexto de nombres de acciones en la Sección 10.3.

La lógica de primer orden no asume estas convenciones, y por lo tanto, necesita ser más explícita.

Para decir que *sólo* se ofertan los cuatro cursos distintos, se podría escribir:

$$\begin{aligned} \text{Curso}(d, n) \Leftrightarrow [d, n] = [\text{CS}, 101] \vee [d, n] = [\text{CS}, 102] \\ \vee [d, n] = [\text{CS}, 106] \vee [d, n] = [\text{EE}, 101] \end{aligned} \quad (10.3)$$

ASUNCIÓN DEL MUNDO CERRADO

COMPLETITUD

FORMA NORMAL DE CLARK

La Ecuación (10.3) se llama de **completitud**¹¹ de 10.2. En general, la completitud contendrá una definición (una sentencia si y sólo si) para cada predicado, y cada definición contendrá una disyunción por cada cláusula definida, teniendo el predicado a la cabeza¹². En general, la completitud se construye como sigue:

1. Unir todas las cláusulas con el mismo nombre de predicado (*P*) y la misma cardinalidad (*n*).
2. Transformar cada cláusula en **forma normal de Clark**: reemplazar

$$P(t_1 \dots, t_n) \leftarrow \text{Cuerpo}$$

donde t_i son términos, con

$$P(v_1 \dots, v_n) \leftarrow \exists w_1 \dots w_m [v_1 \dots, v_n] = [t_1 \dots, t_n] \wedge \text{Cuerpo}$$

¹¹ Algunas veces llamada «Completitud de Clark» debido a su creador, Keith Clark.

¹² Nótese que esta es también la forma de los axiomas estado-sucesor vistos en la Sección 10.3.

donde v_i son variables inventadas recientemente y w_i son las variables que aparecen en la cláusula original. Usar el mismo conjunto de v_i para cada cláusula. Esto proporciona el siguiente conjunto de cláusulas

$$\begin{aligned} P(v_1 \dots, v_n) &\leftarrow B_1 \\ &\vdots \\ P(v_1 \dots, v_n) &\leftarrow B_k \end{aligned}$$

3. Combinar todo esto junto a una cláusula disyuntiva:

$$P(v_1 \dots, v_n) \leftarrow B_1 \vee \dots \vee B_k$$

4. Formar la completitud reemplazando el \leftarrow con la equivalencia:

$$P(v_1 \dots, v_n) \Leftrightarrow B_1 \vee \dots \vee B_k$$

La Figura 10.12 muestra un ejemplo de la completitud de Clark para una base de conocimiento con hechos y reglas. Para añadir la asunción de nombres únicos, simplemente se construye la completitud de Clark para la relación de igualdad, donde sólo se conocen los hechos $CS = CS$, $101 = 101$, etc. Esto se deja como un ejercicio.

La asunción del mundo cerrado nos permite encontrar un **modelo mínimo** de una relación. Es decir, se puede encontrar el modelo de la relación *Curso* con los menores elementos. En la Ecuación (10.2) el modelo mínimo de *Curso* tiene cuatro elementos. Alguno menos y existiría una contradicción. Para las bases de conocimiento de Horn, siempre hay un modelo mínimo **único**. Nótese que la asunción de nombres únicos también se aplica a la relación de igualdad: cada término es igual sólo a él mismo. Paradójicamente, esto significa que los modelos mínimos son máximos en el sentido de tener tantos objetos como sea posible.

Cláusulas de Horn	Completitud de Clark
$Curso(CS, 101)$	$Curso(d, n) \Leftrightarrow [d, n] = [CS, 101]$
$Curso(CS, 102)$	$\vee [d, n] = [CS, 102]$
$Curso(CS, 106)$	$\vee [d, n] = [CS, 106]$
$Curso(EE, 101)$	$\vee [d, n] = [EE, 101]$
$Curso(EE, i) \leftarrow Entero(i)$	$\vee \exists i [d, n] = [EE, i] \wedge Entero(i)$
$\wedge 101 \leq i \wedge i \leq 130$	$\wedge 101 \leq i \wedge i \leq 130$
$Curso(CS, m + 100) \leftarrow$	$\vee \exists m [d, n] = [CS, m + 100]$
$Curso(CS, m) \wedge 100 \leq m$	$\wedge Curso(CS, m) \wedge 100 \leq m$
$\wedge m < 20$	$\wedge m < 200$

Figura 10.12 La Completitud de Clark usa un conjunto de cláusulas de Horn. El programa original de Horn (izquierda), lista explícitamente cuatro cursos y también afirma que hay clase de matemáticas para cada entero desde 101 hasta 130, y que por cada clase CS en la serie 100 (no titulados) hay una clase correspondiente en la serie 200 (titulados). La completitud de Clark (derecha) establece que no hay otras clases. Con la completitud y la asunción de nombres únicos (así como la definición obvia del predicado *Entero*), se consigue la conclusión deseada de la existencia de exactamente 36 cursos: 30 de matemáticas y seis de tipo CS.

Es posible coger un programa de Horn, generar la completitud de Clark y pasar el resultado obtenido a un demostrador de teoremas para realizar inferencia. Pero a menudo es más eficiente utilizar un mecanismo de inferencia de propósito específico como es Prolog, que tiene implementado en el mecanismo de inferencia las asunciones del mundo cerrado y nombres únicos.

Aquellos sistemas que implementan la asunción del mundo cerrado deben ser cuidadosos sobre el tipo de razonamiento que llevarán a cabo. Por ejemplo, en una base de datos del censo, sería razonable asumir la asunción del mundo cerrado cuando se razona sobre la población actual de las ciudades, pero sería erróneo deducir que ningún niño nacería en el futuro porque la base de datos no contenga entradas con futuros nacimientos. La asunción del mundo cerrado hace que la base de datos sea **completa**, en el sentido de que cualquier consulta atómica se responde positiva o negativamente. Cuando se es genuinamente ignorante acerca de los hechos (como futuros nacimientos), no se puede usar la asunción del mundo cerrado. Un sistema de representación del conocimiento más sofisticado podría permitir al usuario especificar reglas para decidir cuándo aplicar la asunción del mundo cerrado.

Negación como fallo y semánticas de modelado estables

Se ha visto en los Capítulos 7 y 9 que las bases de conocimiento en forma de Horn tienen propiedades computacionalmente deseables. En muchas aplicaciones, sin embargo, el requisito de que cada literal en el cuerpo de una cláusula debe ser positivo es un inconveniente grande. Se podría querer afirmar «Tú puedes salir si no está lloviendo», sin tener que crear predicados como *NoLloviendo*. En esta sección, se verá una forma de añadir una negación explícita a las cláusulas de Horn usando la idea de **negación como fallo**. La idea es que un literal negativo, *no P*, puede ser «demostrado» cierto, en el caso de que la prueba de *P* falle. Esta es una forma de razonamiento por defecto relacionada íntimamente con la asunción del mundo cerrado: se asumirá que algo es falso si no puede ser demostrado que sea cierto. Se usará «*no*» para distinguir la negación como fallo del operador lógico « \neg ».

Prolog permite el operador *no* en el cuerpo de una cláusula. Por ejemplo, considérese el siguiente programa escrito en Prolog:

```
dispositivoIDE ← Dispositivo ∧ no dispositivoSCSI
dispositivoSCSI ← Dispositivo ∧ no dispositivoIDE
controladorSCSI ← dispositivoSCSI
Dispositivo
```

(10.4)

La primera regla dice si se dispone de un disco duro en un computador y no es SCSI, entonces debe ser IDE. La segunda establece que si no es IDE debe ser SCSI. La tercera establece que tener una unidad SCSI implica tener un controlador SCSI y la cuarta afirma que efectivamente se dispone de una unidad. Este programa tiene dos modelos mínimos:

$$\begin{aligned} M_1 &= \{\text{Dispositivo}, \text{dispositivoIDE}\} \\ M_2 &= \{\text{Dispositivo}, \text{dispositivoSCSI}, \text{controladorSCSI}\} \end{aligned}$$

Los modelos mínimos no pueden capturar la semántica específica de programas que emplean negación como fallo. Considérese el programa

$$P \leftarrow \text{no } Q \quad (10.5)$$

Tiene dos modelos mínimos, $\{P\}$ y $\{Q\}$. Desde el punto de vista de la lógica de primer orden esto tiene sentido, puesto que $P \Leftarrow \neg Q$ es equivalente a $P \vee Q$. Pero desde el punto de vista de Prolog es inquietante: Q nunca aparece en la parte izquierda de una flecha, por lo que ¿cómo puede ser un consecuente?

Una alternativa es la idea de **modelo estable**, que es un modelo mínimo donde cada átomo en el modelo tiene una **justificación**: una regla en la cual la cabeza es el átomo y cada literal en el cuerpo se satisface. Técnicamente, se dice que M es un modelo estable de un programa H , si M es el único modelo único del **reducto** de H con respecto a M . El reducto de un programa H se obtiene borrando de H cualquier regla que tenga un literal $\textit{no } A$ en el cuerpo, donde A está en el modelo, y posteriormente borrando cualquier literal negativo en las restantes reglas. Puesto que el reducto de H es ahora una lista de cláusulas de Horn, debe tener un único modelo mínimo.

El reducto de $P \Leftarrow \textit{no } Q$ con respecto a $\{P\}$ es P , que tiene como modelo mínimo $\{P\}$. Por lo tanto $\{P\}$ es un modelo estable. El reducto con respecto a $\{Q\}$ es el programa vacío, que tiene como modelo mínimo $\{ \}$. Por lo tanto $\{Q\}$ no es un modelo estable porque Q no tiene justificación en la Ecuación (10.5). Como otro ejemplo, el reducto de (10.4) con respecto a M_1 es como sigue:

```
dispositivoIDE ← Dispositivo
controladorSCSI ← dispositivoSCSI
Dispositivo
```

Tiene como modelo mínimo M_1 , por lo tanto M_1 es un modelo estable. La **programación de conjunto de respuestas** es una clase de lógica de programación que incorpora negación como fallo y funciona transformando la lógica del programa en forma base, para buscar modelos estables (también conocidos como **conjunto de respuestas**) a través del uso de técnicas de prueba de modelos proposicionales. Por lo tanto, la programación de conjunto de respuestas es un descendiente del Prolog y los demostradores rápidos de satisfacción proposicional como WALKSAT. De hecho, la programación de conjunto de respuestas ha sido aplicada con éxito a problemas de planificación, de igual forma que los demostradores rápidos de satisfacción proposicional. La ventaja de la planificación de conjunto de respuestas sobre otros planificadores es el nivel de flexibilidad: los operadores de planificación y las restricciones se pueden expresar como programas lógicos, y no están restringidos a un formato específico de un formalismo planificador particular. La desventaja es la misma que para otras técnicas proposicionales: si hay muchos objetos en el universo, entonces puede tener lugar una ralentización exponencial.

Circunscripción y lógica por defecto

Se han visto dos ejemplos donde, de forma aparente, el proceso natural de razonamiento violaba la propiedad monotónica de la lógica, que fue demostrada en el Capítulo 7¹³. En el primer ejemplo, una propiedad heredada por todos los miembros de una categoría

¹³ Recuérdese que la monotonía requiere que en todas las sentencias de implicación se mantenga la implicación después de que se añadan nuevas sentencias a la base de conocimiento (KB). Es decir, si $KB \models \alpha$ entonces $KB \wedge \beta \models \alpha$.

ría en una red semántica, podía ser sobreescrita por información específica de una subcategoría. En el segundo ejemplo, los literales negados que derivan de la asunción del mundo cerrado podrían ser sobreescritos por la adición de literales positivos.

La introspección simple sugiere que esos fallos de la propiedad monotónica son generalizados en el razonamiento basado en el sentido común. Parece que los humanos a menudo «llegan a conclusiones». Por ejemplo, cuando uno ve un coche aparcado en la calle, uno tiende normalmente a creer que el coche tiene cuatro ruedas aunque sólo sean visibles tres (si usted cree que la existencia de la cuarta rueda es dudosa, considere también la cuestión de si las tres ruedas visibles son reales o simplemente unas maquetas). Ahora, la teoría de la probabilidad puede proporcionar una conclusión de que la cuarta rueda existe con una alta probabilidad, incluso, para la mayoría de la gente, la posibilidad de que el coche no tenga la cuarta rueda *no surge a menos que se presente alguna nueva evidencia*. Por lo tanto, parece que la conclusión de la cuarta rueda se alcanza *por defecto*, en ausencia de otra razón que lo ponga en duda. Si se detectan nuevas evidencias (por ejemplo, si uno ve al propietario del coche llevando una rueda y se da cuenta de que el coche tiene un gato), entonces la conclusión puede ser negada. Esta clase de razonamiento se dice que es **no monotónico**, porque el conjunto de creencias no crece monotónicamente con el tiempo cuando se dispone de nuevo conocimiento. La **lógica no monotónica** ha sido concebida con nociones modificadas de verdad e implicación para capturar este comportamiento. Se examinarán estos dos tipos de lógica que han sido estudiados de forma extensiva: circunscripción y lógica por defecto.

NO MONOTÓNICO

LÓGICA NO MONOTÓNICA

CIRCUNSCRIPCIÓN

PREFERENCIA DE MODELOS

La **circunscripción** puede ser vista como una versión más potente y precisa de la asunción del mundo cerrado. La idea es especificar predicados particulares que son asumidos como «tan falsos como posibles» (es decir, falso para todo objeto excepto para aquellos para los cuales se conoce que es cierto). Por ejemplo, supóngase que se quiere expresar la regla por defecto de que los pájaros vuelan. Se podría introducir un predicado, por ejemplo $Anormal_1(x)$, y escribir

$$Pájaro(x) \wedge \neg Anormal_1(x) \Rightarrow Vuela(x)$$

Si se especifica que $Anormal_1$ va a ser **circunscrito**, un razonador que utiliza circunscripción asumirá $\neg Anormal_1(x)$ a menos que sea conocida la certeza de $Anormal_1(x)$. Esto proporciona que la conclusión $Vuela(Tweety)$ se pueda alcanzar a partir de la premisa $Pájaro(Tweety)$, pero la conclusión no se mantendrá si se afirma que $Anormal_1(Tweety)$.

La circunscripción puede ser vista como un ejemplo de lógica de **preferencia de modelos**. En este tipo de lógica, una sentencia se deduce (con valor por defecto) si es cierta en todos los modelos *preferidos* de la base de conocimiento, en contraposición a los requerimientos de verdad en *todos* los modelos de la lógica clásica. Por circunscripción, un modelo se prefiere a otro si tiene menos objetos anormales¹⁴. Veamos cómo trabaja esta idea en el contexto de la herencia múltiple en las redes semánticas. El ejemplo estándar para el cual la herencia múltiple genera problemas se llama «El diamante de Nixon».

¹⁴ En la asunción del mundo cerrado, se prefiere un modelo a otro si tiene menos átomos ciertos (es decir, se prefieren modelos que son **mínimos**). Existe una conexión natural entre la asunción del mundo cerrado y las cláusulas definidas de las bases de conocimiento, porque el punto fijo alcanzado mediante razonamiento hacia delante en estas bases de conocimiento es el modelo mínimo único (véase Apartado 7.5).

Surge de la observación de que Richard Nixon es un cuáquero (y por lo tanto pacifista) y un republicano (y por lo tanto no pacifista). Se podría escribir esto como sigue:

$$\begin{aligned} & \text{Republicano}(Nixon) \wedge \text{Cuáquero}(Nixon) \\ & \text{Republicano}(x) \wedge \neg \text{Anormal}_2(x) \Rightarrow \neg \text{Pacifista}(x) \\ & \text{Cuáquero}(x) \wedge \neg \text{Anormal}_3(x) \Rightarrow \text{Pacifista}(x) \end{aligned}$$

Si se circunscriben Anormal_2 y Anormal_3 , no existen modelos preferidos: uno en el que se cumpla $\text{Anormal}_2(Nixon)$ y $\text{Pacifista}(Nixon)$ se considera válido, y otro en el que se cumpla $\text{Anormal}_3(Nixon)$ y $\neg \text{Pacifista}(Nixon)$ también se considera válido. Por lo tanto, el razonador basado en circunscripción permanece agnóstico a la idea de que Nixon es pacifista. Si se desea además afirmar que las creencias religiosas tienen precedencia sobre creencias políticas, se podría usar un formalismo denominado **circunscripción priorizada** para dar preferencia a modelos donde Anormal_3 es minimizado.

CIRCUNSCRIPCIÓN PRIORIZADA

LÓGICA POR DEFECTO

REGLAS POR DEFECTO

La **lógica por defecto** es un formalismo en el cual las **reglas por defecto** se pueden escribir para generar conclusiones contingentes no monotónicas. Una regla por defecto se parece a la siguiente:

$$Pájaro(x) : Vuela(x) / Vuela(x)$$

Esta regla significa que si $Pájaro(x)$ es cierto y si $Vuela(x)$ es consistente con la base de conocimiento, entonces se puede concluir $Vuela(x)$ por defecto. En general, una regla por defecto tiene la forma

$$P : J_1, \dots, J_n / C$$

donde P es un prerequisito, C es la conclusión, y J_i son las justificaciones (si cualquiera de ellas se puede probar como falsa, entonces la conclusión no se puede alcanzar). Cualquier variable que aparece en J_i o C también debe aparecer en P . El ejemplo del diamante de Nixon puede ser representado utilizando lógica por defecto con un hecho y dos reglas por defecto:

$$\begin{aligned} & \text{Republicano}(Nixon) \wedge \text{Cuáquero}(Nixon) \\ & \text{Republicano}(x) : \neg \text{Pacifista}(x) / \neg \text{Pacifista}(x) \\ & \text{Cuáquero}(x) : \text{Pacifista}(x) / \text{Pacifista}(x) \end{aligned}$$

EXTENSIÓN

Para interpretar cuál es el significado de las reglas por defecto, se definirá la noción de una **extensión** o una teoría por defecto como el conjunto máximo de consecuentes de la teoría. Es decir, una extensión S consta de los hechos originales conocidos y un conjunto de conclusiones a partir de las reglas por defecto, de tal modo que no se pueden alcanzar conclusiones adicionales desde S y las justificaciones de cada conclusión por defecto en S son consistentes con S . Como en el caso de los modelos preferidos en la circunscripción, se tienen dos posibles extensiones para el diamante de Nixon: uno en el que él es un pacifista y otro en el que no lo es. Existen esquemas priorizados en los que a algunas reglas por defecto se les puede dar precedencia sobre otras, permitiendo resolver algunas ambigüedades.

Desde 1980, cuando se propuso por primera vez la lógica no monotónica, se han hecho muchos progresos para entender sus propiedades matemáticas. Comenzando en los últimos años de la década de los 90, los sistemas prácticos basados en programación ló-

gica han demostrado ser prometedores como herramientas para la representación del conocimiento. Sin embargo, aún hay cuestiones por resolver. Por ejemplo, si «los coches tienen cuatro ruedas» es falso, ¿qué significa el que lo tengamos en nuestra base de conocimiento? ¿Cuál es el conjunto apropiado de reglas por defecto que debemos tener? Si no se puede decidir, para cada regla por separado, si debe pertenecer a la base de conocimiento, entonces existe un problema serio de no modularidad. Por último, ¿cómo se pueden usar las creencias que tienen un valor por defecto para tomar decisiones? Probablemente esta sea la cuestión más difícil de resolver para el razonamiento por defecto. Las decisiones a menudo implican cambios, y por lo tanto, se necesita comparar la fuerza en la creencia de la salida de varias acciones. En aquellos casos donde se toma repetidamente la misma clase de decisiones, es posible interpretar las reglas por defecto como afirmaciones que establecen un «umbral de probabilidad». Por ejemplo, la regla por defecto «mis frenos están siempre correctos» realmente significa «La probabilidad de que mis frenos estén correctos, en ausencia de otra información, es suficientemente alta para que la decisión óptima sea conducir sin revisarlos». Cuando el contexto para la toma de decisiones cambia (por ejemplo, cuando uno conduce un camión pesado muy cargado por una carretera de montaña empinada), el valor por defecto de repente no es el apropiado, incluso cuando no haya evidencia que sugiera que los frenos estén mal. Estas consideraciones han llevado a algunos investigadores a considerar cómo encapsular el razonamiento por defecto en la teoría de la probabilidad.

10.8 Sistemas de mantenimiento de verdad

La sección anterior argumentaba que la mayoría de las inferencias logradas por un sistema de representación del conocimiento tendrán sólo valores por defecto, en vez de ser absolutamente ciertas. Inevitablemente, algunos de los hechos inferidos serán erróneos y tendrán que ser retractados debido a la aparición de nueva información. Este proceso de llama **revisión de la creencia**¹⁵. Supóngase que una base de conocimiento *KB* contiene una sentencia *P* (puede ser una conclusión por defecto generada por un algoritmo de razonamiento hacia delante, o simplemente una afirmación incorrecta) y que se desea ejecutar *Decir(KB, ¬P)*. Para evitar el crear una contradicción, primero se ejecutará *Retractar(KB, P)*. Esto parece sencillo, sin embargo, el problema surge si alguna sentencia *adicional* fue inferida utilizando *P* y afirmada en la base de conocimiento. Por ejemplo, la implicación $P \Rightarrow Q$ podría haber sido utilizada para añadir *Q*. La solución «obvia» (retractar todas las sentencias que se infirieron utilizando *P*) falla, porque esas sentencias podrían tener otras justificaciones a demás de *P*. Por ejemplo, si también están en la base de conocimiento *R* y $R \Rightarrow Q$, entonces *Q* no debe ser borrado. Los sistemas de mantenimiento de verdad, o SMV, están diseñados para manejar justamente este tipo de complicaciones.

REVISIÓN DE LA CREENCIA

SISTEMAS DE MANTENIMIENTO DE VERDAD

¹⁵ La revisión de la creencia se compara a menudo con la **actualización de la creencia**, que ocurre cuando se revisa una base de conocimiento para reflejar un cambio en el mundo, en lugar de trabajar con nueva información de un mundo fijo. La actualización de la creencia combina revisión de la creencia con razonamiento sobre el tiempo y el cambio. También está relacionada con el proceso de **filtrado** descrito en el Capítulo 15.

Una aproximación muy simple al mantenimiento de verdad es mantener la pista del orden en el que las sentencias se introducen en la base de conocimiento numerándolas desde P_1 hasta P_n . Cuando se realiza una llamada a *Retractar(KB, P_i)*, el sistema involuciona hasta el estado anterior en el que P_i fue añadido, eliminando de ese modo P_i y cualquier inferencia que fuera derivada de P_i . Las sentencias P_{i+1} hasta P_n pueden ser añadidas de nuevo. Esto es simple y garantiza que la base de conocimiento será consistente, pero el hecho de retractar P_i requiere retractar y afirmar $n - i$ sentencias, así como deshacer y rehacer todas las inferencias alcanzadas utilizando esas sentencias. En los sistemas donde se añaden mucho hechos (como bases de datos comerciales) esto es impracticable.

SMVJ**JUSTIFICACIÓN**

Una aproximación más eficiente son los sistemas de mantenimiento de verdad basados en justificación, o **SMVJ**. En un SMVJ, cada sentencia en la base de conocimiento se anota con una **justificación**, que consiste en el conjunto de sentencias a partir de las cuales fue inferida. Por ejemplo, si la base de conocimiento actualmente contiene $P \Rightarrow Q$, entonces DECIR(P) causará que se añada Q con la justificación $\{P, P \Rightarrow Q\}$. En general, una sentencia puede tener cualquier número de justificaciones. Las justificaciones se utilizan para que la operación de retractar sea eficiente. Dada la invocación RETRACTAR(P), el SMVJ detectará exactamente aquellas sentencias que tienen P como una justificación. Por lo tanto, si una sentencia Q tiene como única justificación $\{P, P \Rightarrow Q\}$, será borrada. Si además también tuviera la justificación $\{P, P \vee R \Rightarrow Q\}$, también sería borrada, pero si tuviera la justificación $\{R, P \vee R \Rightarrow Q\}$, entonces no sería borrada. De esta forma, el tiempo necesario para retractar P depende sólo del número de sentencias derivadas de P , en lugar de depender del número de sentencias añadidas desde que se insertó P en la base de conocimiento.

Los SMVJ asumen que las sentencias que han sido consideradas una vez, serán consideradas de nuevo, por lo tanto en lugar de borrar una sentencia de la base de conocimiento cuando pierde todas sus justificaciones, simplemente se marcarán las sentencias como si estuvieran *fuera* de la base de conocimiento. Si una afirmación posterior restituye una de las justificaciones, entonces la sentencia se marca de nuevo como *dentro*. De esta forma, los SMVJ mantienen toda la cadena de inferencia que utilizan y no vuelven a inferir sentencias cuando una justificación vuelve a ser válida.

Además de manejar el proceso de retractar información incorrecta, los SMV se pueden utilizar para acelerar el análisis de múltiples situaciones hipotéticas. Supóngase, por ejemplo, que el Comité Olímpico Romano está seleccionando lugares para natación, competiciones atléticas y eventos ecuestres para los Juegos Olímpicos que se celebrarán en Roma en el año 2048. Por ejemplo, sea la primera hipótesis *Lugar(Natación, Pitesti)*, *Lugar(CompeticionesAtléticas, Bucarest)* y *Lugar(EventosEcuestres, Arad)*. Se debe realizar una gran cantidad de razonamiento para obtener las consecuencias logísticas y por lo tanto la bondad de esta selección. Si en su lugar se desea considerar *Lugar(CompeticionesAtléticas, Sibiu)*, el SMV no tiene que empezar otra vez desde cero. En su lugar, simplemente se retractará *Lugar(CompeticionesAtléticas, Bucarest)* y se afirmará *Lugar(CompeticionesAtléticas, Sibiu)* y el SMV llevará a cabo las revisiones necesarias. La cadena de inferencia generada por la opción de Bucarest se puede reutilizar con Sibiu, dado que la conclusión es la misma.

SMVS

Un sistema de mantenimiento de verdad basado en suposiciones, o **SMVS** está diseñado para realizar este tipo de cambios de contexto entre mundos hipotéticos de un

modo muy eficiente. En un SMVJ, el mantenimiento de las justificaciones permite moverse rápidamente de un estado a otro, realizando pocas retracciones y afirmaciones, pero en un momento dado sólo se puede representar un estado. Un SMVS representa *todos* los estados que han sido considerados al mismo tiempo. Mientras que un SMVJ simplemente etiqueta cada sentencia como *dentro* o *fuera*, el SMVS lleva el control, por cada sentencia, de qué suposiciones harán que la sentencia sea verdadera. Es decir, cada sentencia tiene una etiqueta que está formada por un conjunto de suposiciones. La sentencia se cumple sólo en aquellos casos en los que todas las suposiciones de uno de los conjuntos de suposición se cumplen.

Los sistemas de mantenimiento de verdad también proporcionan un mecanismo para generar **explicaciones**. Técnicamente, una explicación de una sentencia *P* es un conjunto de sentencias *E* de modo que *E* implica *P*. Si se conoce que las sentencias en *E* son ciertas, entonces *E* simplemente proporciona una base suficiente para probar que *P* debe ser el caso. Pero las explicaciones también incluyen **suposiciones** (sentencias que no se sabe si son ciertas, pero en el caso de serlo serían suficientes para probar *P*). Por ejemplo, uno puede no tener suficiente información para probar que su coche no arrancará, pero una explicación razonable podría incluir la suposición de que la batería está agotada. Esto, combinado con el conocimiento de cómo funciona un coche, explica el hecho de que no arranque. En la mayoría de los casos, se prefiere una explicación *E* que es mínima, con el significado de que no existe un subconjunto propio de *E* que sea también una explicación. Un SMVS puede generar explicaciones para el problema de que «el coche no arranca» a través de la realización de suposiciones (como «combustible en el coche» o «batería agotada») en el orden que se desee, incluso si algunas suposiciones son contradictorias. A continuación se comprueba la etiqueta de la sentencia «el coche no arranca», para verificar el conjunto de suposiciones que justificarían la sentencia.

Los algoritmos empleados en la implantación de los sistemas de mantenimiento de verdad son un poco complicados y no se comentarán aquí. La complejidad computacional del problema de mantenimiento de verdad es por lo menos tan grande como la de la inferencia proposicional, es decir, de dificultad NP. Por ello, no es de esperar que el mantenimiento de verdad sea una panacea. Sin embargo, cuando se utiliza con cuidado un SMV puede proporcionar un incremento sustancial en la habilidad de un sistema lógico para manejar entornos complejos e hipótesis.

EXPLICACIONES

SUPOSICIONES

10.9 Resumen

Este capítulo ha sido con diferencia el más detallado del libro. Ahondando en los detalles de cómo uno representa una gran variedad de conocimiento, se espera haber dado al lector una idea de cómo se construyen las bases de conocimiento reales. Los puntos más importantes son los siguientes:

- La representación de conocimiento a gran escala necesita una ontología de propósito general para organizar y unir varios dominios de conocimiento específicos.
- Una ontología de propósito general necesita abarcar una amplia gama de conocimiento y debería ser capaz, en principio, de manejar cualquier dominio.

- Se ha presentado una **ontología superior** basada en categorías y en el cálculo de eventos. Se ha cubierto la estructura de objetos, espacio y tiempo, cambio, procesos, sustancias y creencias.
- Las acciones, los eventos y el tiempo se pueden representar utilizando el cálculo de situaciones o una representación más expresiva como el cálculo de eventos y el cálculo de flujos. Estas representaciones capacitan a un agente para construir planes mediante inferencia lógica.
- Los estados mentales de los agentes se pueden representar mediante cadenas que denoten creencias.
- Se ha presentado un análisis detallado del dominio de compra a través de Internet, ejercitando la ontología general y mostrando cómo el conocimiento del dominio se puede utilizar en un agente de compra.
- Los sistemas de representación de propósito específico, como las **redes semánticas** y la **lógica descriptiva**, han sido concebidos para ayudar en la organización jerárquica de categorías. La **herencia** es una forma importante de inferencia, permitiendo deducir las propiedades de los objetos a partir de su pertenencia a categorías.
- La **asunción del mundo cerrado**, implementada en programas lógicos, proporciona una forma simple de evitar el tener que especificar montones de información negativa. Es mejor su representación como **información por defecto** que puede ser sobrescrita por información adicional.
- La **lógica no monotónica**, como la **circunscripción** y la **lógica por defecto**, se utiliza para capturar el razonamiento por defecto en general. La **programación de conjunto de respuestas** acelera la inferencia no monotónica, así como WALKSAT acelera la inferencia proposicional.
- Los **sistemas de mantenimiento de verdad** manejan las actualizaciones y revisiones del conocimiento de forma eficiente.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

Existen afirmaciones plausibles (Briggs, 1985) en el sentido de que la investigación formal sobre la representación del conocimiento se inició en la India con los trabajos teóricos sobre la gramática del sánscrito shástrico, que datan del primer milenio a.C. En Occidente, el primer ejemplo lo constituyó el empleo de las definiciones de términos de la antigua matemática griega. De hecho, el desarrollo de terminología técnica en cualquier campo puede ser entendido como una forma de representación del conocimiento.

Los debates iniciales acerca de la representación en IA se enfocaban hacia la «representación del *problema*» en lugar de la «representación del *conocimiento*» (véase, por ejemplo, La discusión de Amarel (1968) acerca del problema de los caníbales y los misioneros). En los años 70, la IA hizo énfasis en el desarrollo de «sistemas expertos» (también llamados «sistemas basados en conocimiento») que podían, dado un conocimiento apropiado del dominio, alcanzar o superar el rendimiento de expertos humanos en tareas

específicas. Por ejemplo, el primer sistema experto, DENDRAL (Feigenbaum *et al.*, 1971; Lindsay *et al.*, 1980), interpretó la salida de un espectrómetro de masas (un instrumento usado para analizar la estructura de compuestos químicos orgánicos) de forma más precisa que un experto químico. Si bien el éxito obtenido por DENDRAL jugó un papel decisivo para que la comunidad de investigadores en IA tomara conciencia de la importancia que revestía la representación del conocimiento, los formalismos de representación utilizados en DENDRAL son muy específicos para el dominio de la química. Durante tiempo, los investigadores se interesaron en formalismos para la representación estandarizada del conocimiento y en ontologías que pudieran hacer más eficiente el proceso de crear nuevos sistemas expertos. Fue así como se aventuraron en un territorio que anteriormente había sido explorado por los filósofos de la ciencia y del lenguaje. La disciplina impuesta en IA por la necesidad de probar que las propias teorías «funcionan», ha dado como resultado un progreso más rápido y sólido que cuando estos problemas eran dominio exclusivo de la Filosofía (si bien en ocasiones ha dado lugar a la reinvencción de la rueda).

La creación de taxonomías entendibles o clasificaciones data de tiempos muy antiguos. Aristóteles (384-322 a.C.) hizo un fuerte énfasis en la clasificación y en esquemas de categorización. Su *Organon*, una colección de trabajos sobre lógica recapitulados por sus estudiantes después de su muerte, incluye un tratado titulado *Categorías*, en el cual intentó construir lo que nosotros hoy denominamos ontología superior. Él también introdujo la noción de **géneros** y **especies** para una clasificación de bajo nivel, aunque no con el significado moderno específico de biología. El sistema actual de clasificación biológica, incluyendo el uso de «nomenclatura binomial» (clasificación en géneros y especies en un sentido técnico), fue inventada por el biólogo sueco Carolus Linnaeus, o Carl von Linne (1707-1778). Los problemas asociados con las clases naturales y los límites de categorías inexactas han sido estudiados por Wittgenstein (1953), Quine (1953), Lakoff (1987) y Schwartz (1977) entre otros.

El interés en ontologías de gran escala es creciente. El proyecto CYC (Lenat, 1995; Lenat y Guha, 1990) ha creado una ontología superior de 6.000 conceptos con 60.000 hechos, y soporta una ontología global mucho mayor. El IEEE ha establecido el subcomité P1600.1, el grupo de trabajo de ontologías superiores estándar (*Standard Upper Ontology Working Group*), y la iniciativa de mente abierta (*Open Mind Initiative*) tiene afiliados más de 7.000 usuarios de Internet para introducir más de 40.000 hechos sobre conceptos relacionados con el sentido común. En la Web, estándares como RDF, XML y la Semántica de la Web (Berners-Lee *et al.*, 2001) están emergiendo, aunque aún no son ampliamente utilizados. Las conferencias sobre *Ontologías Formales en Sistemas de Información* (FOIS, *Formal Ontology in Information Systems*), contienen artículos muy interesantes sobre las ontologías generales y específicas de un dominio.

La taxonomía utilizada en este capítulo ha sido desarrollada por los autores y está basada en parte en su experiencia en el proyecto CYC, así como en el trabajo de Hwang y Schubert (1993) y Davis (1990). Una debate inspirador acerca del proyecto general de la representación basada en el conocimiento del sentido común, aparece en el trabajo de Hayes *The Naïve Physics Manifesto* (1978, 1985b).

La representación del tiempo, el cambio, las acciones y los eventos ha sido ampliamente estudiado en Filosofía y en Informática Teórica, así como en IA. La aproxi-

mación más antigua es la **lógica temporal**, que es una lógica especializada en la que cada modelo describe una trayectoria completa en el tiempo (usualmente lineal o ramificada), en vez de una estructura relacional estática. La lógica incluye operadores modales que se aplican a las fórmulas; $\Box p$ significa « p será siempre cierto» y $\Diamond p$ significa « p será cierto en algún momento en el futuro». El estudio de la lógica temporal fue iniciado por Aristóteles y las escuelas estoica y de Megara, en la antigua Grecia. En la actualidad, Findlay (1941) fue el primero en sugerir un cálculo formal para razonar sobre el tiempo, pero el trabajo de Arthur Prior (1967) es considerado el de mayor influencia. Los libros de texto incluyen las aproximaciones de Rescher y Urquhart (1971) y van Benthem (1983).

Los informáticos teóricos se han interesado desde hace tiempo en la formalización de las propiedades de los programas, vistas como secuencias de acciones computacionales. Burstall (1974) introdujo la idea de utilizar operadores modales para razonar acerca de los programas de computador. Poco después, Vaughan Pratt (1976) diseñó la **lógica dinámica**, en la que los operadores modales indican los efectos producidos por programas u otras acciones (véase también Harel, 1984). Por ejemplo, en la lógica dinámica, si α es el nombre de un programa, entonces $\langle\alpha|p\rangle$ significaría « p será válida en todos los estados del mundo que son resultado de la ejecución del programa α en el actual estado del mundo» y $\langle\langle\alpha|p\rangle$ significaría « p sería válida por lo menos en uno de los estados del mundo producido por la ejecución del programa α en el actual estado del mundo». Fischer y Ladner (1977) utilizaron la lógica dinámica en el análisis real de programas. Pnueli (1977) fue el primero en utilizar la lógica temporal clásica para razonar sobre los programas.

Mientras que la lógica temporal posiciona el tiempo directamente en la teoría modelo del lenguaje, las representaciones del tiempo en IA han tendido a incorporar axiomas sobre el tiempo y los eventos de forma explícita en la base de conocimiento, y no han dado al tiempo una posición especial en la lógica. Esta aproximación puede permitir una mayor claridad y flexibilidad en algunos casos. Además, el conocimiento temporal expresado utilizando lógica de primer orden se puede integrar más fácilmente con otro conocimiento que haya sido acumulado con esta notación.

El cálculo de situaciones de John McCarthy (1963), fue el primer tratamiento del tiempo y la acción en IA. El primer sistema de IA en hacer un uso sustancial del razonamiento de propósito general sobre acciones utilizando lógica de primer orden fue QA3 (Green, 1969b). Kowalski (1979b) desarrolló la idea de reificar proposiciones con el cálculo de situaciones.

El **problema del marco** fue reconocido por primera vez por McCarthy y Hayes (1969). Muchos investigadores consideraron el problema irresoluble utilizando lógica de primer orden, y esto estimuló un gran trabajo de investigación en lógica no monotónica. Los filósofos desde Dreyfus (1972) a Crockett (1994) han citado el problema del marco como un síntoma inevitable del fallo de toda la iniciativa de la IA. La solución parcial al problema de la representación del marco usando axiomas estado-sucesor se debe a Ray Reiter (1991). Una solución a la inferencia en el problema del marco fue hecha por el trabajo de Holldobler y Schneeberger (1990), que ha sido conocido como cálculo de flujos (Thielscher, 1999). El debate en este capítulo se basa parcialmente en el análisis realizado por Lin y Reiter (1997) y Thielscher (1999). Los libros de Shanahan

(1997) y Reiter (2001b) dan una visión completa y moderna del tratamiento del razonamiento sobre las acciones en el cálculo de situaciones.

La solución parcial del problema del marco ha reavivado el interés sobre la aproximación declarativa del razonamiento acerca de las acciones, que se ha visto eclipsado por los sistemas de planificación de propósito específico desde principios de los años 70 (véase el Capítulo 11). Bajo la etiqueta de **robótica cognitiva** se ha realizado un enorme progreso acerca de las representaciones lógicas de la acción y el tiempo. El lenguaje Golog usa toda la potencia expresiva de la lógica de la programación para describir acciones y planes (Levesque *et al.*, 1997a) y se ha extendido para manejar acciones concurrentes (Giacomo *et al.*, 2000), entornos estocásticos (Boutilier *et al.*, 2002) y percepción a través de los sentidos (Reiter, 2001a).

El cálculo de eventos fue introducido por Kowalski y Sergot (1986) para manejar tiempo continuo, y han surgido numerosas variaciones (Sadri y Kowalski, 1995). Shanahan (1999) presenta una sencilla e interesante revisión. James Allen introdujo intervalos de tiempo por la misma razón (Allen, 1983, 1984), argumentando que los intervalos eran mucho más naturales que las situaciones para el razonamiento sobre eventos extendidos y concurrentes. Peter Ladkin (1986a, 1986b) introdujo intervalos temporales «cóncavos» (intervalos con huecos, esencialmente uniones de intervalos temporales «convexos») y aplicó las técnicas del álgebra abstracta matemática a la representación del tiempo. Shoham (1987) describe la reificación de eventos y conjuntos a partir de un nuevo esquema de su invención para tal propósito. Existen aspectos significativos en común entre la ontología basada en eventos vista en este capítulo y los análisis de eventos debidos al filósofo Donald Davidson (1980). Las **historias** de Pat Hayes (1985a) como ontología de líquidos también tienen mucho en común.

El tema de la ontología de las sustancias también tiene una larga historia. Plato propuso que las sustancias fueran entidades abstractas totalmente distintas a los objetos físicos. Él diría *HechoDe(Mantequilla₃, Mantequilla)* en lugar de *Mantequilla₃ ∈ Mantequilla*. Esto lleva a una jerarquía de sustancias en la cual por ejemplo, *Mantequilla-SinSal* es una sustancia más específica que *Mantequilla*. La posición adoptada en este capítulo, en la cual las sustancias son categorías de objetos, fue defendida por Richard Montague (1973). También ha sido adoptada en el proyecto CYC. Copeland (1993) creó un serio, pero no insuperable, ataque. La aproximación mencionada en el capítulo, en el que la mantequilla es un objeto que consiste en todos los objetos mantecosos del universo, fue propuesto originalmente por el logístico polaco Lesniewski (1916). Su **mereología** (el nombre se deriva de la palabra griega que significa «parte») usa la relación parte-conjunto como un sustituto para la teoría de conjuntos matemáticos, con la finalidad de eliminar entidades abstractas como los conjuntos. Una exposición de estas ideas fácil de leer la dan Leonard y Goodman (1940), y el trabajo *The Structure of Appearance* (Goodman, 1977) aplica las ideas a varios problemas de representación del conocimiento. Aunque algunos aspectos de la aproximación mereológica son confusos (por ejemplo, la necesidad de un mecanismo de herencia separado basado en relaciones parte-conjunto), la aproximación ganó el apoyo de Quine (1960). Harry Bunt (1985) ha proporcionado un análisis exhaustivo de su uso en la representación del conocimiento.

Los objetos mentales y los estados han sido objeto de intenso estudio en Filosofía e IA. La **lógica modal** es el método clásico para el razonamiento acerca del conocimien-

to en Filosofía. La lógica modal añade operadores modales a la lógica de primer orden, tales como *B* (cree) y *K* (conoce), que reciben *sentencias* en lugar de términos como argumentos. La teoría de la demostración en la lógica modal limita la sustitución al ámbito de los contextos modales, lo que le permite lograr opacidad referencial. La lógica modal del conocimiento fue inventada por Jaakko Hintikka (1962). Saul Kripke (1963) definió la semántica de la lógica modal del conocimiento en función de **mundos posibles**. En términos generales, se dice que un mundo es posible para un agente siempre y cuando dicho mundo sea congruente con todo lo que el agente sabe. Con base en lo anterior, se pueden deducir reglas de inferencia en las que participe el operador *K*. Robert C. Moore vincula la lógica modal del conocimiento con un estilo para razonar sobre el conocimiento que hace referencia directa a los mundos posibles de la lógica de primer orden (Moore, 1980, 1985). Si bien la lógica modal puede parecer a veces un campo arcano e intimidante, son muchas sus aplicaciones en el razonamiento acerca de la información en sistemas distribuidos. El libro *Reasoning about Knowledge* de Fagin *et al.* (1995) proporciona una esmerada introducción a la aproximación modal. La conferencia bianual *Theoretical Aspects of Reasoning About Knowledge* (TARK) abarca aplicaciones de la teoría del conocimiento en IA, ciencias económicas y sistemas distribuidos.

La teoría sintáctica de los objetos mentales fue por primera vez estudiada en profundidad por Kaplan y Montague (1960), quienes mostraron que se puede llegar a paradoxas si no se maneja con cuidado. Debido a que tiene un modelo natural en términos de creencias como configuraciones físicas de un computador o un cerebro, se ha hecho popular en IA en los últimos años. Konolige (1982) y Haas (1986) la usaron para describir motores de inferencia de potencia limitada, mientras que Morgenstern (1987) mostró cómo podría ser usada para describir precondiciones del conocimiento en planificación. Los métodos para acciones basadas en la observación de planes descritos en el Capítulo 12 están basados en la teoría sintáctica. Ernie Davis (1990) establece una excelente comparación entre las teorías del conocimiento sintáctica y modal.

El filósofo griego Porphyry (234-305 a.C.) comenta sobre las *Categorías* de Aristóteles que establecen lo que podría calificarse como la primera red semántica. Charles S. Peirce (1909) desarrolló grafos existenciales como el primer formalismo de redes semánticas usando lógica moderna. Ross Quillian (1961), conducido por un interés acerca de la memoria humana y el procesamiento del lenguaje, inició el trabajo con las redes semánticas dentro del campo de la IA. Un artículo influyente de Marvin Minsky (1975) presentó una versión de las redes semánticas denominadas **marcos**. Un marco era una representación de un objeto o categoría, con atributos y relaciones con otros objetos o categorías. Aunque el artículo sirvió para iniciar el interés en el campo de la representación del conocimiento en sí, fue criticado por ser un conjunto de ideas recicladas desarrolladas en el campo de la programación orientada a objetos, como la herencia y el uso de valores por defecto (Dahl *et al.*, 1970; Birtwistle *et al.*, 1973). No está muy claro hasta qué punto los artículos posteriores sobre programación orientada a objetos fueron influenciados por trabajos anteriores de IA en el campo de las redes semánticas.

La cuestión semántica surgió muy perspicazmente en relación con las redes semánticas de Quillian (y todos aquellos que siguieron su aproximación), con su omnipresente y muy vago «enlace ES-UN», así como otros formalismos de representación del conocimiento anteriores como el de MERLIN (Moore y Newell, 1973), con sus misterio-

sas operaciones «flat» y «cover». El famoso artículo *What's In a Link?* de Wood (1975) atrajo la atención de los investigadores de IA hacia la necesidad de disponer de una semántica precisa en los formalismos de representación del conocimiento. Brachman (1979) entró en detalles sobre este punto y proporcionó soluciones. El trabajo de Patrick Hayes (1979) *The Logic of Frames* fue aún más allá, reivindicando que «la mayoría de los 'marcos' no son más que una nueva sintaxis para partes de la lógica de primer orden». Drew McDermott (1978b) en su trabajo *Tarskian Semantics, or, No Notation without Denotation!* argumentó que la aproximación basada en la teoría de modelos para la semántica usada en la lógica de primer orden debería ser aplicada a todos los formalismos para la representación del conocimiento. Esto permanece como una idea contradictoria. En particular, McDermott ha cambiado su opinión en *A Critique of Pure Reason* (McDermott, 1987). NETL (Fahlman, 1979) fue un sistema sofisticado de red semántica cuyos enlaces ES-UN (llamados «copias virtuales», o enlaces VC), se basaron más en la noción de «herencia» característica de los sistemas de marco o de los lenguajes de programación orientada a objetos, que en el subconjunto de relaciones, y fueron mucho mejor definidos que los enlaces de Quillian de la era anterior a Wood. NETL es particularmente interesante porque fue diseñado para ser implementado en *hardware* paralelo para superponerse a la dificultad de recuperar información de redes semánticas grandes. David Touretzky (1986) trata la herencia en el análisis matemático riguroso. Selman y Levesque (1993) tratan la complejidad de la herencia con excepciones, mostrando que en la mayoría de las implementaciones en NP-complejo.

El desarrollo de la lógica descriptiva es la etapa más reciente de una larga línea de investigación que ha tenido como objetivo el encontrar subconjuntos de la lógica de primer orden, para los cuales la inferencia sea computacionalmente tratable. Hector Levesque y Ron Brachman (1987) mostraron que ciertas construcciones lógicas (en particular, ciertos usos de la disyunción y la negación) fueron responsables en primer grado de que la herencia en lógica fuera no tratable computacionalmente. Basados en el sistema KL-ONE (Schmolze y Lipkis, 1983), se han desarrollado un gran número de sistemas cuyos diseños incorporan los resultados del análisis teórico de la complejidad, destacando KRYPTON (Brachman *et al.*, 1983) y CLASSIC (Borgida *et al.*, 1989). El resultado ha sido un incremento notable en la velocidad de la inferencia y un mejor entendimiento de la interacción entre complejidad y expresividad en los sistemas de razonamiento. Calvanese *et al.* (1999) realiza un resumen del estado del arte. En contra de esta tendencia Doyle y Patil (1991) han argumentado que restringir la expresividad de un lenguaje hace imposible resolver ciertos tipos de problemas o fomenta al usuario a rodear las restricciones del lenguaje utilizando significados no lógicos.

Los tres formalismos principales para tratar con la inferencia no monotónica (circunscripción (McCarthy, 1980), lógica por defecto (Reiter, 1980) y lógica no monotónica modal (McDermott y Doyle, 1980)), fueron introducidos en un volumen especial de la revista de IA. La programación de conjunto de respuestas puede ser vista como una extensión de la negación como fallo, o como un refinamiento de la circunscripción. La teoría de base de la semántica estable de modelos fue introducida por Gelfond y Lifschitz (1988), y los sistemas de referencia para la programación de conjunto de respuestas son DLV (Eiter *et al.*, 1998) y SMODELS (Niemelä *et al.*, 2000). El ejemplo de la unidad de disco procede del manual de usuario de SMODELS (Syrjänen, 2000). Lifschitz (2001)

aborda el tema de la programación de conjunto de respuestas aplicado a planificación. Brewka *et al.* (1997) hace una buena revisión de varias aproximaciones a la lógica no monotónica. Clark (1978) trata la aproximación de negación como fallo en la programación lógica y la completitud de Clark. Van Emden y Kowalski (1976) muestran que cualquier programa Prolog sin la negación tiene un modelo mínimo único. Últimamente se ha renovado el interés en la aplicación de lógica no monotónica a los sistemas de representación de conocimiento a gran escala. El sistema BENINQ para el manejo de preguntas sobre los beneficios de seguros seguramente fue la primera aplicación comercial con éxito de un sistema de herencia no monotónica (Morgenstern, 1998). Lifschitz (2001) trata sobre la aplicación de la programación de conjunto de respuestas aplicado a planificación. Un amplio conjunto de sistemas de razonamiento no monotónicos basados en programación lógica se encuentran documentados en las actas de la conferencia en *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

El estudio de sistemas de mantenimiento de verdad comenzó con los sistemas TMS (Doyle, 1979) y RUP (McAllester, 1980), los cuales eran esencialmente un SMVJ. La aproximación de los SMVS se describe en una serie de artículos de Johan de Kleer (1986a, 1986b, 1986c). *Building Problem Solvers* (Forbus y de Kleer, 1993) explica en profundidad cómo se pueden utilizar los SMV en aplicaciones de IA. Nayak y Williams (1997) muestran cómo un SMV eficiente hace posible planear las operaciones de una nave espacial de la NASA en tiempo real.

Por razones obvias este capítulo no cubre *todas* las áreas de la representación del conocimiento en profundidad. Los temas principales omitidos son los siguientes:

FÍSICA CUALITATIVA

- **Física cualitativa:** subcampo de la representación del conocimiento que se ocupa específicamente de la construcción de una teoría lógica, no numérica de los objetos y procesos físicos. El término fue acuñado por Johan de Kleer (1975), si bien podría considerarse a Fahlman (1974) como el precursor con su planificador BUILD. Éste era un planificador complejo para construir complicadas torres de bloques. Durante el proceso de diseño, Fahlman se dio cuenta de que la mayor parte del trabajo (80 por ciento) se invertía en modelar la física del mundo de los bloques para determinar la estabilidad de diversos subconjuntos de bloques, en vez de hacerlo dentro de la planificación en sí. Esbozó un hipotético procedimiento semejante a la física intuitiva (*naive*) para explicar el por qué los niños son capaces de resolver problemas semejantes a los de BUILD sin recurrir a la veloz aritmética de punto flotante empleada en el modelado físico de BUILD. Hayes (1985a) emplea «historias», segmentos espacio-temporales de cuatro dimensiones semejantes a los eventos de Davidson, para construir una física intuitiva de los líquidos bastante compleja. Hayes fue el primero en demostrar que una bañera con un tapón terminará desbordándose si el grifo se mantiene abierto y que una persona que cae en un lago terminará empapada. De Kleer y Brown (1985) y Ken Forbus (1985) intentaron la construcción de algo así como una teoría de propósito general para el mundo físico, basada en las abstracciones cualitativas de ecuaciones físicas. En los últimos años, la física cualitativa se ha desarrollado al grado de que permite el análisis de una impresionante diversidad de sistemas físicos complejos (Sacks y Joskowicz, 1993; Yip, 1991). Las técnicas cualitativas se han empleado también para construir novedosos diseños de relojes, lim-

piaparabrisas y robots de seis piernas (Subramanian, 1993; Subramanian y Wang, 1994). Una buena introducción a este campo es la colección *Readings in Qualitative Reasoning about Physical Systems* (Weld y de Kleer, 1990).

RAZONAMIENTO ESPACIAL

- **Razonamiento espacial:** el razonamiento necesario para navegar en el mundo de *wumpus* y el del mercado es trivial comparado con la rica estructura espacial del mundo real. El esfuerzo más completo por capturar el razonamiento con sentido común acerca del espacio está plasmado en el trabajo de Ernest Davis (1986; 1990). El cálculo de conexión de regiones de Cohn *et al.*, (1997) utiliza una forma de razonamiento espacial cualitativo y ha llevado a nuevas clases de sistemas de información geográfica. Como en el caso de la física cualitativa, muestra que los agentes pueden avanzar bastante, por decirlo así, sin recurrir a la ayuda de la representación métrica. Cuando tal representación es necesaria, se pueden emplear las técnicas diseñadas por la robótica (Capítulo 25).

RAZONAMIENTO PSICOLÓGICO

- **Razonamiento psicológico:** consiste en el desarrollo de una *psicología* de trabajo para que los agentes la utilicen al razonar sobre sí mismos y otros agentes. A menudo, está basada en lo que se conoce como psicología popular; aquellas teorías que en general utilizan los humanos al razonar sobre sí mismos y sobre otros seres humanos. Cuando los investigadores de IA proporcionan a sus agentes artificiales teorías psicológicas para razonar sobre otros agentes, tales teorías por lo general se basan en la descripción de los investigadores del propio diseño de los agentes lógicos. El razonamiento psicológico es realmente más útil en el contexto de la interpretación del lenguaje natural, en donde la capacidad para anticipar las intenciones del hablante es de suma importancia.

Las actas de las conferencias internacionales sobre los *Principios de la Representación del Conocimiento y del Razonamiento* constituyen las fuentes más actualizadas de los trabajos que se realizan en esta área. *Readings in Knowledge Representation* (Brachman y Levesque, 1985) y *Formal Theories of the Commonsense World* (Hobbs y Moore, 1985) son excelentes antologías de la representación del conocimiento. La primera se enfoca más en artículos de importancia histórica, en lenguajes de representación y formalismos, mientras que la segunda lo hace en el conocimiento acumulado en este campo. Davis (1990), Stefik (1995) y Sowa (1999) proporcionan introducciones a los libros de texto acerca de la representación del conocimiento.



EJERCICIOS

10.1 Escriba sentencias para definir los efectos de la acción *Disparar* en el mundo de *wumpus*. Describa sus efectos en *wumpus* y recuerde que el disparo usa la flecha del agente.

10.2 En el cálculo de situaciones, escriba un axioma para asociar el tiempo 0 con la situación S_0 y otro axioma para asociar el tiempo t con cualquier situación derivada de S_0 a través de una secuencia t de acciones.

10.3 En este ejercicio, se considerará el problema de planear una ruta para un robot para ir desde una ciudad a otra. La acción básica que puede tomar el robot es *Ir(x, y)*, que lo lleva desde la ciudad *x* a la ciudad *y* si existe una ruta directa entre esas ciudades. *RutaDirecta(x, y)* es cierto si y sólo si, existe una ruta directa entre *x* e *y*. Se puede asumir que todos estos hechos están ya en la base de conocimiento (véase el mapa en el Apartado 3.1). El robot comienza en Arad y debe llegar a Bucarest.

- a) Escriba una descripción lógica adecuada para la situación inicial del robot.
- b) Escriba una petición lógica adecuada cuyas soluciones proporcionarán rutas posibles hacia el objetivo.
- c) Escriba una sentencia describiendo al acción *Ir*.
- d) Ahora supóngase que siguiendo la ruta directa entre dos ciudades se consume una cantidad de combustible igual a la distancia entre las ciudades. El robot comienza con el depósito lleno de combustible. Aumente la representación para incluir estas consideraciones. La descripción de la acción debe ser tal, que la petición formulada anteriormente proporcione planes viables.
- e) Describa la situación inicial, y escriba una nueva regla o conjunto de reglas para describir la acción *Ir*.
- f) Ahora supóngase que algunos de los vértices tienen también gasolineras, donde el robot puede llenar su depósito. Extienda la representación y escriba todas las reglas necesarias para describir las gasolineras, incluyendo la acción de *Repostar*.

10.4 Utilizando los axiomas de la Sección 10.3, muestre los pasos de razonamiento lógico que permiten concluir que la acción *Ambos(Impulsar(Izquierda), Impulsar(Derecha))* no derramarán la sopa.

10.5 Represente las siguientes siete sentencias utilizando y extendiendo las representaciones desarrolladas en el capítulo:

- a) El agua es líquida entre 0 y 100 grados.
- b) El agua hiere a 100 grados.
- c) El agua de la botella de John está congelada.
- d) Perrier es un tipo de agua.
- e) John tiene agua de tipo Perrier en su botella.
- f) Todos los líquidos tienen un punto de congelación.
- g) Un litro de agua pesa más que un litro de alcohol.

Ahora repita el ejercicio usando una representación que utilice mereología, en el cual, por ejemplo, *Agua* es un objeto que contiene como partes toda el agua del mundo.

10.6 Escriba definiciones para lo siguiente:

- a) *DescomposiciónDeParteExhaustiva*.
- b) *ParticiónDeParte*.
- c) *PartesDisjuntas*.

Esto debería ser análogo a la definición de *DecomposiciónExhaustiva*, *Partición* y *Disyunción*. ¿Es el caso de *ParticiónDeParte(s, GrupoDe(s))*? si es así demuéstrelo, si no, proporcione un contraejemplo y defina las condiciones suficientes bajo las cuales sí es cierto.

10.7 Escriba un conjunto de sentencias que permitan calcular el precio de un tomate individual (u otro objeto), dado el precio por kilo. Extienda la teoría para permitir el cálculo del precio de una bolsa de tomates.

10.8 Un esquema alternativo para representar medidas consiste en aplicar la función de medida a un objeto de longitud abstracta. En este esquema, ¿se podría escribir $Pulgadas(Longitud(L_1)) = 1,5$? ¿Cómo se compara este esquema con el visto en el capítulo? Algunos aspectos a tener en cuenta son la necesidad de incluir axiomas de conversión, nombres para cantidades abstractas (como «50 dólares») y comparaciones de medidas abstractas en diferentes unidades (50 pulgadas son más que 50 centímetros).

10.9 Construya una aplicación para intercambio de monedas que permita fluctuaciones diarias del cambio de moneda.

10.10 Este ejercicio contempla las relaciones entre categorías de eventos y el intervalo de tiempo en el cual ocurren:

- a) Defina el predicado $T(c, i)$ en términos de $SubEvento$ y \in .
- b) Explique de forma precisa por qué no se necesitan dos notaciones diferentes para describir categorías de eventos disjuntas.
- c) Dé una definición formal para $T(UnoDe(p, q), i)$ y $T(O(p, q), i)$.
- d) Explique si tiene sentido tener dos formas de negación de eventos, análogas a las dos formas de disyunción. Denomine *No* y *Nunca* y dé para ellas una definición formal.

10.11 Defina el predicado *Constante*, donde $Constante(Localización(x))$ significa que la localización del objeto x no cambia en el tiempo.

10.12 Defina los predicados *Antes*, *Después*, *Durante* y *Superpuesto*, utilizando el predicado *Coincidir* y las funciones *Inicio* y *Fin*, pero no la función *Tiempo* o el predicado $<$.

10.13 En la Sección 10.5 se utilizaron los predicados *Enlace* y *TextoDeEnlace* para describir conexiones entre páginas web. Utilice los predicados *EnEtiqueta* y *ObtenerPágina*, entre otros, para escribir definiciones para *Enlace* y *TextoDeEnlace*.

10.14 Una parte del proceso de compra que no fue cubierto en este capítulo es la comprobación de compatibilidad entre artículos. Por ejemplo, si un cliente pide un computador, ¿se conecta adecuadamente con los periféricos deseados? Si se solicita una cámara digital, ¿tiene la tarjeta de memoria correcta y las baterías? Escriba una base de conocimiento que decida cuándo un conjunto de artículos es compatible y puede ser utilizado para sugerir cambios o artículos adicionales si los originales no son compatibles. Asegúrese de que la base de conocimiento trabaja con al menos una gama de productos y es fácilmente extensible a otras.

10.15 Añada reglas para extender la definición del predicado $NOMBRE(s, c)$ de tal forma que una cadena como «computador portátil» se corresponda con los nombres de categorías adecuados de diferentes almacenes. Trate de hacer la definición general. Compruebe si funciona buscando en diez almacenes *online* y en los nombres de categorías que tiene cada uno. Por ejemplo, para la categoría de portátiles, se encuentran los nom-

bres «Notebooks», «Portátiles», «Computadores notebook», «Portátiles y notebooks» y «Notebooks PC». Algunos de ellos pueden ser soportados por ocurrencias *Nombre* explícitas, mientras que otros podrían ser implementados por reglas para manejar plurales, conjunciones, etc.

10.16 Una solución completa al problema de las correspondencias inexactas en las descripciones de un cliente respecto a su compra es muy difícil, y requiere un completo despliegue de procesamiento de lenguaje natural y técnicas de recuperación de información (véanse los Capítulos 22 y 23). Un pequeño paso es permitir al usuario especificar unos valores mínimos y máximos para varios atributos. Se insistirá en que el usuario utilice la siguiente gramática a la hora de describir sus productos:

$$\begin{aligned} \text{Descripción} &\rightarrow \text{Categoría} \mid \text{Conector Modificador}^* \\ \text{Conector} &\rightarrow \ll\text{con}\rr \mid \ll\text{y}\rr \mid \ll,\rr \\ \text{Modificador} &\rightarrow \text{Atributo} \mid \text{Atributo Op Valor} \\ \text{Op} &\rightarrow \ll=\rr \mid \ll>\rr \mid \ll<\rr \end{aligned}$$

Aquí, *Categoría* da nombre a una categoría de productos, *Atributo* es cualquier característica como «CPU» o «precio» y *Valor* es el valor objetivo para el atributo. Por lo tanto, la petición «computador con al menos una CPU de 2,5 GHz por menos de 1.000 dólares» se ha de expresar como «computador con CPU $> 2,5$ GHz y precio < 1.000 dólares». Implemente un agente de compra que acepte descripciones en este lenguaje.

10.17 En la descripción de compra por Internet se omitió el paso importante de realmente *comprar* el producto. Proporcione una descripción lógica formal para comprar, utilizando cálculo de eventos. Es decir, defina la secuencia de eventos que ocurren cuando un comprador envía su pedido con el número de su tarjeta y finalmente se emite un recibo y recibe el producto.

10.18 Describa el evento de intercambiar algo por otra cosa. Describa la compra como una clase de intercambio donde uno de los objetos intercambiados es una cantidad de dinero.

10.19 Los dos ejercicios anteriores asumen una noción bastante primitiva de la propiedad. Por ejemplo, el comprador comienza *poseyendo* los billetes. Este panorama comienza a desmoronarse cuando, por ejemplo, el dinero de uno está en el banco, porque no hay ninguna colección específica de billetes que uno posee. El panorama se complica cuando se utilizan préstamos, *leasing*, alquileres y fianzas. Investigue los aspectos de sentido común y conceptos legales sobre la pertenencia, y proponga un esquema mediante el cual puedan ser representados formalmente.

10.20 Se trata de crear un sistema para aconsejar a los alumnos de informática sobre qué cursos seleccionar durante un período de tiempo prolongado para satisfacer los requerimientos del programa (use cualquier requerimiento que sea adecuado para su institución). Primero, decida un vocabulario para representar toda la información y después represéntela. A continuación utilice una petición adecuada para el sistema, que devolverá un programa legal de estudios como solución. Se debería permitir, para adaptar el sistema a usuarios individuales, que el sistema preguntara por los cursos o equivalencias que los estudiantes ya han cursado, y no generar programas que repitan dichos cursos.



Sugiera formas en las que el sistema podría ser mejorado (por ejemplo teniendo en cuenta el conocimiento sobre las preferencias de los estudiantes, la carga de trabajo, instructores buenos y malos etc.). Para cada clase de conocimiento, explique cómo podría ser expresado lógicamente. ¿Podría el sistema incorporar fácilmente esta información para encontrar el *mejor* programa de estudios para un estudiante?

10.21 La Figura 10.1 muestra los niveles superiores de una jerarquía para cualquier cosa. Extiéndase para incluir tantas categorías como sean posibles. Una buena forma de hacer esto es cubrir todas las cosas de la vida a diario. Esto incluye objetos y eventos. Comience por levantarse y seguir de una forma ordenada anotando todo lo que uno ve, toca, hace y piensa. Por ejemplo, un ejemplo de prueba aleatoria produce música, noticias, leche, andar, conducir, repostar, soda, alfombra, hablar, profesor Fateman, pollo al curry, lengua, 7 dólares, sol, la prensa del día, etc.

Se debería generar una gráfico de jerarquía simple (en un trozo de papel grande) y un listado de objetos y categorías con las relaciones que satisfacen los miembros de cada categoría. Todo objeto debería estar en una categoría, y cada categoría debería estar en la jerarquía.

10.22 (Adaptado de un ejemplo de Doug Lenat.) La misión es capturar, de una forma lógica, conocimiento suficiente para responder una serie de preguntas sobre la siguiente sentencia simple:

Ayer John fue al supermercado Safeway del norte de Berkeley y compró dos libras de tomates y una libra de carne picada.

Comience por tratar de representar el contenido de la sentencia en un conjunto de aserciones. Se deberían escribir sentencias con una estructura lógica sencilla (por ejemplo, sentencias en las que los objetos tienen ciertas propiedades, en las que los objetos se relacionan de cierta forma, en las que los objetos que satisfacen una propiedad satisfacen otra). Lo siguiente puede ayudar para empezar:

- ¿Qué clases, objetos y relaciones son necesarios? ¿Cuáles son sus padres, hermanos y demás? (Se necesitarán eventos y ordenamiento temporal, entre otras cosas.)
- ¿En qué lugar tendrían cabida en una jerarquía más general?
- ¿Cuáles son las restricciones y las interrelaciones entre ellos?
- ¿Qué nivel de detalle se debe tener con cada uno de los conceptos?

La base de conocimiento que se construya deberá ser capaz de responder una lista de preguntas que se proporcionarán en breve. Algunas de las preguntas tratan sobre el material dado explícitamente en el texto, pero la mayoría requieren tener otro conocimiento de fondo (una lectura entre líneas). Se deberá tratar con las clases de cosas que hay en un supermercado, lo que rodea el proceso de comprar las cosas seleccionadas, para qué serán usadas, etc. Trate de realizar la representación lo más general posible. Por dar un ejemplo trivial: no diga «La gente compra comida en Safeway», porque esto no le ayudará con la gente que compra en otro supermercado. No diga «Joe hace spaghetti con tomate y carne picada», porque no le ayudará con absolutamente nada más. Además, no le dé la vuelta a las preguntas convirtiéndolas en respuestas. Por ejemplo, la pregunta (c) dice: «¿Compró John carne?» (no «¿Compró John una libra de carne picada?»).

Esboce el proceso de razonamiento que respondería a las preguntas. Para realizarlo, necesitará sin lugar a dudas crear conceptos adicionales, realizar nuevas aserciones, etc. Si es posible, utilice un sistema de razonamiento lógico para demostrar la suficiencia de su base de conocimiento. La mayoría de las cosas que escriba no serán del todo correctas en la realidad, pero no se preocupe demasiado. La idea es extraer el sentido común que le permita responder a todas las cuestiones. Una respuesta totalmente correcta a todas las preguntas es *extremadamente* compleja, probablemente más allá del estado de arte en las investigaciones actuales sobre representación del conocimiento. Pero usted debería ser capaz de proporcionar un conjunto consistente de axiomas para las preguntas en concreto aquí expresadas.

- a) ¿John es un niño o un adulto? [Adulto]
- b) ¿Tiene John ahora por lo menos dos tomates? [Sí]
- c) ¿Compró John carne? [Sí]
- d) ¿Si Mary estaba comprando tomates al mismo tiempo que John, le vio él a ella? [Sí]
- e) ¿Se hacen los tomates en el supermercado? [No]
- f) ¿Qué va a hacer John con los tomates? [Comérselos]
- g) ¿Se vende desodorante en Safeway? [Sí]
- h) ¿Trajo John dinero al supermercado? [Sí]
- i) ¿Tiene John menos dinero después de ir al supermercado? [Sí]

10.23 Añada ó realice los cambios necesarios a la base de conocimiento del ejercicio anterior para que las preguntas que siguen a continuación se puedan responder. Muestre que se pueden realmente responder utilizando la base de conocimiento, e incluya en el informe una referencia de los cambios, explicando por qué fueron necesarios, si fueron grandes o pequeños, etc.

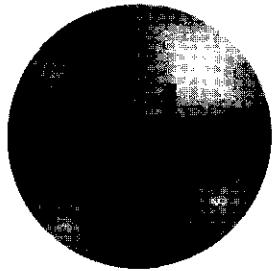
- a) ¿Hay otra gente en Safeway mientras John está allí? [Sí, ¡el personal!]
- b) ¿Es John vegetariano? [No]
- c) ¿De quién es el desodorante de Safeway? [De la compañía Safeway]
- d) ¿Obtuvo John una libra de carne picada? [Sí]
- e) ¿Tenía la gasolinera cerca de su puerta combustible? [Sí]
- f) ¿Cupieron los tomates en el maletero del coche de John? [Sí]

10.24 Recuerde que la información sobre la herencia en las redes semánticas puede ser capturada lógicamente por sentencias de implicación adecuadas. En este ejercicio, se considera la eficiencia de usar estas sentencias para la herencia.

- a) Considere la información contenida en un catálogo de coches usados como el Libro Azul de Kelly (por ejemplo, que las furgonetas Dodge de 1973 valen 575 dólares). Suponga que toda esta información (para 11.000 modelos) está codificada en reglas lógicas como se sugiere en este capítulo. Escriba tres reglas así, incluyendo las de la furgoneta Dodge de 1973. ¿Cómo usaría las reglas para encontrar el valor de un vehículo concreto (por ejemplo, JB), el cual es una furgoneta Dodge de 1973?
- b) Compare la eficiencia en tiempo del método de encadenamiento hacia atrás para resolver este problema con el método de herencia usado en las redes semánticas.

- c) Explique cómo el encadenamiento hacia delante permite a un sistema basado en lógica resolver el mismo problema eficientemente, suponiendo que la base de conocimiento sólo contiene las 11.000 reglas sobre precios.
- d) Describa una situación en la que ni el encadenamiento hacia delante ni el que va hacia atrás permitirá manejar eficientemente una consulta de precio para un vehículo particular.
- e) ¿Puede sugerir una solución que permita resolver eficientemente este tipo de consulta en todos los casos en sistemas lógicos? (*Pista:* Recuerde que dos vehículos de la misma categoría tienen el mismo precio.)

10.25 Uno podría suponer que la distinción sintáctica entre los enlaces no encuadrados y los encuadrados de forma simple en las redes semánticas es innecesaria, porque los enlaces encuadrados de forma simple están unidos siempre a categorías. Un algoritmo de herencia podría asumir simplemente que un enlace no encuadrado debe aplicarse a todos los miembros de esa categoría. Muestre que este argumento es falaz, dando ejemplos de los errores que podrían surgir.



Planificación

Donde veremos cómo un agente puede extraer ventaja del conocimiento de la estructura de un problema para construir complejos planes de acción.

Llamaremos **planificación** al proceso de búsqueda y articulación de una secuencia de acciones que permitan alcanzar un objetivo. Hemos visto dos ejemplos de agentes planificadores hasta el momento: el agente solucionador de problemas basado en búsquedas, Capítulo 3, y el agente planificador lógico del Capítulo 10. Este capítulo se ocupará principalmente de *ampliar* el estudio a problemas de planificación complejos que no pueden abordarse mediante los enfoques propuestos hasta ahora.

La Sección 11.1 presenta un lenguaje adecuado para la formulación de problemas de planificación, incluyendo acciones y estados. Este lenguaje está estrechamente relacionado con el que se mostró para la representación proposicional y de primer-orden de acciones, estudiadas en los Capítulos 7 y 10. La Sección 11.2 muestra cómo los algoritmos de búsqueda hacia-adelante y hacia-atrás aprovechan esta forma de representación, principalmente a través de heurísticas adecuadas que pueden derivarse automáticamente de la estructura de la representación de los problemas (análogamente al modo en que vimos en el Capítulo 5 cómo se construyeron heurísticas adecuadas en problemas a los que se imponían restricciones). La Secciones 11.3, 11.4 y 11.5 describen algoritmos de planificación que van más allá que las búsquedas hacia-adelante y hacia-atrás, aprovechándose de la información proporcionada por la estructura de los problemas. En particular, exploraremos enfoques que no están restringidos exclusivamente a la consideración de secuencias de acciones ordenadas.

En este capítulo, tendremos en cuenta entornos que son completamente observables, deterministas, finitos, estáticos (los cambios suceden sólo cuando los agentes actúan) y discretos (en tiempo, acciones, objetos y efectos). Estos contextos se deno-

minan entornos de **planificación clásica**. En contraste, la planificación no-clásica se ocupará de contextos parcialmente observables o estocásticos donde se aplicarán diferentes propuestas y diseños de agentes y algoritmos, tal como se expondrá en los Capítulos 12 y 17.

11.1 El problema de planificación

Consideremos qué sucedería si un agente solucionador de problemas, que utilizable algoritmos de búsqueda clásicos para llevar a cabo su tarea (búsqueda en profundidad, A*, etc.), se enfrentase a problemas en entornos reales. Este planteamiento nos ayudará a diseñar mejores agentes de planificación.

La primera dificultad y más obvia es la posibilidad de que el agente pueda ser desbordado por acciones irrelevantes al problema. Consideremos la tarea de comprar un ejemplar del libro «*Inteligencia Artificial: un enfoque moderno*» en una librería *online*. Supongamos que tenemos una acción de compra para cada número ISBN de 10 dígitos y para un total de 10 billones de acciones. El algoritmo de búsqueda tendría que examinar los resultados de los 10 billones de acciones para encontrar una que satisficiera el objetivo de adquirir la copia de un libro con ISBN 0137903952. En contraposición, un agente de planificación razonable debería ser capaz de trabajar con expresiones de objetivos explícitas tales como *Tener(ISBN 0137903952)* y generar la acción *Comprar(ISBN 0137903952)* directamente. Para hacer esto, el agente simplemente necesita un conocimiento general del tipo «*Comprar(x)* se reduce a *Tener(x)*». Dado este conocimiento y el objetivo propuesto, el planificador puede decidir en un único paso unificado que *Comprar(ISBN 0137903952)* es la acción correcta.

La siguiente dificultad es encontrar una **función heurística** adecuada. Supongamos que el objetivo del agente es comprar cuatro libros diferentes en una librería *online*. Tendremos 10^{40} planes formados por cuatro etapas, por lo que una búsqueda sin recurrir a la ayuda de una heurística no puede ser ni cuestionada. Es obvio que para un humano sería una buena estimación heurística que el coste de un estado fuese el número de libros que permanecen sin ser comprados; desafortunadamente, esta visión perspicaz no es obvia para un agente que evalúa su objetivo como una caja negra que devuelve un valor Verdadero o Falso para cada estado. Por tanto, al agente solucionador de problemas le falta autonomía; requiere de un humano que le suministre una función heurística para cada nuevo problema. Sin embargo si un agente planificador tiene acceso a una representación explícita del objetivo como una secuencia de subobjetivos, puede utilizar una simple heurística *independiente de dominio*: el número de conjunciones insatisfechas. Para el problema anterior de la compra del libro, el objetivo podría ser *Tener(A) \wedge Tener(B) \wedge Tener(C) \wedge Tener(D)*, y un estado contenido en *Tener(A) \wedge Tener(C)* podría presentar coste dos. De este modo, el agente automáticamente adquiere la heurística correcta para este problema y para otros. Veremos más tarde en el capítulo cómo construir heurísticas sofisticadas que examinen tanto las acciones disponibles como la estructura del objetivo.

DESCOMPOSICIÓN DE PROBLEMAS

PRÁCTICAMENTE DESCOMponIBLES

Finalmente, el solucionador de problemas podría ser ineficaz porque no pudiera aprovecharse de la **descomposición del problema**. Consideremos el problema de trasladar un conjunto de maletas de viaje a sus respectivos destinos, los cuales están distribuidos a lo largo de Australia. Una estrategia lógica sería buscar el aeropuerto más próximo a cada destino y dividir el problema total en varios subproblemas, uno por cada aeropuerto. Para cada conjunto de maletas asignadas a un aeropuerto determinado, el problema se descompone más profundamente, en función del destino concreto. Vimos en el Capítulo 5 que este tipo de descomposición contribuye a la eficiencia en la resolución de problemas que deben satisfacer restricciones impuestas. Esto es igualmente válido para planificadores: en el peor de los casos, el orden temporal para encontrar el mejor plan que entregase n paquetes sería $O(n!)$, pero solamente un orden temporal $O((n/k)! \times k)$ si el problema puede ser descompuesto en k partes iguales.

Como hicimos notar en el Capítulo 5, problemas perfectamente descomponibles no son frecuentes¹. El diseño de muchos sistemas de planificación (especialmente los planificadores de orden-parcial descritos en la Sección 11.3) están basados en la hipótesis de que la mayor parte de los problemas originados en contextos reales son **prácticamente descomponibles**. Esto es, el planificador puede trabajar siguiendo subobjetivos de manera independiente, aunque podría necesitar trabajo adicional para combinar el resultado de los subplanes generados. Para algunos problemas, esta hipótesis no es válida porque trabajar en la consecución de un subobjetivo probablemente deje de lado algún otro. Estas interacciones entre subobjetivos son uno de los motivos que hacen a los puzzles desconcertantes.

El lenguaje de los problemas de planificación

La discusión precedente sugiere que la representación de los problemas de planificación (estados, acciones y objetivos) debe hacer posible que los algoritmos de planificación se aprovechen de la estructura lógica del problema. La clave está en encontrar un lenguaje que sea suficientemente expresivo para describir un amplio rango de problemas, pero suficientemente restrictivo para permitir algoritmos operativos y eficientes. En esta sección, destacaremos en primer lugar el lenguaje de representación básico de los planificadores clásicos, conocido como el lenguaje STRIPS². Posteriormente, expondremos algunas de las diferentes modificaciones que se han desarrollado de lenguajes tipo-STRIPS.

Representación de estados. Los planificadores descomponen el mundo en términos de condiciones lógicas y representan un estado como una secuencia de literales positivos conectados. Consideraremos literales proposicionales: por ejemplo, *Pobre \wedge Desconocido* puede representar el estado de un agente desafortunado. Utilizaremos también literales de primer-orden como por ejemplo, *En(Avión₁, Melbourne) \wedge En(Avión₂, Sydney)*, que

¹ Nótese que incluso la entrega de un paquete no es un problema perfectamente descomponible. Pueden existir casos en los cuales sea mejor asignar paquetes a un aeropuerto más lejano si eso evita un vuelo a un aeropuerto más cercano innecesariamente. Sin embargo, muchas compañías de transporte prefieren la simplicidad organizacional y computacional de soluciones descomponibles.

² STRIPS significa *Stanford Research Institute Problem Solver*.

puede representar un estado en el problema del reparto de equipajes. Los literales, en un marco de descripciones de primer orden, deben ser **simples** y **sin dependencias funcionales**. Por ejemplo, literales del tipo $En(x, y)$ o $En(Padre(Fred), Sydney)$ no se permiten. La **hipótesis de un mundo cerrado** es asumida por la que todas las condiciones que no son mencionadas en un estado, se asume que son falsas.

Representación de objetivos. Un objetivo es un estado parcialmente especificado, representado como una secuencia de literales positivos y simples, tales como $Rico \wedge Famoso$ o $En(P_2, Tahiti)$. Un estado proposicional s **satisface** un objetivo g si s contiene todos sus elementos en g (y posiblemente otros además). Por ejemplo, el estado $Rico \wedge Famoso \wedge Miserable$ satisface el objetivo $Rico \wedge Famoso$.

Representación de acciones. Una acción es especificada en términos de las precondiciones que deben cumplirse antes de ser ejecutada y de las consecuencias que se siguen cuando se ejecuta. Por ejemplo, la acción que nos indica cómo un avión vuela desde una ciudad a otra puede exponerse como:

*Acción (avión(p , *desde*, *hasta*),*

PRECOND: $En(p, \text{desde}) \wedge \text{avión}(p) \wedge \text{aeropuerto}(\text{desde}) \wedge \text{aeropuerto}(\text{hasta})$

EFFECTO: $\neg En(p, \text{desde}) \wedge En(p, \text{hasta})$

SATISFACCIÓN DE OBJETIVOS

ESQUEMA DE ACCIÓN

PRECONDICIÓN

EFFECTO

LISTA AÑADIR

LISTA BORRAR

Esto es más propio llamarlo **esquema de acción**, para indicar que representa un número de diferentes acciones que pueden ser derivadas mediante la instanciación de las variables p , *desde* y *hasta* pudiendo adquirir diferentes valores. En general, un esquema de acción consta de tres partes:

- El nombre de la acción y la lista de parámetros de los que depende la acción (por ejemplo, $Volar(p, \text{desde}, \text{hasta})$ sirve para identificar la acción).
- La **precondición** es la unión de literales positivos sin dependencia funcional estableciendo lo que debe ser verdad en un estado antes de que una acción sea ejecutada. Todas las variables en las precondiciones deben también aparecer en la lista de parámetros de acción.
- El **efecto** es la unión de literales sin dependencia funcional describiendo cómo el estado cambia cuando la acción es ejecutada. Un literal positivo P en el efecto se espera que sea verdadero en el estado resultante de la acción, mientras que un literal negativo $\neg P$ se espera que sea falso. Las variables en el efecto deben pertenecer a la lista de parámetros de acción.

Para hacer más sencilla la legibilidad, algunos sistemas de planificación dividen el efecto en dos listas de literales: los positivos en la **lista Añadir**, y los negativos en la **lista Borrar**.

Hasta ahora hemos definido la sintaxis en la representación de problemas de planificación; veremos ahora cómo definir la semántica. El modo más sencillo es mediante la descripción de cómo las acciones afectan a los estados (un método alternativo consiste en especificar una traslación directa a un conjunto de axiomas de estado sucesivo, cuya semántica viene directamente de las reglas de la lógica de primer orden; véase Ejercicio 11.3). Diremos que una acción es **aplicable** en cualquier estado que satisfaga sus precondiciones; en otro caso, la acción no tendrá efecto. La aplicación, para un esque-

APLICABLE

ma de acción de primer orden, se reduce a la sustitución θ de las variables en la precondición. Por ejemplo, supongamos que el estado actual está descrito por

$$\begin{aligned} & En(P_1, JFK) \wedge En(P_2, SFO) \wedge Avión(P_1) \wedge Avión(P_2) \\ & \wedge Aeropuerto(JFK) \wedge Aeropuerto(SFO) \end{aligned}$$

Este estado satisface la precondición

$$En(p, \text{desde}) \wedge Avión(p) \wedge Aeropuerto(\text{desde}) \wedge Aeropuerto(\text{hasta})$$

con la sustitución $\{p/P_1, \text{desde}/JFK, \text{hasta}/SFO\}$ (junto a otros; véase Ejercicio 11.2). De esta forma, la acción concreta $Volar(P_1, JFK, SFO)$ es aplicable.

RESULTADO

Comenzando en el estado s , el **resultado** de ejecutar una acción aplicable a nos lleva a otro estado s' , que es el mismo que s excepto que cualquier literal positivo P en el efecto de a es añadido a s' , y cualquier literal negativo $\neg P$ es eliminado de s' . De este modo, después de $Volar(P_1, JFK, SFO)$, el estado actual se convierte en

$$\begin{aligned} & En(P_1, SFO) \wedge En(P_2, SFO) \wedge Avión(P_1) \wedge Avión(P_2) \\ & \wedge Aeropuerto(JFK) \wedge Aeropuerto(SFO) \end{aligned}$$

HIPÓTESIS STRIPS

Notemos que si un efecto positivo está ya en s no se añade dos veces, y si un efecto negativo no está en s , entonces esa parte del efecto es ignorada. Esta definición expresa la llamada **hipótesis STRIPS**: cada literal no mencionado en el efecto permanece sin modificar. De este modo, STRIPS evita el **problema del marco** representacional descrito en el Capítulo 10.

SOLUCIÓN

Finalmente, definimos la **solución** de un problema de planificación. En su forma más sencilla, es simplemente una secuencia de acciones que, ejecutada en el estado inicial, da como resultado un estado final que satisface el objetivo. Posteriormente, en este mismo capítulo, describiremos las soluciones como conjuntos de acciones parcialmente ordenados, siempre que cada secuencia de acciones que respete el orden parcial sea solución.

Expresividad y extensiones

Las diferentes restricciones impuestas por la representación STRIPS fueron elegidas con el deseo de diseñar algoritmos más simples y más eficientes, sin complicarlo demasiado para poder describir problemas reales. Una de las restricciones más importantes impuestas es que los literales no tengan *dependencia funcional* de otros atributos. Con esta restricción, estamos seguros de que dado un problema, todo sistema de acción puede ser proposicionalizado, esto es, transformado en una colección finita de representaciones de acciones estrictamente proposicionales (véase Capítulo 9). Por ejemplo, en el dominio del transporte aéreo para un problema de 10 aviones y cinco aeropuertos, podemos convertir la sentencia $Volar(p, \text{desde}, \text{hacia})$ en $10 \times 5 \times 5 = 250$ acciones puramente proposicionales. Los planificadores en los Apartados 11.4 y 11.5 trabajan directamente con descripciones proposicionalizadas. Si admitimos símbolos funcionales, podremos construir infinitamente muchos más estados y acciones.

En los últimos años, se ha mostrado claramente que STRIPS no posee expresividad suficiente para ciertos dominios reales. Como resultado, muchos lenguajes han sido

desarrollados. La Figura 11.1 describe brevemente uno de los más importantes, el Lenguaje de Descripción de Acciones (*Action Description Language, ADL*), y la comparación con el lenguaje STRIPS. En ADL, la acción *Volar* podría ser escrita como

*Acción (Volar(p : Avión, desde : Aeropuerto, hasta : Aeropuerto),
PRECOND: $\neg En(p, \text{desde}) \wedge (\text{desde} \neq \text{hasta})$
EFFECTO: $\neg En(p, \text{desde}) \wedge en(P, \text{hasta})$)*

La notación $p : Avión$ en la lista de parámetros es una abreviatura de $Avión(p)$ en la sintaxis de precondición; este cambio no añade valor expresivo, pero es más sencillo de leer (también reduce el número de posibles acciones proposicionales que pueden ser construidas). La precondición ($\text{desde} \neq \text{hasta}$) expresa el hecho de que un vuelo no discurre entre un aeropuerto y él mismo. Esto podría no haber sido expresado suficientemente por STRIPS.

Los diferentes formalismos de planificación usados en la IA han sido sistematizados dentro de una sintaxis estándar llamada Lenguaje de Definición de Dominios para la Planificación (*Planning Domain Definition Language, o PDDL*). Este lenguaje permite a los investigadores el intercambio y la comparación de problemas y resultados. PDDL incluye sublenguajes para STRIPS, ADL y para las redes jerárquicas de tareas que veremos en el Capítulo 12.

Lenguaje STRIPS	Lenguaje ADL
Sólo literales positivos en estados: <i>Pobre</i> \wedge <i>Desconocido</i>	Literales positivos y negativos en estados: $\neg Rico \wedge \neg Famoso$
Hipótesis de Mundo Cerrado: Los literales no mencionados son falsos	Hipótesis de Mundo Abierto: Los literales no mencionados son desconocidos
El efecto de $P \wedge \neg Q$ significa añadir P y eliminar Q	El efecto de $P \wedge \neg Q$ significa añadir P y $\neg Q$ y eliminar $\neg P$ y Q
Sólo literales simples en objetivos: <i>Rico</i> \wedge <i>Famoso</i>	Variables cuantificadas en objetivos: $\exists x En(P_1, x) \wedge En(P_2, x)$ es el objetivo de tener P_1 y P_2 en el mismo lugar
Los objetivos son conjunciones: <i>Rico</i> \wedge <i>Famoso</i>	Se permiten conjunciones y disyunciones en los objetivos: $\neg Pobre \wedge (Famoso \vee Inteligente)$
Los efectos son conjunciones	Se permiten efectos condicionales: cuando P : E significa que E es un efecto sólo si P es satisfecho
No tiene infraestructura para soportar igualdades	Predicados de igualdad ($x = y$) son admisibles
No tiene infraestructura para soportar tipos	Las variables pueden tener tipos, como (p : Avión)

Figura 11.1 Comparación de lenguajes STRIPS y ADL para la representación de problemas de planificación. En ambos casos, los objetivos se comportan como las precondiciones de una acción sin parámetros.

CONSTRICIONES
DE ESTADO

Tanto la notación STRIPS como ADL son adecuadas para muchos dominios reales. Las subsecciones que siguen muestran algunos ejemplos. Sin embargo, existen algunas restricciones significativas. La más obvia es que no pueden representar de un modo natural las **ramificaciones** de las acciones. Por ejemplo, si existe gente, paquetes o motas de polvo en el avión, todas éstas cambiarán de localización cuando el avión vuela. Podemos representar estos cambios como el efecto directo de volar, considerando que parece más natural representar la localización del contenido del avión como la consecuencia lógica de la localización del mismo. Veremos otros ejemplos de estados con **constricciones de estado** en la Sección 11.5. Los sistemas de planificación clásica no son capaces de tratar problemas de **requisitos**: el problema de no representar circunstancias que puedan causar que una acción fracase. Veremos cómo tratar con los requisitos en el Capítulo 12.

Ejemplo: transporte de carga aéreo

La Figura 11.2 muestra cómo un problema de transporte de carga aéreo lleva asociado procesos de carga y descarga entre aviones que vuelan entre diferentes destinos. El problema puede ser definido con tres acciones: *Carga*, *Descarga*, y *Vuelo*. Las acciones afectan a dos predicados: *Dentro(c, p)* significa que la carga *c* está dentro del avión *p*, y *En(x, a)* significa que el objeto *x* (tanto avión como carga) está en el aeropuerto *a*. Notemos que la carga no está *En* cualquier sitio cuando se encuentra *Dentro* de un avión concreto, por tanto *En* realmente significa «disponible para su uso en una localización determinada». El siguiente plan es una solución al problema:

$$\begin{aligned} & [Carga(C_1, P_1, SFO), Volar(P_1, SFO, JFK), \\ & \quad Carga(C_2, P_2, JFK), Volar(P_2, JFK, SFO)] \end{aligned}$$

Nuestra representación es estrictamente STRIPS. En particular, se permite que un avión vuele hacia y desde el mismo aeropuerto. Literales de tipo desigualdades, podrían prevenir este tipo de situaciones.

```

Iniciar (En(C1, SFO) ∧ En(C2, JFK) ∧ En(P1, SFO) ∧ En(P2, JFK))
        ∧ Carga(C1) ∧ Carga(C2) ∧ Avión(P1) ∧ Avión(P2)
        ∧ Aeropuerto(JFK) ∧ Aeropuerto(SFO))
Objetivo(En(C1, JFK) ∧ En(C2, SFO))
Acción (Cargar(c, p, a),
        PRECOND: En (c, a) ∧ En(p, a) ∧ Carga(c) ∧ Avión(p) ∧ Aeropuerto(a)
        EFECTO: ¬En (c, a) ∧ Dentro (c, p))
Acción (Descargar(c, p, a),
        PRECOND: Dentro (c, p) ∧ En (p, a) ∧ Carga(c) ∧ Avión(p) ∧ Aeropuerto(a)
        EFECTO: En (c, a) ∧ ¬Dentro (c, p))
Acción (Volar(p, desde, hasta),
        PRECOND: En (p, desde) ∧ Avión(p) ∧ Aeropuerto(desde) ∧ Aeropuerto(hasta)
        EFECTO: ¬En (p, desde) ∧ En (p, hasta))

```

Figura 11.2 Problema STRIPS de transporte de carga aérea entre aeropuertos.

Ejemplo: el problema de la rueda de recambio

Considérese el problema de cambiar una rueda pinchada. De modo más preciso, el objetivo es tener la rueda de repuesto montada correctamente en el eje del coche, mientras que el estado inicial consiste en la rueda pinchada sobre el eje y la rueda de repuesto en el maletero. Para exponerlo sencillamente, nuestra versión del problema es abstracta, sin considerar otras complicaciones. Simplemente contamos con cuatro acciones: sacar la rueda del maletero, quitar la rueda pinchada del eje, colocar la rueda nueva en el eje y dejar el coche sin vigilancia durante la noche. Se asume que el coche queda aparcado en un barrio peligroso, de modo que el efecto de dejarlo sin vigilancia puede ser el que roben las ruedas.

La descripción ADL del problema se muestra en la Figura 11.3. Notemos que es estrictamente proposicional. Vamos más allá que con el simple uso de lenguaje STRIPS, pues se usan precondiciones negativas, $\neg En(Deshinchada, Eje)$, para la acción *Colocar(Repuesto, eje)*. Esto podría ser evitado mediante la utilización de *Despejar(eje)*, como veremos en el siguiente ejemplo.

```

Iniciar (En(Deshinchada, Eje)  $\wedge$  En(Repuesto, maletero))
Objetivo(En(Repuesto, eje))
Acción (Quitar, (Repuesto, maletero)),
  PRECOND: En(Repuesto, maletero)
  EFECTO:  $\neg En(Repuesto, maletero) \wedge En(Repuesto, Suelo)$ 
Acción (Quitar, (Deshinchada, Eje)),
  PRECOND: En(Deshinchada, Eje),
  EFECTO:  $\neg En(Deshinchada, Eje) \wedge En(Deshinchada, Suelo)$ 
Acción (Colocar(Repuesto, eje)).
  PRECOND: En(Repuesto, Suelo)  $\wedge$   $\neg En(Deshinchada, Eje)$ 
  EFECTO:  $\neg En(Repuesto, Suelo) \wedge En(Repuesto, Eje)$ )
Acción (DejarloDeNoche),
  PRECOND:
  EFECTO:  $\neg En(Repuesto, Suelo) \wedge \neg En(Repuesto, Eje) \wedge \neg En(Repuesto, maletero)$ 
 $\wedge \neg En(Deshinchada, Suelo) \wedge \neg En(Deshinchada, Eje))$ 

```

Figura 11.3 El problema simplificado de la rueda de repuesto.

Ejemplo: el mundo de los bloques

Uno de los más famosos dominios de planificación es conocido como el **mundo de los bloques**. Este dominio consiste en un conjunto de bloques con forma de cubo situados en un tablero³. Los bloques pueden ser amontonados, pero únicamente podemos situar uno sobre otro. Un brazo mecánico puede cambiar bloques de sitio, tanto sobre el ta-

³ El mundo de los bloques utilizado en planificación es mucho más simple que la versión SHRDLU'S, mostrada en el Apartado 1.3.

blero como sobre otros bloques. El brazo puede tomar sólo un bloque por cada instante de tiempo, de modo que no puede tomar uno si aún no ha soltado otro anterior. El objetivo será construir uno o más montones de bloques, que quedarán especificados en términos de cuántos bloques están sobre cuántos otros bloques. Por ejemplo, podríamos tener un objetivo que fuese colocar un bloque A sobre otro B y uno C sobre otro D .

Usemos $Sobre(b, x)$ para indicar que el bloque b se encuentra sobre x , donde x puede ser también otro bloque sobre el tablero. La acción para mover el bloque b desde el lugar x hasta el lugar y será $Mover(b, x, y)$. Una de las precondiciones para poder mover b es que ningún otro bloque se encuentre sobre él. En una lógica de primer orden, esta idea podría ser expresada como $\neg \exists x Sobre(x, b)$ o, alternativamente, $\forall x \neg Sobre(x, b)$. Éstas podrían ser establecidas como precondiciones en ADL. Podemos expresarlas dentro de un lenguaje STRIPS, sin embargo, añadiendo un nuevo predicado, $Despejar(x)$, que es verdadero siempre que ningún bloque esté sobre x .

La acción $Mover$ cambia un bloque b desde x hasta y si tanto b como y están «despejados». Después de que el movimiento sea ejecutado, x estará despejado pero y no lo estará. Una descripción formal de $Mover$ en STRIPS es:

Acción(Mover (b, x, y),
PRECOND: Sobre(b, x) \wedge Despejar(b) \wedge Despejar(y),
EFFECTO: Sobre(b, y) \wedge Despejar(x) \wedge \neg Sobre(b, x) \wedge \neg Despejar(y))

Desafortunadamente, esta acción no mantiene $Despejar$ adecuadamente cuando x o y se encuentran sobre el tablero. Cuando $x = Tablero$, esta acción tiene el efecto $Despejar(Tablero)$, pero el tablero no debe estar vacío; de igual modo cuando $y = Tablero$. Para determinar estas situaciones, podemos hacer dos cosas. En primer lugar, incluimos otra acción para mover un bloque b desde x al tablero:

Acción (MoverSobreTablero(b, x),
PRECOND: Sobre(b, x) \wedge Despejar(b)),
EFFECTO: Sobre(b, Tablero) \wedge Despejar(x) \wedge \neg Sobre(b, x))

```
Iniciar(Sobre(A, Tablero)  $\wedge$  Sobre(B, Tablero)  $\wedge$  Sobre(C, Tablero)
 $\wedge$  Bloque(A)  $\wedge$  Bloque(B)  $\wedge$  Bloque(C)
 $\wedge$  Despejar(A)  $\wedge$  Despejar(B)  $\wedge$  Despejar(C))
Objetivo (Sobre(A, B)  $\wedge$  Sobre(B, C))
Acción ( Mover (b, x, y),
PRECOND: Sobre(b, x)  $\wedge$  Despejar(b)  $\wedge$  Despejar(y)  $\wedge$  Bloque (b)  $\wedge$ 
(b  $\neq$  x)  $\wedge$  (b  $\neq$  y)  $\wedge$  (x  $\neq$  y),
EFFECTO: Sobre(b, y)  $\wedge$  Despejar(x)  $\wedge$   $\neg$ Sobre(b, x)  $\wedge$   $\neg$ Despejar(y))
Acción (MoverSobreTablero(b, x),
PRECOND: Sobre(b, x)  $\wedge$  Despejar(b)  $\wedge$  Bloque (b)  $\wedge$  (b  $\neq$  x),
EFFECTO: Sobre(b, Mesa)  $\wedge$  Despejar(x)  $\wedge$   $\neg$ Sobre(b, x))
```

Figura 11.4 Un problema de planificación en el mundo de los bloques: construir una torre de tres bloques. Una solución es la secuencia [$Mover(B, Tablero, C)$, $Mover(A, Tablero, B)$].

En segundo lugar, tomaremos la interpretación de *Despejar(b)* como «existe un espacio vacío sobre *b* para trasladar un bloque». Bajo esta interpretación, *Despejar(Tablero)* siempre será correcto. El único problema es que nada le impide al planificador usar *Mover(b, x, Tablero)* en lugar de *MoverSobreTablero(b, x)*. Podríamos vivir con este problema (nos llevará a un espacio de búsqueda mayor que el necesario, pero no se tendrán respuestas incorrectas) o podemos incluir el predicado *Bloque* y añadir *Bloque(b) ∧ Bloque(y)* como precondición de *Mover*.

Finalmente, existe el problema de las acciones espurias tales como *Mover(B, C, C)*, que tiene efectos contradictorios. Es común ignorar tales problemas, porque casi nunca provocan planes incorrectos. El enfoque correcto añade precondiciones de tipo desigualdad como se muestra en la Figura 11.4.

11.2 Planificación con búsquedas en espacios de estado

Prestemos atención a los algoritmos de planificación. El enfoque más sencillo es el uso de una búsqueda en un espacio de estados. Es conocido que las descripciones de las acciones en un problema de planificación especifican tanto las precondiciones como los efectos, por tanto son posibles las búsquedas en ambas direcciones: búsquedas hacia-delante desde el estado inicial o búsquedas hacia-atrás desde el estado final, como se muestra en la Figura 11.5. También usaremos las representaciones de objetivos y acciones explícitas para derivar automáticamente heurísticas efectivas.

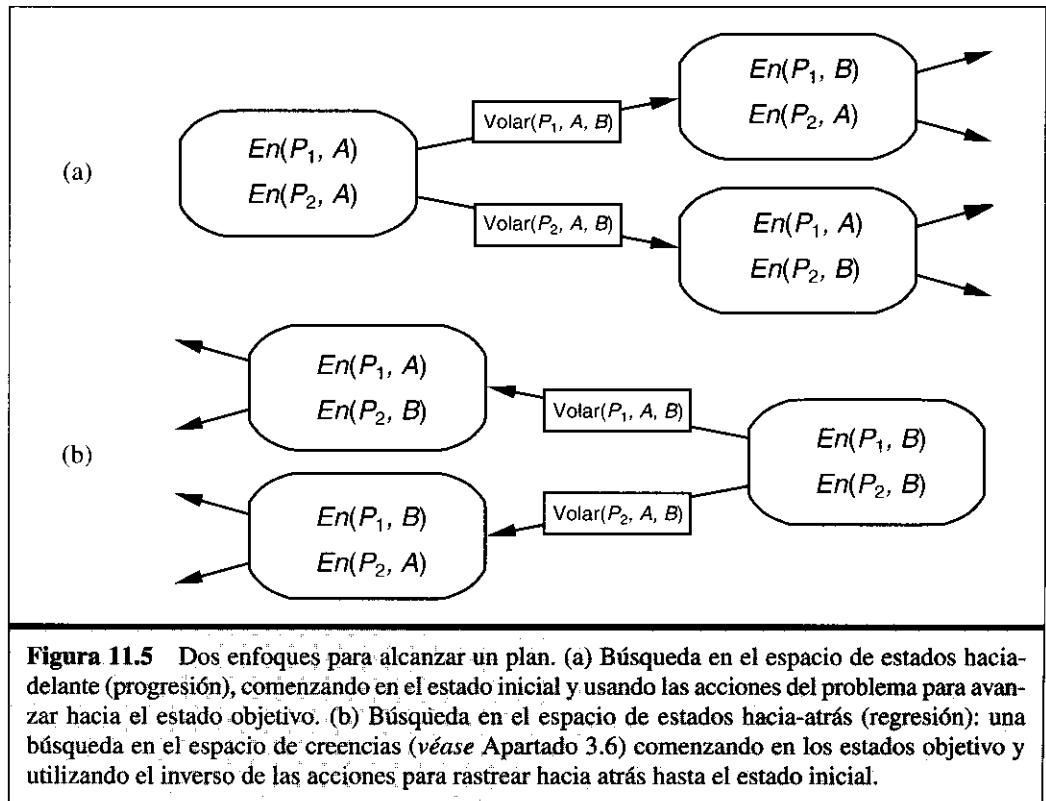
Búsquedas hacia-delante en el espacio de estados

PROGRESIÓN

La planificación mediante búsquedas hacia-delante en el espacio de estados es similar al enfoque que veíamos en el Capítulo 3 para resolver problemas. Algunas veces se le llama planificación de **progresión**, porque mantiene una dirección de avance.

Comenzamos en el estado inicial del problema, considerando secuencias de acciones hasta que encontramos una secuencia que alcance un estado objetivo. La formulación de problemas de planificación como problemas de búsqueda en un espacio de estados es como sigue:

- El **estado inicial** de la búsqueda es el estado inicial del problema de planificación. En general, cada estado será un conjunto de literales simples y positivos; los literales que no aparecen expresados se asume que son falsos.
- Las **acciones** que son aplicables en un estado son todas aquellas cuyas precondiciones son satisfechas. El estado resultante de una acción es generado añadiendo literales positivos y eliminando los negativos. Notemos que una simple función sucesor trabaja para todos los problemas de planificación (una consecuencia de utilizar una representación de acción explícita).
- El **test de objetivos** chequea si el estado satisface el objetivo del problema de planificación.



- El **coste del paso** entre acciones es típicamente 1. Aunque podría ser sencillo permitir diferentes costes para diferentes acciones, esto es rara vez utilizado por los planificadores STRIPS.

Recalquemos que, en ausencia de símbolos funcionales, el espacio de estados de un problema de planificación es finito. Por tanto, cualquier algoritmo de búsqueda en grafos que es completo (por ejemplo, A^*) será un algoritmo de planificación completa.

Desde los inicios de la investigación en planificación (en torno a 1961) hasta la actualidad (1998) se ha constatado que la búsqueda hacia-delante en un espacio de estados es ineficaz en la práctica. No es difícil argumentar esta posición con varios resultados, simplemente revisando la Sección 11.1. En primer lugar, las búsquedas hacia-delante no son capaces de tratar con el problema de acciones irrelevantes, todas las posibles acciones aplicables son consideradas desde el estado en el que nos encontramos. En segundo lugar, este enfoque rápidamente queda empantanado sin una buena heurística. Consideremos el problema del transporte de carga aéreo con los siguientes datos, 10 aeropuertos, cada aeropuerto posee cinco aviones y 20 piezas para ser cargadas. El objetivo es trasladar toda la carga desde el aeropuerto A al B . Existe una solución muy simple para este problema: descargar las 20 piezas en uno de los aviones en A , volar hasta B y proceder a su descarga. Sin embargo, encontrar la solución puede ser difícil porque el factor promedio de ramificación es enorme: cada uno de los 50 aviones puede volar a otros nueve aeropuertos, y cada uno de los 200 paquetes pueden ser tanto descargados (si antes

fueron cargados) o cargados dentro de cada avión en un aeropuerto (si se encuentran descargados). En promedio, tenemos unas 1.000 acciones posibles, de modo que el árbol de búsqueda para encontrar la solución obvia tiene del orden de 1.000^{41} nodos. Es evidente que una heurística adecuada al problema es necesaria para hacer este tipo de búsqueda eficiente. Discutiremos algunas posibles heurísticas después de presentar la búsqueda hacia-atrás.

Búsquedas hacia-atrás en el espacio de estados

Las búsquedas hacia-atrás en el espacio de estados fueron descritas brevemente como parte de búsquedas bidireccionales en el Capítulo 3. Destacamos, en primer lugar, que la búsqueda hacia-atrás puede ser difícil de implementar cuando el estado objetivo es descrito por un conjunto de restricciones más que siendo explícitamente especificado. En particular, no es siempre obvio cómo generar una descripción de los posibles estados **predecesores** del conjunto de estados objetivo. Veremos que la representación de un problema de planificación mediante STRIPS se simplifica porque el conjunto de estados puede ser descrito por el conjunto de literales que deben verificarse en ellos.

La principal ventaja de las búsquedas hacia-atrás es que nos permiten considerar solamente acciones **relevantes**. Una acción es relevante para una secuencia encadenada de objetivos si alcanza un conjunto de ellos. Por ejemplo, el objetivo en nuestro problema de transporte de carga aérea considerando 10 aeropuertos, es tener 20 piezas de carga en el aeropuerto B , o de manera más precisa:

$$En(C_1, B) \wedge En(C_2, B) \wedge \dots \wedge En(C_{20}, B)$$

Ahora consideremos el conjunto $En(C_1, B)$. Trabajando hacia-atrás, podremos obtener acciones que tengan ésta como efecto. Por ejemplo, una de ellas: $Descargar(C_1, p, B)$, donde el avión p no está especificada.

Notemos que existen muchas acciones *irrelevantes* que pueden dirigirnos hacia un estado objetivo. Por ejemplo, podemos volar en un avión vacío desde *JFK* hasta *SFO*; esta acción alcanza un estado objetivo desde un estado predecesor en el que el avión está en *JFK* y todo el conjunto de objetivos es satisfecho. Una búsqueda hacia-atrás que permita acciones irrelevantes será completa pero mucho menos eficiente. Si la solución existe, será encontrada por una búsqueda hacia atrás que permita solamente acciones relevantes. La restricción a acciones relevantes se traduce en búsquedas hacia atrás que tienen un factor de *ramificación* mucho menor que en las búsquedas hacia delante. Por ejemplo, nuestro problema de carga aérea tiene cerca de 1.000 acciones dirigidas hacia delante desde el estado inicial pero sólo 20 acciones hacia atrás desde el estado objetivo.

Las búsquedas hacia atrás son conocidas habitualmente como planificación por **regresión**. La principal pregunta en la planificación por regresión es: ¿Cuáles son los estados desde los cuales iniciar una acción determinada que nos dirige hacia el objetivo? La descripción de estos estados se conoce como la regresión de un objetivo a través de la acción. Para ver cómo hacer esto, consideremos el ejemplo del transporte de carga aéreo. Tenemos el siguiente objetivo:

$$En(C_1, B) \wedge En(C_2, B) \wedge \dots \wedge En(C_{20}, B)$$

y la acción relevante $\text{Descargar}(C_1, p, B)$, que alcanza el primer término. La acción será adecuada solamente si sus precondiciones son satisfechas. Sin embargo, cualquier estado predecesor debe incluir las precondiciones siguientes: $\text{Dentro}(C_1, p) \wedge \text{En}(p, B)$. Sin embargo, el subconjunto $\text{En}(C_1, B)$ no debería ser cierto en el estado predecesor⁴. De este modo, la descripción precedente es

$$\text{Dentro}(C_1, p) \wedge \text{En}(p, B) \wedge \text{En}(C_2, B) \wedge \dots \wedge \text{En}(C_{20}, B)$$

CONSISTENCIA

Además de insistir en que las acciones alcancen algunos literales deseados, debemos insistir en que las acciones *no deshagan* ningún literal deseado. Una acción que satisfaga dicha restricción es llamada **consistente**. Por ejemplo, la acción $\text{Cargar}(C_2, p)$ podría no ser consistente con el objetivo actual, porque podría negar el literal $\text{En}(C_2, B)$.

Dadas definiciones de relevancia y consistencia, podemos describir el proceso general de construcción de predecesores mediante búsquedas hacia-atrás. Sea la descripción de un objetivo G , y supongamos que A sea una acción que es relevante y consistente. El correspondiente predecesor es como sigue:

- Cualquier efecto positivo de A que aparezca en G es eliminado.
- Cada precondición literal de A es añadida, a no ser que ya apareciese.

Cualquiera de los algoritmos de búsqueda estándar pueden ser usados para llevar adelante la búsqueda. Se finaliza el proceso cuando la descripción de un predecesor generada es satisfecha por el estado inicial del problema de planificación. En el caso de primer-orden, la satisfacción puede requerir la sustitución por variables en la descripción del precedente. Por ejemplo, la descripción del precedente en el párrafo anterior es satisfecha por el estado inicial

$$\text{Dentro}(C_1, P_{12}) \wedge \text{En}(P_{12}, B) \wedge \text{En}(C_2, B) \wedge \dots \wedge \text{En}(C_{20}, B)$$

sustituyendo $\{p/P_{12}\}$. La sustitución debe ser aplicada a las acciones que se dirigen hacia el objetivo, produciendo la solución $[\text{Descarga}(C_1, P_{12}, B)]$.

Heurísticas para la búsqueda en el espacio de estados

Se ha indicado previamente que tanto para que algoritmos de búsqueda hacia-delante como hacia-atrás sean eficientes deben utilizar una función heurística adecuada. Recordemos del Capítulo 4 que una función heurística estima la distancia desde un estado al objetivo; en la planificación STRIPS, el coste de cada acción es 1, de modo que la distancia es el número de acciones. La idea básica es observar los efectos de las acciones y los objetivos que deben ser alcanzados y estimar cuántas acciones son necesarias para alcanzar todos los objetivos. Encontrar el número exacto es un problema NP-completo, pero es posible encontrar razonables estimaciones en muchos casos sin demasiado gasto computacional. Podríamos, de igual modo, ser capaces de derivar una heurística **admisible**, esto es, que no sobreestime. Podríamos usar A* para encontrar soluciones óptimas.

⁴ Si el subobjetivo fuese verdadero en el estado predecesor, la acción podría aún dirigirse hasta el estado objetivo. Por otro lado, tales acciones son irrelevantes porque no *hacen* el objetivo verdadero.

INDEPENDENCIA DE SUB-OBJETIVOS

Existen dos enfoques que deben ser mencionados. El primero consiste en derivar un **problema aproximado** desde las especificaciones del problema dado, tal como se describió en el Capítulo 4. El coste de la solución óptima del problema aproximado (que suponemos sencillo de resolver) nos da una heurística admisible para el problema original. El segundo enfoque pretende usar un simple algoritmo divide-y-vencerás. Este planteamiento asume la **independencia de sub-objetivos**: el coste de resolver una secuencia de sub-objetivos es aproximadamente la suma de los costes que supone resolver cada uno de los sub-objetivos *independientemente*. La hipótesis de la independencia de sub-objetivos puede ser optimista o pesimista. Se llama optimista cuando existen interacciones negativas entre los subplanes de cada sub-objetivo, por ejemplo, cuando una acción en un sub-plan hace fracasar el objetivo a alcanzar por otro sub-plan. Es pesimista, y por tanto inadmisible, cuando los sub-planes contienen acciones redundantes, por ejemplo, dos acciones que podrían ser reemplazadas por una acción sencilla en un plan conjunto.

Consideremos cómo obtener problemas aproximados a partir de uno dado. Si tenemos disponibles las representaciones explícitas de las precondiciones y de los efectos, el proceso consistirá en la modificación de dichas representaciones (comparar este enfoque con los problemas de búsqueda, donde la función sucesora es una caja negra). La idea más simple para calcular un problema aproximado es mediante la *eliminación de todas las precondiciones* de las acciones. En esta situación, todas las acciones serán aplicables, y cualquier literal puede ser alcanzado en cada etapa (si existe una acción que es aplicable, en caso contrario, el objetivo es imposible). Esto prácticamente implica que el número de pasos requeridos para resolver una secuencia de objetivos es el número de objetivos no satisfechos, prácticamente pero no totalmente porque (1) puede haber dos acciones, cada una de las cuales elimine el literal del objetivo alcanzado por el otro, y (2) alguna acción puede alcanzar múltiples objetivos. Si combinamos el problema aproximado junto con la hipótesis de la independencia de sub-objetivos, ambas se encuentran lejos de que la heurística resultante sea exactamente el número de objetivos no satisfechos por alcanzar.

En algunos casos, una heurística más adecuada se obtiene mediante la consideración de interacciones positivas que surgen de las acciones que permiten alcanzar múltiples objetivos. En primer lugar, derivamos el problema aproximado mediante la *eliminación de efectos negativos* (véase Ejercicio 11.6). Después, contamos el mínimo número de acciones requeridas tales que la unión de los efectos de acciones positivas satisfagan el objetivo. Por ejemplo, consideremos

$$\begin{aligned}
 & \text{Objetivo}(A \wedge B \wedge C) \\
 & \text{Acción}(X, \text{EFFECTO: } A \wedge P) \\
 & \text{Acción}(Y, \text{EFFECTO: } B \wedge C \wedge Q) \\
 & \text{Acción}(Z, \text{EFFECTO: } B \wedge P \wedge Q)
 \end{aligned}$$

El mínimo conjunto que cubre el objetivo $\{A, B, C\}$ viene dado por las acciones $\{X, Y\}$, de modo que el conjunto que cubre la heurística devuelve un coste de 2. Esto mejora la hipótesis de la independencia de sub-objetivos, cuya heurística nos da un valor de 3. Existe un pequeño inconveniente: el problema es NP-duro. Un algoritmo simple que cubra el conjunto garantiza el retorno de un valor que esté dentro de un orden $\log n$ del valor mínimo verdadero, donde n es el número de literales en el objetivo; normalmente fun-

ciona bien en la práctica. Desafortunadamente un algoritmo de este tipo pierde la garantía de admisibilidad para la heurística.

Es también posible generar problemas aproximados eliminando efectos negativos sin eliminar precondiciones. Esto es, si una acción tiene el efecto $A \wedge \neg B$ en el problema original, tendrá el efecto A en el problema aproximado. Esto significa que no necesitamos preocuparnos acerca de las interacciones negativas entre sub-planes, porque ninguna acción puede eliminar los literales alcanzados por otra acción. El coste de la solución del resultante problema aproximado da lo que se conoce como heurística para **suprimir listas vacías**. La heurística es bastante precisa, pero ponerla a funcionar lleva asociado ejecutar un algoritmo de planificación simple. En la práctica, la búsqueda en el problema aproximado es suficientemente rápida como para que el coste merezca la pena.

Las heurísticas descritas aquí pueden ser usadas tanto en dirección de progresión como regresión. Actualmente, los planificadores de progresión que usan la heurística para suprimir listas vacías marcan la tendencia. Es probable que se produzcan cambios y que nuevas heurísticas y nuevas técnicas de búsqueda sean exploradas. Dado que la planificación es exponencialmente difícil⁵, ningún algoritmo será eficiente para todos los problemas, pero una gran cantidad de problemas prácticos pueden ser resueltos con los métodos heurísticos mencionados en este capítulo, muchos más que los que se podían resolver solamente hace unos pocos años.

SUPRIMIR LISTAS VACÍAS

11.3 Planificación ordenada parcialmente

Las búsquedas en el espacio de estados hacia-delante y hacia-atrás son tipos de planes de búsqueda *totalmente ordenados*. Sólo exploran secuencias estrictamente lineales de acciones conectadas directamente al inicio o al objetivo. Esto significa que no se puede sacar provecho de la descomposición del problema. Preferiblemente que trabajar sobre cada subproblema separadamente, se deben tomar decisiones sobre el orden en que se sucedan las acciones desde todos los subproblemas. Preferiremos un enfoque que trabaje en varios sub-objetivos independientemente, los solucione con varios sub-planes, y por último, combine el conjunto de sub-planes utilizados.

Este enfoque presenta la ventaja de flexibilizar el orden en el que se *construye* el plan. Es decir, el planificador puede trabajar sobre «obvias» o «importantes» decisiones primariamente, antes que estar forzado a trabajar siguiendo las etapas en un orden cronológico. Por ejemplo, un agente planificador que se encuentre en Berkeley y desee viajar a Monte Carlo podría primero intentar encontrar un vuelo desde San Francisco a París; dada información acerca de los horarios de partida y llegada, puede ponerse a trabajar en los modos de salir y llegar a los aeropuertos.

La estrategia general de aplazar una opción durante la búsqueda se conoce como una estrategia de **mínimo compromiso**. No damos una definición formal, aunque claramente

MÍNIMO COMPROMISO

⁵ Técnicamente, la planificación tipo STRIPS es PSPACE-completa a menos que tengan sólo precondiciones positivas y sólo un literal como efecto (Bylander, 1994).

un grado de compromiso es necesario, dado que si no, la búsqueda podría no progresar. Dejando a un lado esta informalidad, el mínimo compromiso es un concepto útil para analizar cuándo las decisiones deben llevarse a cabo en cualquier problema de búsqueda.

Nuestro primer ejemplo concreto será más simple que planificar unas vacaciones. Consideremos el sencillo problema de ponerse un par de zapatos. Podemos describirlo como un problema de planificación formal como sigue:

```

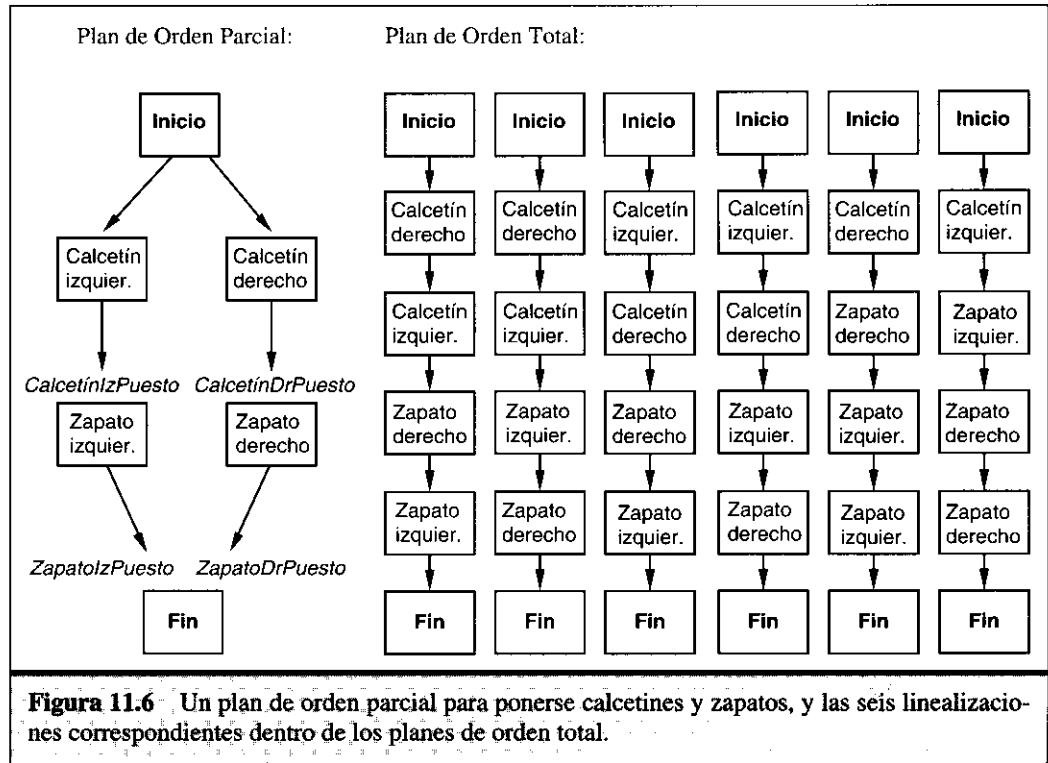
Objetivo (ZapatoDerechoPuesto ∧ ZapatoIzquierdoPuesto)
Inicio()
Acción(ZapatoDerecho,
    PRECOND: CalcetínDerechoPuesto,
    EFECTO: ZapatoDerechoPuesto)
Acción(CalcetínDerecho,
    EFECTO: CalcetínDerechoPuesto)
Acción(ZapatoIzquierdo,
    PRECOND: CalcetínIzquierdoPuesto,
    EFECTO: ZapatoIzquierdoPuesto)
Acción(CalcetínIzquierdo,
    EFECTO: CalcetínIzquierdoPuesto)

```

Un planificador debe ser capaz de trabajar con las secuencias de dos-acciones *CalcetínDerecho* seguido por *ZapatoDerecho* para alcanzar el primer conjunto de objetivos, y la secuencia *CalcetínIzquierdo* seguido de *ZapatoIzquierdo* para el segundo conjunto. Por tanto, las dos secuencias pueden ser combinadas para cumplir el plan total. En este desarrollo, el planificador manipulará las dos secuencias independientemente, sin preocuparse de si una acción pertenece a una secuencia o a otra. Cualquier algoritmo de planificación que pueda conjuntar dos acciones dentro del mismo plan sin necesidad de conocer cuál de ellas es previa a la otra, se conoce como **planificador de primer orden**. La Figura 11.6. nos muestra un plan de orden parcial que constituye la solución para el problema de los zapatos y los calcetines. Destaquemos que la solución es representada como un *grafo* de acciones y no como una secuencia. Destaquemos también que las acciones «dummy», las acciones *Inicio* y *Final*, marcan el principio y el final del plan. Llamarlas acciones simplifica el tratamiento del problema, porque de este modo cada una de las etapas del plan es una acción. La solución de orden-parcial se corresponde con seis posibles planes de orden total; cada uno de ellos se conoce como una **linealización** del plan de orden parcial.

El planificador de orden-parcial puede ser implementado como una búsqueda en el espacio de los planes de orden parcial (desde ahora, los llamaremos únicamente «planes»). Esto es, comenzaremos con un plan vacío. Posteriormente consideraremos formas de refinar este plan hasta que tengamos un plan completo que resuelve el problema. Las acciones en esta búsqueda no son acciones en el mundo, sino acciones sobre planes: añadir una etapa a un plan, imponer una ordenación que coloca una acción antes de otra, etc.

Definiremos el algoritmo POP para procesos de planificación de orden-parcial. Es tradicional escribir el algoritmo POP como un programa autónomo, preferible a la formulación de planes de orden parcial como ejemplos de problemas de búsqueda. Esto nos permitirá centrarnos sobre los refinamientos de las etapas del plan que pueden ser apli-



cados, en lugar de preocuparnos de cómo los algoritmos exploran el espacio. De hecho, una amplia variedad de métodos de búsqueda heurísticos poco fundamentados pueden ser aplicados una vez que el problema de búsqueda es formulado.

Recordemos que los estados de nuestros problemas de búsqueda serán mayoritariamente planes inacabados. Para evitar confusiones con los estados del mundo, nos referiremos a planes más que a estados. Cada plan tiene los siguientes cuatro componentes, donde los dos primeros definen las etapas del plan y los dos últimos sirven como función de contabilidad para determinar cómo los planes pueden ser extendidos:

- Un conjunto de **acciones** que confeccionen las etapas del plan. Éstas son tomadas del conjunto de acciones en el problema de planificación. El plan «vacío» contiene simplemente las acciones *Iniciar* y *Finalizar*. *Iniciar* no posee precondiciones y tiene como efectos los literales en el estado inicial del problema de planificación. *Finalizar* no tiene efectos y tiene como precondiciones los literales del objetivo del problema de planificación.
- Un conjunto de **restricciones ordenadas**. Cada limitación ordenada es de la forma $A < B$, la cual se lee como «*A* antes de *B*» y significa que la acción *A* debe ser ejecutada en algún momento antes de *B*, pero no necesariamente en el estado inmediatamente anterior. Las restricciones ordenadas deben describir un orden parcial apropiado. Cualquier ciclo (del tipo $A < B$ y $B < A$) representa una contradicción, de modo que una limitación de orden no podrá ser añadida a un plan si crea un ciclo.

RELACIONES
CAUSALES
ALCANCES

CONFLICTOS

PRECONDICIONES
ABIERTAS

PLAN CONSISTENTE



- Un conjunto de **relaciones causales**. Un enlace causal entre dos acciones A y B en un plan es escrito como $A \xrightarrow{p} B$ y se lee como « A alcanza B a través de p ». Por ejemplo, el enlace causal

$$\text{CalcetínDerecho} \xrightarrow{\text{CalcetínDerechoPuesto}} \text{ZapatoDerecho}$$

expresa que *CalcetínDerechoPuesto* es un efecto de la acción *CalcetínDerecho* y una precondición para *ZapatoDerecho*. También nos informa de que *CalcetínDerechoPuesto* debe ser cierto durante el tiempo de acción que discurre desde la acción *CalcetínDerecho* a la acción *ZapatoDerecho*. En otras palabras, el plan no podría ser extendido mediante la aportación de una nueva acción C que crease un **conflicto** con el enlace causal. Una acción C entraría en conflicto con $A \xrightarrow{p} B$ si C tiene el efecto $\neg p$, y si C pudiera (de acuerdo con el conjunto de restricciones ordenadas) traer A antes que B . Algunos autores lo llaman **intervalos de protección** de enlaces causales, porque el link $A \xrightarrow{p} B$ protege a p de ser negado a lo largo del intervalo que va desde A hasta B .

- Un conjunto de **precondiciones abiertas**. Una precondición es abierta si no es alcanzada por ninguna acción en un plan. Los planificadores trabajan para reducir el conjunto de precondiciones abiertas al conjunto vacío, sin introducir contradicciones.

Por ejemplo, el plan definitivo de la Figura 11.6 tiene los siguientes componentes:

Acciones: $\{\text{CalcetínDerecho}, \text{ZapatoDerecho}, \text{CalcetínIzquierdo}, \text{ZapatoIzquierdo}, \text{Iniciar}, \text{Finalizar}\}$

Relaciones de orden: $\{\text{CalcetínDerecho} < \text{ZapatoDerecho}, \text{CalcetínIzquierdo} < \text{ZapatoIzquierdo}\}$

Enlaces: $\{\text{CalcetínDerecho} \xrightarrow{\text{CalcetínDerechoPuesto}} \text{ZapatoDerecho}, \text{CalcetínIzquierdo} \xrightarrow{\text{CalcetínIzquierdoPuesto}} \text{ZapatoIzquierdo}, \text{ZapatoDerecho} \xrightarrow{\text{ZapatoDerechoPuesto}} \text{Finalizar}, \text{ZapatoIzquierdo} \xrightarrow{\text{ZapatoIzquierdoPuesto}} \text{Finalizar}\}$

PrecondicionesAbiertas: {}

Definimos un **plan consistente** como un plan en el cual no hay ciclos en las restricciones ordenadas y no existen conflictos con los enlaces causales. Un plan consistente con precondiciones no abiertas es una **solución**. El lector debe estar convencido del siguiente hecho: *cada linealización de una solución de orden-parcial es una solución de orden-total cuya ejecución desde el estado inicial alcanza el estado objetivo*. Esto significa que podremos extender la noción de «ejecución de un plan» de planes de orden total a planes de orden parcial. Un plan de orden parcial es ejecutado por la elección de alguna de las acciones posibles. Veremos en el Capítulo 12 que la flexibilidad disponible para un agente cuando ejecuta un plan puede ser útil si el entorno no coopera. La flexibilidad en la ordenación también hace más sencillo combinar pequeños planes dentro de otros mayores, porque cada uno de los pequeños puede reordenar sus acciones para evitar conflicto con otros planes.

Ahora estamos preparados para formular el problema de búsqueda que POP resuelve. Comenzaremos con una formulación adecuada para problemas de planificación proposicional, dejando para más tarde las complicaciones derivadas de formulaciones de primer orden. Como es habitual, la definición incluye el estado inicial, las acciones y la evaluación del objetivo.

- El plan inicial contiene *Iniciar* y *Finalizar*, la restricción de orden *Iniciar* < *Finalizar*, sin enlaces causales y todas las precondiciones en *Finalizar* como precondiciones abiertas.
- La función sucesora de manera arbitraria selecciona una precondición abierta p sobre una acción B y genera un plan sucesor para cada posible modo de selección consistente de una acción A que alcance p . La consistencia es impuesta como sigue:
 1. El enlace causal $A \xrightarrow{p} B$ y la restricción de orden $A < B$ son añadidos al plan. A puede ser una acción que ya existe en el plan o una nueva. Si es nueva, se añade al plan junto a las condiciones *Iniciar* < A y $A < \text{Finalizar}$.
 2. Resolvemos conflictos entre el nuevo enlace causal y el resto de acciones existentes y entre la acción A (si es nueva) y el resto de enlaces causales existentes. Un conflicto entre $A \xrightarrow{p} B$ y C es resuelto haciendo que C ocurra en algún momento fuera de la protección del intervalo, tanto añadiendo $B < C$ o $C < A$.
- La evaluación del objetivo chequea si un plan es una solución para el problema de planificación original. Como solamente son generados planes consistentes, la evaluación del objetivo simplemente necesita que no existan precondiciones abiertas.

Recordemos que las acciones consideradas por los algoritmos de búsqueda bajo esta formulación son etapas de refinamiento de planes más que acciones reales del propio dominio. El coste del camino es irrelevante, estrictamente hablando, porque lo único que nos preocupa es el coste total de las acciones reales en el plan llevado a cabo. Sin embargo, es posible especificar una función de coste del camino que refleje los costes reales del plan: computamos 1 por cada acción real añadida al plan y 0 para todo el resto de etapas de refinamiento. De este modo, $g(n)$, donde n representa un plan, será igual al número de acciones reales en el plan. Una estimación heurística $h(n)$ puede también ser usada.

A primera vista, uno podría pensar que la función sucesora debería incluir sucesores para cada p abierta, y no simplemente para uno de ellos. Esto podría ser redundante e ineficaz, sin embargo, por la misma razón, los algoritmos de satisfacción de restricciones no incluyen sucesores para cada variable posible: el orden en el que consideremos las precondiciones abiertas (como el orden en el que consideramos variables CSP) es commutativo (véase Apartado 5.2). De este modo, podemos elegir una ordenación arbitraria y aún tener un algoritmo completo. La elección del orden correcto puede llevarnos a una búsqueda más rápida, pero todas las ordenaciones finalizan con el mismo conjunto de soluciones candidatas.

Ejemplo de planificación de orden parcial

Veamos cómo POP resuelve el problema de la rueda de repuesto de la Sección 11.2. La descripción del problema es repetida en la Figura 11.7.

```

Iniciar (En(Deshinchada, Eje)  $\wedge$  En(Repuesto, Maletero))
Objetivo(En(Repuesto, Eje))
Acción (Quitar(Repuesto, Maletero)),
  PRECOND: En(Repuesto, Maletero)
  EFECTO:  $\neg$ En(Repuesto, Maletero)  $\wedge$  En(Repuesto, Suelo))
Acción (Quitar(Deshinchada, Eje)),
  PRECOND: En(Deshinchada, Eje),
  EFECTO:  $\neg$ En(Deshinchada, Eje)  $\wedge$  En(Deshinchada, Suelo))
Acción (Colocar(Repuesto, Eje)),
  PRECOND: En(Repuesto, Suelo)  $\wedge$  \neg En(Deshinchada, Eje),
  EFECTO:  $\neg$ En(Repuesto, Suelo)  $\wedge$  En(Repuesto, Eje))
Acción (DejarloDeNoche),
  PRECOND:
  EFECTO:  $\neg$ En(Repuesto, Suelo)  $\wedge$  \neg En(Repuesto, Eje)  $\wedge$  \neg En(Repuesto, Maletero)
 $\wedge$   $\neg$ En(Deshinchada, Suelo)  $\wedge$  \neg En(Deshinchada, Eje))

```

Figura 11.7 Descripción del problema simplificado de la rueda de repuesto.

La búsqueda de una solución comienza con el plan inicial, que contiene una acción *Iniciar* con el efecto *En(Repuesto, Maletero) \wedge En(Deshinchada, Eje)* y una acción *Finalizar* con la única precondición *En(Repuesto, Eje)*. Entonces, generamos sucesores mediante la elección de una precondición abierta sobre la que trabajar (irrevocablemente) y la elección de diferentes acciones posibles para alcanzarlo. Hasta ahora, no nos preocupamos acerca de una función heurística que nos ayude en la toma de estas decisiones; aparentemente tendremos elecciones arbitrarias. La secuencia de eventos es como sigue:

1. Seleccionar la única precondición abierta de *Finalizar*, esto es, *En(Repuesto, Maletero)*. Elegir la única acción aplicable, *Colocar(Repuesto, Eje)*.
2. Seleccionar *En(Repuesto, Suelo)* precondición de *Colocar(Repuesto, Eje)*. Elegir la única acción aplicable, *Quitar(Repuesto, Maletero)* para lograrlo. El plan resultante se muestra en la Figura 11.8.

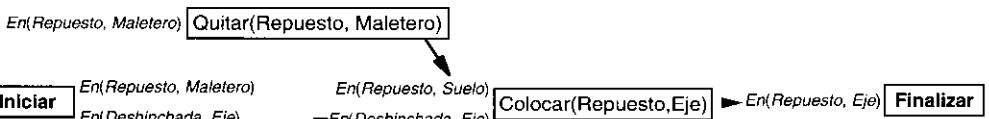
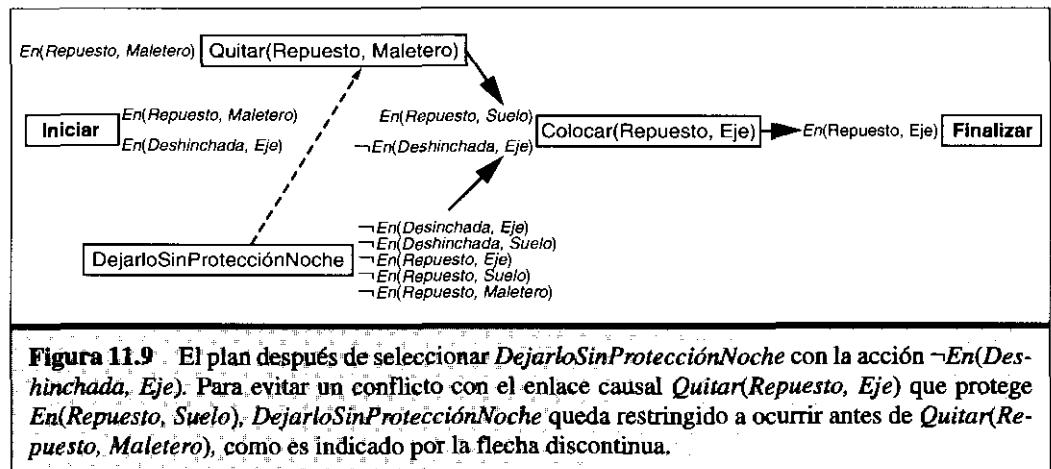


Figura 11.8 Un plan de orden parcial incompleto para el problema de repuesto, después de seleccionar acciones para las primeras dos precondiciones abiertas. Las cajas representan acciones, las precondiciones a la izquierda y los efectos a la derecha. (Los efectos son omitidos, excepto para el de la acción *Iniciar*.) Las flechas oscuras representan enlaces causales protegiendo la proposición que indica la flecha.

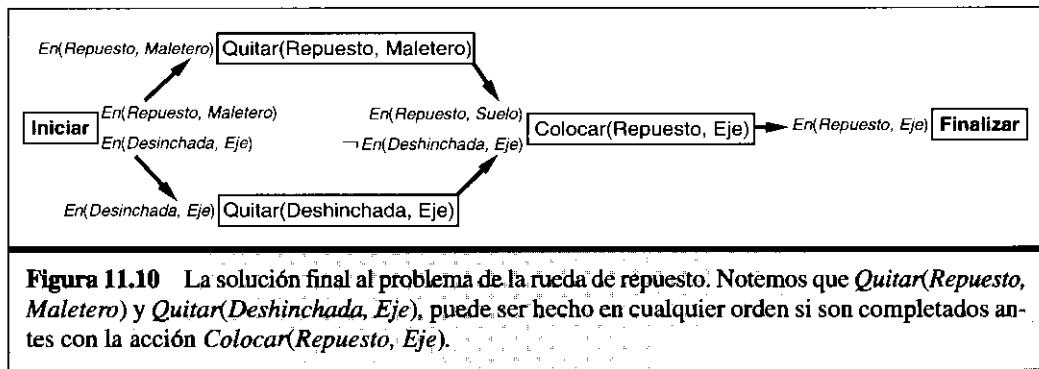
3. Seleccionar $\neg En(Deshinchada, Eje)$, precondición de *Colocar(Repuesto, Eje)*. En sentido opuesto, seleccionar la acción *DejarloSinProtecciónNoche* en lugar de la acción *Quitar(Repuesto, Eje)*. Destacamos que *DejarloSinProtecciónNoche* también tiene el efecto $\neg En(Repuesto, Suelo)$, lo cual significa que entra en conflicto con el enlace causal

$$Quitar(Repuesto, Maletero) \xrightarrow{En(Repuesto, Suelo)} Colocar(Repuesto, Eje)$$

Para resolver el conflicto añadimos una limitación de orden colocando *DejarloSinProtecciónNoche* antes de *Quitar(Repuesto, Maletero)*. El plan resultante es mostrado en la Figura 11.9 (¿Por qué esto resuelve el conflicto y por qué no existe otro modo de resolverlo?)



4. La única precondición abierta que permanece en este momento es *En(Repuesto, Maletero)*. La única acción que puede alcanzarla es la acción *Iniciar*, pero el enlace causal entre *Iniciar* y *Quitar(Repuesto, Maletero)* está en conflicto con $\neg En(Repuesto, Maletero)$, efecto de *DejarloSinProtecciónNoche*: no se puede ordenar antes que *Quitar(Repuesto, Maletero)*, porque ya existe una restricción de orden antes de *Quitar(Repuesto, Maletero)*. De este modo, estamos forzados a volver hacia atrás, eliminar la acción *Quitar(Repuesto, Maletero)* y los dos últimos enlaces causales, regresando al estado de la Figura 11.8. En esencia, el planificador ha demostrado que *DejarloSinProtecciónNoche*, no funciona como una forma de cambiar una rueda.
5. Consideremos de nuevo $\neg En(Deshinchada, Eje)$, precondición de *Colocar(Repuesto, Eje)*. En este momento, seleccionamos *Quitar(Deshinchada, Eje)*.
6. Una vez más, seleccionamos *En(Repuesto, Rueda)* precondición de *Quitar(Repuesto, Maletero)* y seleccionar *Iniciar* para alcanzarlo. Esta vez no existen conflictos.
7. Seleccionar *En(Deshinchada, Eje)* precondición de *Quitar(Deshinchada, Eje)*, y elegir *Iniciar* para alcanzarlo. Esto nos da un plan completo, consistente (en otras palabras, una solución) como mostramos en la Figura 11.10.



Aunque este ejemplo es muy simple, ilustra algunos de los puntos fuertes de la planificación de orden-parcial. En primer lugar, los enlaces causales nos llevan pronto a la poda de porciones del espacio de búsqueda que, por causa de conflictos irresolubles, no contiene soluciones. En segundo lugar, la solución en la Figura 11.10 es un plan de orden-parcial. En este caso la ventaja es pequeña, porque solamente existen dos posibles linealizaciones; sin embargo, un agente podría agradecer la flexibilidad, por ejemplo, si la rueda tiene que ser cambiada en la mitad de una carretera con tráfico denso.

El ejemplo también apunta a algunas posibles mejoras que podrían ser hechas. Por ejemplo, existe duplicación de esfuerzo: *Iniciar* se enlaza con *Quitar(Repuesto, Maletero)* antes que el conflicto cause un retroceso y sea entonces desconectado incluso aunque no esté implicado en el conflicto. Es entonces conectado de nuevo y la búsqueda continúa. Esta situación es típica de retrocesos cronológicos y podría estar atenuada por retrocesos dirigidos a dominios.

Planificación de orden parcial con variables independientes

En esta sección, consideraremos las complicaciones que pueden surgir cuando POP es usado con representación de acciones de primer orden que incluyen variables. Supongamos que tenemos un problema en un mundo de bloques (Figura 11.4) con las precondiciones abiertas *Sobre(A, B)* y la acción:

Acción (Mover (b, x, y),
PRECOND: Sobre(b, x) \wedge Despejar(b) \wedge Despejar(y),
EFFECTO: Sobre(b, y) \wedge Despejar(x) \wedge \neg Sobre(b, x) \wedge \neg Despejar(y))

Esta acción alcanza *Sobre(A, B)* porque el efecto *Sobre(b, y)* unifica *Sobre(A, B)* con la sustitución $\{b/A, y/B\}$. Aplicando esta sustitución en la acción, nos queda:

Acción (Mover (A, x, B),
PRECOND: Sobre(A, x) \wedge Despejar(A) \wedge Despejar(B),
EFFECTO: Sobre(A, B) \wedge Despejar(x) \wedge \neg Sobre(A, x) \wedge \neg Despejar(B))

Esto nos deja la variable *x* sin explicitar. Es decir, la acción nos habla de mover un bloque *A* desde algún lugar, sin decir cuál. Este es otro ejemplo del último principio de com-

promiso: podemos retrasar la toma de decisión hasta que alguna otra etapa en el plan lo haga por nosotros. Por ejemplo, supongamos que tenemos *Sobre(A, D)* en el estado inicial. Entonces la acción *Iniciar* puede ser usada para alcanzar *Sobre(A, x)*, vinculando *x* a *D*. Esta estrategia que espera más información antes de fijar *x* es normalmente más eficiente que intentarlo para cualquier valor posible de *x* y hacer regresiones para cada una de las que fracase.

La presencia de variables en las precondiciones y en las acciones complica los procesos en la detección y resolución de conflictos. Por ejemplo, cuando *Mover(A, x, B)* es añadido al plan, necesitaremos una relación causal

$$\text{Mover}(A, x, B) \xrightarrow{\text{Sobre}(A, B)} \text{Finalizar}$$

RESTRICCIONES DE DESIGUALDAD

Si existe otra acción *M₂* con efecto $\neg\text{Sobre}(A, z)$, entonces *M₂* entra en conflicto sólo si *z* es *B*. Para acomodar esta posibilidad, extendemos la representación de planes con el fin de incluir un conjunto de **restricciones de desigualdad** de la forma $z \neq X$ donde *z* es una variable y *X* otra variable o una constante. En este caso, podríamos resolver el conflicto añadiendo $z \neq B$, lo que significa que las extensiones futuras al plan pueden instanciar *z* a cualquier valor excepto a *B*. En cualquier momento que apliquemos una sustitución al plan, debemos asegurarnos que las desigualdades no contradicen la sustitución. Por ejemplo, una sustitución que incluya *x/y* entra en conflicto con la inecuación *x ≠ y*. Este tipo de conflictos no pueden ser resueltos, por tanto el planificador debe ejecutar regresiones en su desarrollo.

Un ejemplo más exhaustivo de la planificación POP con variables en el mundo de los bloques viene dado en la Sección 12.6.

Heurísticas para planificación de orden parcial

Comparado con la planificación de orden total, la planificación de orden parcial posee una clara ventaja por su capacidad de descomponer problemas en subproblemas. Tiene también la desventaja de que no representa los estados directamente, de modo que es más difícil estimar cuánto de alejado está un plan de orden parcial de alcanzar un objetivo.

La heurística más obvia es el recuento del número de precondiciones abiertas distintas. Esto puede ser mejorado mediante la eliminación del número de precondiciones abiertas que se ajustan a los literales en el estado *Iniciar*. En un caso de orden total, esto sobrestima el coste cuando existen acciones que alcanzan múltiples objetivos y subestima el coste si existen interacciones negativas entre etapas del plan. La siguiente sección presenta un enfoque que nos permite obtener heurísticas más adecuadas para un problema aproximado.

La función heurística es usada para seleccionar qué plan refinar. Dada esta elección, el algoritmo genera sucesores basados en la selección de una única precondición abierta para trabajar sobre ella. En el caso de la selección variable en algoritmos de satisfacción de restricciones, esta selección tiene un impacto importante sobre la eficiencia. La **heurística más restrictiva** de CSPs puede ser adaptada para algoritmos de planificación y funcionar adecuadamente. La idea es seleccionar la condición abierta que pueda ser satisfecha por el *menor número* de caminos. Existen dos casos especiales para

esta heurística: primero, si una condición abierta no puede ser alcanzada por ninguna acción, la heurística la seleccionará. Esta es una buena estrategia porque la temprana detección de imposibilidad puede ahorrar un gran trabajo perdido. Segundo, si una condición abierta puede ser alcanzada de un único modo, debe ser seleccionada porque la decisión es inevitable y podría proporcionar restricciones adicionales sobre otras opciones aún sin ser hechas. Aunque la computación completa del número de formas de satisfacer cada condición abierta es costosa y no siempre merece la pena, los experimentos demuestran que trabajar con estos dos casos especiales proporciona una agilización sustanciosa.

11.4 Grafos de planificación

GRAFO DE PLANIFICACIÓN

NIVELES

Todas las heurísticas que hemos sugerido para planificación de orden parcial y total pueden sufrir imprecisiones. Esta sección muestra cómo una estructura especial llamada **grafo de planificación** puede ser usada para dar mejores estimaciones heurísticas. Estas heurísticas pueden ser aplicadas a cualquiera de las técnicas de búsqueda que hemos visto hasta ahora. Alternativamente, podemos extraer una solución directamente del grafo de planificación, usando un algoritmo especializado llamado GRAPHPLAN.

Un grafo de planificación consiste en una secuencia de **niveles** que corresponden a escalones de tiempo en el plan, y donde el nivel 0 es el estado inicial. Cada nivel contiene un conjunto de literales y un conjunto de acciones. A grandes rasgos, los literales son todos aquellos que *pueden* ser ciertos en cualquiera de las etapas, dependiendo solamente de las acciones ejecutadas en las etapas previas. También a grandes rasgos, las acciones son todas aquellas que *pueden* tener todas sus precondiciones satisfechas en cualquiera de las etapas, dependiendo de cuáles sean los literales sobre los que realmente actúan. Decimos a «grandes rasgos», porque el grafo de planificación solamente registra un restringido subconjunto de posibles interacciones negativas entre sus acciones; por tanto, se puede ser optimista acerca del mínimo número de etapas temporales que se requieren para que un literal sea correcto. Sin embargo, el número de etapas en el grafo de planificación proporciona una buena estimación sobre la dificultad de alcanzar desde el estado inicial un literal dado. De manera más importante, el grafo de planificación es definido de tal manera que puede ser construido muy eficientemente.

Los grafos de planificación funcionan solamente en problemas de planificación proposicional, (aquellos sin variables). Como mencionamos en la Sección 11.1, tanto las representaciones STRIPS como ADL pueden ser proposicionales. Para problemas con gran cantidad de objetos, esto podría convertirse en un desbordamiento del número de esquemas de acción. Dejando esto a un lado, los grafos de planificación han demostrado ser efectivas herramientas para solucionar problemas de planificación complejos.

Ilustraremos el tema de los grafos de planificación con el siguiente ejemplo (ejemplos más complejos llevarían asociados gráficos que no cabrían en la página). La Figura 11.11 nos muestra un problema, y la Figura 11.12 nos muestra su grafo de planificación.

Iniciar (Tener(Pastel))
Objetivo(Tener(Pastel) \wedge Comido(Pastel))
Acción (Comer(Pastel))
 PRECOND: *Tener(Pastel)*
 EFECTO: $\neg Tener(Pastel) \rightarrow Comido(Pastel)$
Acción (Cocinar(Pastel))
 PRECOND: $\neg Tener(Pastel)$
 EFECTO: *Tener(Pastel)*

Figura 11.11 El problema de «tener y comer pastel».

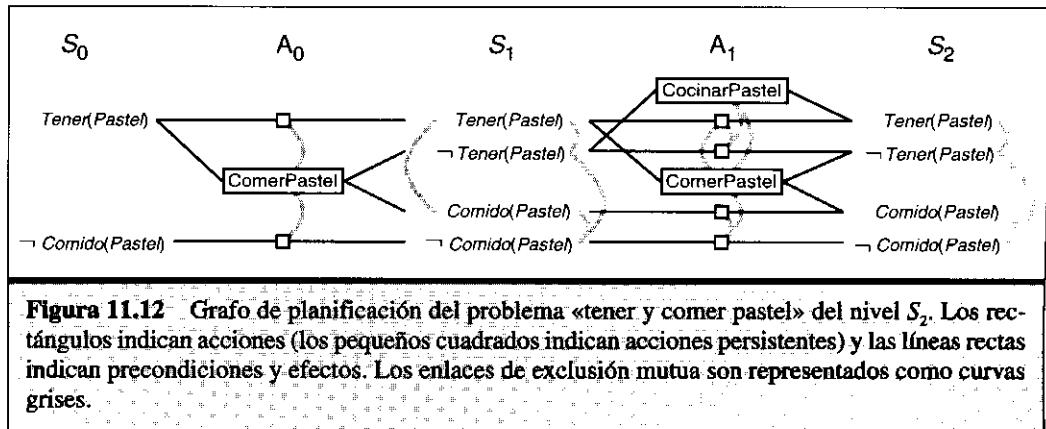


Figura 11.12 Grafo de planificación del problema «tener y comer pastel» del nivel S_2 . Los rectángulos indican acciones (los pequeños cuadrados indican acciones persistentes) y las líneas rectas indican precondiciones y efectos. Los enlaces de exclusión mutua son representados como curvas grises.

Comenzaremos con el nivel S_0 , que representa el estado inicial del problema. Continuamos con la acción A_0 , en la que consideramos todas las acciones cuyas precondiciones se satisfagan en el nivel previo. Cada acción está conectada con sus precondiciones en S_0 y sus efectos en S_1 , en este caso introduciendo dos nuevos literales en S_1 que no estaban en S_0 .

El grafo de planificación necesita un modo de representar tanto la falta de acción como la acción. Esto es, necesita el equivalente a los axiomas del marco en el cálculo situado que permitan que un literal permanezca siendo verdad desde una situación a la siguiente si no existe acción que lo altere. En un grafo de planificación esto se logra con un conjunto de **acciones persistentes**. Para cada literal positivo y negativo C , añadimos al problema una acción persistente con la precondición C y el efecto C . La Figura 11.12 muestra una acción «real» *Comer(Pastel)* en A_0 , junto a dos acciones persistentes dibujadas como pequeñas cajas cuadradas.

El nivel A_0 contiene todas las acciones que podrían ocurrir en el estado S_0 , pero sólo registra de manera importante los conflictos entre las acciones que podrían prevenirles de suceder conjuntamente. Las líneas grises en la Figura 11.12 indican enlaces de **exclusión mutua**. Por ejemplo, *Comer(Pastel)* se excluye mutuamente con la persistencia de *Tener(Pastel)* o de $\neg Comer(Pastel)$. Veamos cómo los enlaces de exclusión mutua son computados.

ACCIONES PERSISTENTES

EXCLUSIÓN MUTUA

El nivel S_i contiene todos los literales que podrían resultar de escoger cualquier subconjunto de acciones en A_0 . También contiene enlaces de exclusión mutua (líneas grises) que informan de los literales que podrían no aparecer juntos, pese a la elección de las acciones. Por ejemplo, *Tener(Pastel)* y *Comer(Pastel)* son mutuamente independientes; esto es, según la selección de acciones en A_0 , una u otra podría ser solución, pero no ambas. En otras palabras, S_i representa estados múltiples, al igual que una búsqueda de regresión en el espacio de estados, y los enlaces mutuamente excluyentes son restricciones que definen el conjunto de posibles estados.

Continuaremos de este modo, alternando entre el estado de nivel S_i y el nivel de acción A_i , hasta que alcancemos un nivel donde dos niveles consecutivos sean idénticos. En esta situación, decimos que el grafo está **estabilizado**. Cada nivel posterior será idéntico, hasta que la expansión sea innecesaria.

Lo que obtenemos con esto es una estructura donde cada nivel A_i contiene todas las acciones que son aplicables en S_i , junto con restricciones que nos indican qué parejas de acciones no pueden ser ejecutadas juntas. Cada nivel S_i contiene todos los literales que podrían obtenerse de cualquier posible elección de acciones en A_{i-1} , junto con las restricciones que especifican las parejas de literales que no son posibles. Es importante hacer notar que el proceso de diseño de un grafo de planificación no requiere elección entre acciones, lo que podría llevarnos a una búsqueda combinatoria. En lugar de eso, se registra la imposibilidad de ciertas selecciones usando enlaces mutuamente excluyentes. La complejidad de construir grafos de planificación es de orden polinomial bajo, dependiente del número de acciones y literales, mientras que el espacio de estados tiene orden exponencial en términos del número de literales.

Definiremos ahora enlaces mutuamente excluyentes entre acciones y literales. Una relación mutuamente excluyente ocurre entre dos *acciones* en un nivel dado si cualquiera de las tres condiciones siguientes se cumplen:

- *Efectos inconsistentes*: una acción niega el efecto de la otra. Por ejemplo, *Comer(Pastel)* y la persistencia de *Tener(Pastel)* tienen efectos inconsistentes porque no coinciden sus efectos.
- *Interferencia*: uno de los efectos de una de las acciones es la negación de una precondición de la otra. Por ejemplo, *Comer(Pastel)* interfiere con la persistencia de *Tener(Pastel)* mediante la negación de sus precondiciones.
- *Necesidades que entran en competencia*: una de las precondiciones de una acción es mutuamente excluyente con una precondición de la otra. Por ejemplo, *Cocinar(Pastel)* y *Comer(Pastel)* son mutuamente excluyentes porque ambas compiten sobre el valor de la precondición de *Tener(Pastel)*.

Una relación de exclusión mutua sucede entre dos *literales* en el mismo nivel, si uno es la negación del otro o si cada posible par de acciones que podrían alcanzar los literales son mutuamente excluyentes. A esta condición se le llama *soporte inconsistente*. Por ejemplo, *Tener(Pastel)* y *Comer(Pastel)* son mutuamente excluyentes en S_1 porque la única forma de alcanzar *Tener(Pastel)*, la acción persistente, es mutuamente excluyente con la única forma de alcanzar *Comido(Pastel)*. En S_2 , los dos literales no son mutuamente excluyentes porque hay otros modos de alcanzarlos, tal como *Cocinar(Pastel)* y la persistencia de *Comido(Pastel)*, que no son mutuamente excluyentes.

Grafos de planificación para estimación de heurísticas



COSTE DE NIVEL

GRAFO DE PLANIFICACIÓN SERIAL

NIVEL MÁXIMO

NIVEL SUMA

NIVEL DE CONJUNTO

Un grafo de planificación, una vez construido, es una fuente rica en información acerca de un problema. Por ejemplo, *un literal que no aparece en el nivel final del grafo no puede ser alcanzado por ningún plan*. Esta observación puede usarse en búsquedas hacia atrás como se muestra a continuación: cualquier estado contenido un literal inalcanzable tiene un coste $h(n) = \infty$. De manera similar, en planificaciones de orden parcial, cualquier plan con una condición abierta inalcanzable tiene un $h(n) = \infty$.

Esta idea puede plantearse de un modo más general. Podemos estimar el coste que supone alcanzar cualquier literal del objetivo como el nivel en el cual aparece primariamente el grafo de planificación. Lo llamaremos **coste de nivel** del objetivo. En la Figura 11.12, *Tener(Pastel)* tiene coste de nivel 0 y *Comido(Pastel)* tiene coste de nivel 1. Es sencillo mostrar (Ejercicio 11.9) que estas estimaciones son admisibles para objetivos individuales. La estimación puede no ser muy buena, sin embargo, los grafos de planificación permiten varias acciones en cada nivel mientras la heurística compute simplemente niveles y no el número de las acciones. Por esta razón, es común usar un **grafo de planificación serial** para las heurísticas. Un grafo serial exige que sólo una acción pueda ocurrir en una etapa de tiempo dada; esto se logra añadiendo enlaces de exclusión mutua entre cada par de acciones excepto acciones persistentes. Los costes de nivel extraídos de grafos seriales son frecuentemente estimaciones razonables de costes reales.

Para estimar el coste de una secuencia de objetivos, existen tres enfoques simples. La heurística de **nivel máximo** simplemente toma el coste del máximo nivel de cualquiera de los objetivos; esto es admisible pero no necesariamente preciso. La heurística de **nivel suma** (asumiendo la hipótesis de independencia de objetivos) devuelve la suma de los costes de los niveles objetivo. Esto no es admisible pero funciona bien en la práctica en problemas que son en buena parte descomponibles. Es más preciso que la aplicación de heurísticas de número de objetivos insatisfechos presentada en la Sección 11.2. En nuestro problema, la estimación heurística de la secuencia de objetivos *Tener(Pastel) ∧ Comido(Pastel)* será $0 + 1 = 1$, mientras que la respuesta correcta es 2. Sin embargo, si eliminamos la acción *Cocinar(Pastel)*, la estimación podría ser 1, pero la secuencia de objetivos sería imposible. Finalmente, la heurística de **nivel de conjunto** encuentra el nivel en el que todos los literales de la secuencia de objetivos aparecen en el grafo de planificación, sin que ningún par de ellos sean mutuamente excluyentes. Esta heurística nos da el valor correcto, 2, para nuestro problema original sin *Cocinar(Pastel)*. Es mejor que la heurística de nivel máximo y además trabaja bien en tareas en las que hay buenas relaciones de interacción entre subplanes.

El grafo de planificación, como herramienta de generación de heurísticas adecuadas que es, nos permite entenderlo también como un problema aproximado que es eficientemente resoluble. Para entender la naturaleza del problema aproximado, necesitamos entender exactamente qué significa que un literal g aparezca en un nivel S_i en un grafo de planificación. Idealmente, queríramos que fuese una garantía de la existencia de un plan la aparición de una acción de nivel i que alcance g , y de igual modo que si g no apareciese no existe tal plan. Desafortunadamente, obtener esa garantía es tan difícil como resolver el problema de planificación original. De manera que el grafo de planificación

simple cumple la segunda condición de la garantía (si g no aparece, no hay plan), pero en el caso de que g aparezca, entonces todo lo que el grafo de planificación asegura es que existe un plan que posiblemente alcance g y sin defectos «obvios». Un defecto obvio se define como un error que puede ser detectado considerando dos acciones o dos literales en un instante de tiempo (en otras palabras, observando las relaciones de exclusión mutua). Podrían existir más errores no obvios implicando a tres, cuatro o más acciones, pero la experiencia ha mostrado que no merece la pena el esfuerzo computacional que supone seguir el rastro de estos posibles errores. Es un resultado similar al que se mostró al estudiar los problemas de satisfacción de restricciones, donde era preferible computar 2-consistencias antes de la búsqueda de soluciones, pero no tan preferible como el hecho de computar 3-consistencias o más (véase Sección 5.2).

El algoritmo GRAPHPLAN

Esta subsección muestra cómo extraer un plan directamente de un grafo de planificación, preferiblemente al uso del plan como forma de obtener una heurística. El algoritmo GRAPHPLAN (Figura 11.13) tiene dos etapas fundamentales, las cuales se alternan dentro del ciclo del algoritmo. Primeramente, se chequea si todos los literales del objetivo están presentes en el nivel actual sin que existan enlaces mutuamente excluyentes entre cualquier par de ellos. Si éste es el caso, entonces una solución *podría* existir dentro del grafo actual, de modo que el algoritmo intentase extraer esa solución. Además, se extiende el grafo añadiendo acciones para el nivel actual y literales de estado para el siguiente nivel. El proceso continúa hasta que una solución sea encontrada o se compruebe que la solución no existe.

```

función GRAPHPLAN (problema) devuelve solución o error
    grafo  $\leftarrow$  GRAFO-PLANIFICACIÓN-INICIAL (problema)
    objetivos  $\leftarrow$  OBJETIVOS [problema]
bucle
    si objetivos todos los enlaces mutuamente excluyentes están en el último nivel del grafo
    entonces
        solución  $\leftarrow$  EXTRAER SOLUCIÓN (grafo, objetivos, LONGITUD(grafo))
        si solución  $\neq$  error entonces devuelve solución
        en caso contrario si No SOLUCIÓN POSIBLE (grafo) entonces devuelve error
    grafo  $\leftarrow$  GRAFO-EXPAND (grafo, problema)

```

Figura 11.13 El algoritmo GRAPHPLAN alterna entre una etapa de extracción de solución y una etapa de expansión de grafo. EXTRAER-SOLUCIÓN busca si un plan puede ser encontrado, comenzando desde el final y hacia atrás. EXPANDIR-GRAFO añade las acciones para el estado actual y los literales de estado para el nivel siguiente.

Representemos las operaciones de GRAPHPLAN en el problema de la rueda de repuesto de la Sección 11.1. El grafo completo es mostrado en la Figura 11.14. La primera línea del GRAPHPLAN inicia el grafo de planificación en un grafo de un único nivel (S_0) que

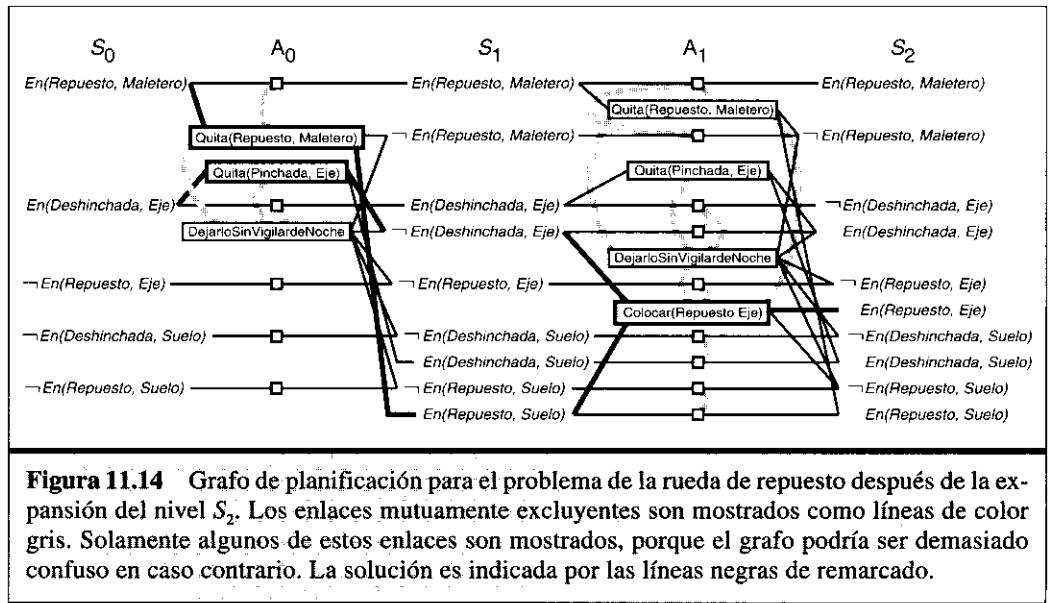


Figura 11.14 Grafo de planificación para el problema de la rueda de repuesto después de la expansión del nivel S_2 . Los enlaces mutuamente excluyentes son mostrados como líneas de color gris. Solamente algunos de estos enlaces son mostrados, porque el grafo podría ser demasiado confuso en caso contrario. La solución es indicada por las líneas negras de remarcado.

consiste en cinco literales del estado inicial. El literal objetivo $En(Repuesto, Eje)$ no está presente en S_0 , de modo que no necesitamos llamar a EXTRAER-SOLUCIÓN pues estamos seguros de que no hay una solución. Por ejemplo, EXPANDIR-GRAFO añade tres acciones cuyas precondiciones existen en el nivel S_0 (esto es, todas las acciones excepto *Colocar(Repuesto, Eje)*), junto con las acciones persistentes para todos los literales en S_0 . Los efectos de las acciones son añadidos en el nivel S_1 . EXPANDIR-GRAFO busca las relaciones mutuamente excluyentes y las añade al grafo.

$En(Repuesto, Eje)$ aún no está presente en S_1 , así que de nuevo no llamamos a EXTRAER-SOLUCIÓN. La llamada a EXPANDIR-SOLUCIÓN nos da el grafo de planificación mostrado en la Figura 11.14. Ahora que tenemos el conjunto completo de acciones, vale la pena pasar a ver algunos ejemplos de relaciones de exclusión mutua y sus causas:

- **Efectos inconsistentes:** *Quitar(Repuesto, Maletero)* es una exclusión mutua con *DejarloSinProtecciónNoche* porque una tiene el efecto $En(Repuesto, Eje)$ y la otra su negación.
- **Interferencia:** *Quitar(Repuesto, Maletero)* es una exclusión mutua con *DejarloSinProtecciónNoche* porque uno tiene la precondición $En(Pinchada, Eje)$ y la otra tiene su negación como un efecto.
- **Necesidades en competencia:** *Colocar(Repuesto, Eje)* es una exclusión mutua con *Quitar(Pinchada, Eje)* porque uno tiene $En(Pinchada, Eje)$ como una precondición y el otro tiene su negación.
- **Soporte inconsistente:** $En(Repuesto, Eje)$ es excluyente mutuamente con $En(Pinchada, Eje)$ en S_2 porque el único modo de alcanzar $En(Repuesto, Eje)$ es mediante *Colocar(Repuesto, Eje)*, y esto es excluyente mutuamente con la acción persistente que es el único modo de alcanzar $En(Pinchada, Eje)$. De este modo, las relaciones mutuamente excluyentes detectan el conflicto inmediato que aparece al intentar colocar dos objetos en el mismo lugar y en el mismo instante de tiempo.

En el momento en el que regresamos al inicio del ciclo del algoritmo, todos los literales del objetivo están presentes en S_2 , y ninguno de ellos es excluido mutuamente por otro. Esto significa que una solución podría existir, y EXTRAER-SOLUCIÓN intentaría encontrarla. En esencia, EXTRAER-SOLUCIÓN resuelve un CSP booleano cuyas variables son las acciones de cada nivel, y los valores de cada variable están dentro o fuera del plan. Podemos usar algoritmos CSP estándares para esto, o podemos definir EXTRAER-SOLUCIÓN como un problema de búsqueda, donde cada estado en la búsqueda contiene un puntero a un nivel en el grafo de planificación y un conjunto de objetivos no satisfechos. Definimos este problema de búsqueda como sigue:

- El estado inicial es el último nivel del grafo de planificación; S_n , junto con el conjunto de objetivos del problema de planificación.
- Las acciones disponibles en un estado de nivel S_i están para seleccionar cualquier subconjunto de acciones libre de conflicto en A_{i-1} cuyos efectos cubren los objetivos en el estado. El estado resultante tiene nivel S_{i-1} y tiene como conjunto de objetivos las precondiciones para el conjunto seleccionado de acciones. Por «libre de conflicto» queremos decir conjunto de acciones tales que ninguna de ellas sean mutuamente excluyentes, y que ninguna de sus precondiciones lo sean tampoco.
- El objetivo es alcanzar un estado a nivel S_0 tal que todos los objetivos sean satisfechos.
- El coste de cada acción sea 1.

Para este problema particular, comenzamos en S_2 con el objetivo *En(Repuesto, Eje)*. La única elección posible que tenemos para alcanzar el conjunto objetivo es *Colocar(Repuesto, Eje)*. Esto nos lleva a una búsqueda de estado en S_1 con objetivos *En(Repuesto, Suelo)* y no $\neg En(Pinchado, Eje)$. El primero puede ser alcanzado sólo mediante *Quitar(Repuesto, Maletero)*, y el último o por *Quitar(Pinchado, Eje)* o *DejarloSinProtecciónNoche*. Como *DejarloSinProtecciónNoche* es mutuamente excluyente con *Quitar(Repuesto, Maletero)*, la única solución es elegir *Quitar(Repuesto, Maletero)* y *Quitar(Pinchado, Eje)*. Esto nos lleva a una búsqueda de estado en S_0 con los objetivos *En(Repuesto, Maletero)* y *En(Pinchado, Eje)*. Ambos están presentes en el estado, de modo que tenemos una solución: las acciones *Quitar(Repuesto, Maletero)* y *Quitar(Pinchado, Eje)* en el nivel A_0 seguido por *Colocar(Repuesto, Eje)* en A_1 .

Conocemos que la planificación es PSPACE-completa y que el desarrollo del grafo de planificación lleva asociado un orden en tiempo polinomial, de modo que puede darse la situación en que la extracción de la solución sea intratable en el peor de los casos. Por tanto, necesitaremos alguna orientación heurística para la selección entre acciones durante la búsqueda hacia atrás. Un enfoque que trabaja bien en la práctica es un algoritmo basado en el coste de nivel de los literales. Para cualquier conjunto de objetivos, procedemos en el siguiente orden:

1. Seleccionar primero el literal con el coste de nivel más alto.
2. Para alcanzar este literal, elegir la acción con la precondición más sencilla. Esto es, elegir una acción tal que la suma (o el máximo) del nivel de coste de sus precondiciones sea la más pequeña.

Interrupción de GRAPHPLAN

Hasta ahora, hemos tratado por encima la cuestión de la interrupción. Si un problema no tiene solución ¿podemos asegurar que GRAPHPLAN no caerá en un bucle, extendiendo el grafo de planificación en cada iteración? La respuesta es afirmativa, pero la demostración completa está más allá de las pretensiones de este libro. Aquí, simplemente indicamos las ideas principales, particularmente aquellas que arrojen luz sobre los grafos de planificación en general.

El primer paso es destacar que ciertas propiedades de los grafos de planificación están creciendo o decreciendo monótonamente. Que « X crezca monótonamente» significa que el conjunto de X s en el nivel $i + 1$, es un superconjunto (no necesariamente propio) del conjunto en el nivel i . Las propiedades son las siguientes:

- *Los literales crecen monótonamente*: una vez que un literal aparece en un nivel dado, aparecerá en todos los niveles siguientes. Esto está motivado por las acciones persistentes; una vez que un literal se pone de manifiesto, las acciones de persistencia ocasionan que permanezca indefinidamente.
- *Las acciones crecen monótonamente*: una vez que una acción aparece en un nivel dado, aparecerá en todos los niveles siguientes. Esto es consecuencia del aumento de literales; si las precondiciones de una acción aparecen en un nivel, entonces aparecerán en los niveles siguientes.
- *Los enlaces de exclusión mutua decrecen monótonamente*: si dos acciones son mutuamente excluyentes en un nivel dado A_i , entonces también lo son para todos los niveles previos en los cuales ambas aparezcan. Lo mismo sucede para exclusiones mutuas entre literales. Puede que no aparezca esta opción en las figuras porque las figuras son una simplificación: no presentan ni los literales que no pueden cumplirse en el nivel S_i , ni las acciones que no pueden ser ejecutadas en el nivel A_i . Podemos ver que «los enlaces mutuamente excluyentes decrecen monótonamente» es cierto si consideramos que estos literales no son visibles y las acciones son excluyentes mutuamente con todos lo demás.

La demostración es algo compleja, pero puede ser presentada por casos: si las acciones A y B son excluyentes mutuamente en el nivel A_i , debe ser por causa de uno de los tres tipos de exclusión mutua. Los dos primeros, efectos inconsistentes e interferencia, son propiedades de las acciones, de modo que si las acciones son mutuamente excluyentes en A_i , lo serán en cualquier nivel. El tercer caso, necesidades en competencia, depende de las condiciones de nivel S_i : ese nivel debe contener una precondición de A que sea mutuamente excluyente con una precondición de B . Ahora, esas dos precondiciones pueden ser mutuamente excluyentes si son negaciones una de otra (en cuyo caso, serían mutuamente excluyentes en cada nivel) o si todas las acciones para alcanzar una son mutuamente excluyentes con todas las acciones para alcanzar la otra. Pero ya conocemos que las acciones disponibles son crecientes monótonamente, así que por inducción, las relaciones excluyentes mutuamente deben estar decreciendo.

Como las acciones y los literales crecen y las relaciones mutuamente excluyentes decrecen, y como sólo existe un número finito de acciones y literales, cada grafo de pla-

nificación se estabiliza (todos los niveles siguientes serán idénticos). Una vez que un grafo se ha estabilizado, si le falta uno de los objetivos del problema, o si dos de los objetivos son mutuamente excluyentes, entonces el problema nunca podrá ser resuelto, y podemos detener el algoritmo GRAPHPLAN y determinar el error. Si el grafo se estabiliza con todos los objetivos y sin relaciones mutuamente excluyentes, pero EXTRAEERSOLUCIÓN fracasa al encontrar una solución, entonces tendremos que extender el grafo de nuevo un número finito de veces, aunque finalmente podremos parar. Este aspecto de interrupción es más complejo y no se mostrará aquí.

11.5 Planificación con lógica proposicional

Vimos en el Capítulo 10 que la planificación puede ser hecha mediante la demostración de un teorema del cálculo situado. Este teorema dice que, dado el estado inicial y dados los axiomas de estados sucesivos que describen los efectos de las acciones, el objetivo será correcto en una situación resultado de una cierta secuencia de acciones. A principios de 1969, se pensaba que este enfoque era demasiado ineficiente para encontrar planes interesantes. Recientes desarrollos en algoritmos de razonamiento eficiente para lógica proposicional (véase Capítulo 7) han generado renovado interés en la interpretación de la planificación como razonamiento lógico.

El enfoque que tomamos en esta sección está basado en la evaluación de la **satisfactoriedad** de una secuencia lógica preferiblemente a la demostración de un teorema. Buscaremos modelos de sentencias proposicionales que sigan la estructura:

$$\text{Estado inicial} \wedge \text{todas las posibles descripciones de acción} \wedge \text{objetivo}$$

La sentencia contendrá símbolos proposicionales correspondiéndose con cada acción posible; un modelo que satisfaga la sentencia le asignará *verdad* a las acciones que son parte de un plan correcto y *falso* a las otras. Una asignación que se corresponda a un plan incorrecto no será un modelo, porque será inconsistente con la afirmación de que el objetivo es correcto. Si el problema de planificación es irresoluble, la sentencia no será satisfecha.

Descripción de problemas de planificación en lógica proposicional

El proceso que seguiremos para traducir problemas STRIPS a una formulación en lógica proposicional es un ejemplo clásico del ciclo de representación de conocimiento: comenzaremos con lo que parece un conjunto razonable de axiomas, veremos que estos axiomas tienen en cuenta modelos espurios no buscados, y escribiremos más axiomas.

Comencemos con un problema muy simple de transporte aéreo. En el estado inicial (tiempo 0), el avión P_1 está en *SFO* y el avión P_2 en *JFK*. El objetivo es tener P_1 en *JFK* y P_2 en *SFO*; esto es, los aviones deben cambiar de aeropuerto. En primer lu-

gar, necesitamos distinguir símbolos proposicionales para afirmaciones en cada instante de tiempo. Usaremos superíndices para denotar el instante, más bien el escalón temporal, como hicimos en el Capítulo 7. De este modo, el estado inicial se expresará:

$$\text{En}(P_1, \text{SFO})^0 \wedge \text{En}(P_2, \text{JFK})^0$$

Recordemos que $\text{En}(P_1, \text{SFO})^0$ es un símbolo atómico. Como la lógica proposicional no tiene una hipótesis de mundo cerrado, podemos especificar las proposiciones que *no* son correctas en el estado inicial. Si algunas proposiciones son desconocidas en el estado inicial, entonces pueden ser dejadas sin satisfacer (**hipótesis de mundo abierto**). En este ejemplo especificamos:

$$\neg \text{En}(P_1, \text{JFK})^0 \wedge \neg \text{En}(P_2, \text{SFO})^0$$

El objetivo debe ser asociado con un instante de tiempo particular. Dado que no conocemos a priori cuántas etapas se necesitan para alcanzar el objetivo, podemos intentar afirmar que el objetivo es correcto en el instante inicial $T = 0$. Esto es, afirmamos $\text{En}(P_1, \text{JFK})^0 \wedge \text{En}(P_2, \text{SFO})^0$. Si esto fracasa, lo intentaremos de nuevo en $T = 1$, y así sucesivamente hasta que alcancemos el plan más corto posible. Para cada valor de T , la base de conocimiento incluirá sólo sentencias cubriendo el intervalo temporal desde cero hasta T . Para asegurar que se interrumpe, un límite superior arbitrario, T_{\max} , es impuesto. Este algoritmo se muestra en la Figura 11.15. Un enfoque alternativo que evita tentativas de múltiples soluciones es discutido en el Ejercicio 11.17.

```

función SATPLAN (problema,  $T_{\max}$ ) devuelve solución o error
    inputs : problema, problema de planificación
             $T_{\max}$ , un límite superior para la longitud del plan
    para  $T = 0$  hasta  $T_{\max}$  hacer
        cnf, correspondencia  $\leftarrow$  TRADUCIR A SAT (problema,  $T$ )
        asignación  $\leftarrow$  RESOLVEDOR SAT (cnf)
        si asignación no es nula entonces
            devuelve EXTRAER-SOLUCIÓN(asignación, correspondencia)
        devuelve error
```

Figura 11.15 El algoritmo SATPLAN. El problema de planificación es traducido a sentencias CNF en las que el objetivo se pretende alcanzar en un tiempo fijo T y son incluidos axiomas para cada estado previo a T . (Los detalles de la traducción son ofrecidos en el texto.) Si el algoritmo de satisfactoriedad encuentra un modelo, entonces es extraído un plan a partir de símbolos proposicionales que refieran acciones y que sean *verdaderos* en el modelo. Si el modelo no existe, el proceso es repetido con el objetivo desplazado a una posición posterior.

La siguiente tarea es la de codificar descripciones de acción en lógica proposicional. El enfoque más sencillo es tener un símbolo proposicional para cada acción; por ejemplo, $\text{Volar}(P_1, \text{SFO}, \text{JFK})^0$ es verdad si el avión P_1 vuela de *SFO* a *JFK* en el estado $T = 0$. Como en el Capítulo 7, escribimos versiones proposicionales de los axiomas de

estado-sucesivo desarrollado para el cálculo situado (*véase* Capítulo 10). Por ejemplo, tenemos

$$\begin{aligned} En(P_1, JFK)^1 \Leftrightarrow & (En(P_1, JFK)^0 \wedge \neg(Volar(P_1, JFK, SFO)^0 \wedge En(P_1, JFK)^0)) \\ & \vee (Volar(P_1, SFO, JFK)^0 \wedge En(P_1, SFO)^0). \end{aligned} \quad (11.1)$$

Esto es, el avión P_1 estará en JFK a tiempo 1 si estaba en JFK a tiempo 0 y no ha volado a otro lugar, o estaba en SFO a tiempo 0 y ha volado a JFK . Necesitaremos uno de tales axiomas para cada avión, aeropuerto y tramo temporal. Sin embargo, cada aeropuerto adicional añade otra forma de viajar hacia (o desde) un aeropuerto dado y de este modo añade más disyunciones al término del lado derecho de cada axioma.

Con estos axiomas en juego, podemos utilizar el algoritmo de satisfabilidad para encontrar un plan. Debería ser un plan que alcance el objetivo a instante $T = 1$, a saber, el plan en el que dos aviones cambian el lugar. Ahora, supongamos que la KB es:

$$Estado\ initial \wedge axiomas\ de\ estado\ -sucesivo \wedge objetivo^1 \quad (11.2)$$

lo que afirma que el objetivo es verdadero a tiempo $T = 1$. Se puede comprobar que un modelo de KB es la asignación en la que

$$Volar(P_1, JFK, SFO)^0 \text{ y } Volar(P_2, JFK, SFO)^0$$

son verdaderos y todos los otros símbolos de acción son falsos. Hasta aquí, todo correcto. ¿Existen otros modelos posibles que el algoritmo de satisfabilidad podría conocer? Por supuesto que sí. ¿Son todos estos modelos planes satisfactorios? No. Considérese el plan absurdo especificado por los símbolos de acción

$$Volar(P_1, SFO, JFK)^0 \text{ y } Volar(P_1, JFK, SFO)^0 \text{ y } Volar(P_2, JFK, SFO)^0$$

Este plan es absurdo porque el avión P_1 comienza en SFO , de modo que la acción $Volar(P_1, JFK, SFO)^0$ es inviable. Sin embargo, ¡el plan es un modelo de la sentencia de la Ecuación (11.2)!, es decir, es consistente con todo lo que hemos dicho hasta ahora acerca del problema. Para entender por qué, necesitamos mirar más cuidadosamente qué dicen los axiomas de estado sucesivos (Ecuación 11.1) acerca de las condiciones cuyas precondiciones no son satisfechas. Los axiomas predicen correctamente que nada sucederá cuando una de estas acciones sea ejecutada (Ejercicio 11.15), pero *no* dicen nada acerca de qué acciones no pueden ser ejecutadas. Para evitar la generación de planes con acciones prohibidas, debemos añadir **axiomas de precondición** estableciendo que para que una acción ocurra requiere que sus precondiciones sean satisfechas⁶. Por ejemplo, necesitamos:

$$Volar(P_1, JFK, SFO)^0 \Rightarrow En(P_1, JFK)^0$$

Como está establecido que $En(P_1, JFK)^0$ es falso en el estado inicial, este axioma asegura que cada modelo que tenga también $Volar(P_1, JFK, SFO)^0$ es falso. Añadiendo axiomas de precondición, existe un modelo que satisface todos los axiomas cuando el objetivo pretende ser alcanzado a tiempo 1, en concreto el modelo en el que el avión P_1

⁶ Notemos que añadir axiomas de precondición significa que no necesitamos incluir precondiciones para la acción en axiomas de estado-sucesor.

vuela hasta *JFK* y el P_2 vuela a *SFO*. Notemos que esta solución posee dos acciones paralelas, de modo equivalente a utilizar GRAPHPLAN o POP.

Más sorpresas emergen cuando añadimos un tercer aeropuerto, *LAX*. Ahora, cada avión tiene dos acciones que son legales en cada estado. Cuando utilizamos el algoritmo de satisfabilidad, encontramos que un modelo con $Volar(P_1, SFO, JFK)^0$ y $Volar(P_2, JFK, SFO)^0$ y $Volar(P_2, JFK, LAX)^0$ satisface todos los axiomas. Esto es, el estado-sucesivo y los axiomas de precondición permiten a un avión volar hacia dos destinos a la misma vez. Las precondiciones para ambos vuelos de P_2 son satisfechas en el estado inicial; los axiomas de estado-sucesivo nos dicen que P_2 estará en *SFO* y en *LAX* a tiempo 1; por tanto el objetivo es satisfecho. Evidentemente, debemos añadir más axiomas para eliminar estas soluciones espurias. Un método es añadir **axiomas de exclusión de acciones** que preventgan acciones simultáneas. Por ejemplo, podemos insistir en la exclusión completa añadiendo todos los posibles axiomas de la forma,

$$\neg(Volar(P_2, JFK, SFO)^0 \wedge Volar(P_2, JFK, LAX)^0)$$

Estos axiomas aseguran que no pueden ocurrir dos acciones al mismo tiempo. Eliminan planes espurios, y también fuerzan a que cada plan esté totalmente ordenado. Esto hace perder la flexibilidad a los planes ordenados parcialmente; también, mediante el aumento de etapas temporales en un plan, el tiempo de computación puede extenderse.

En vez de exclusiones completas, podemos exigir sólo una exclusión parcial, esto es, descartar acciones simultáneas sólo si interfieren con otras. Las condiciones son las mismas que para las acciones excluyentes mutuamente: dos acciones no pueden ocurrir simultáneamente si una niega una precondición o el efecto de la otra. Por ejemplo, $Volar(P_2, JFK, SFO)^0$ y $Volar(P_2, JFK, LAX)^0$ no pueden ocurrir ambas, porque cada una niega la precondición de la otra; por otro lado, $Volar(P_1, SFO, JFK)^0$ y $Volar(P_2, JFK, SFO)^0$ pueden ocurrir juntas porque los dos planes no interfieren. La exclusión parcial elimina planes espurios sin exigir una ordenación completa.

Los axiomas de exclusión a veces parecen instrumentos muy contundentes. En lugar de decir que un avión no puede volar a dos aeropuertos al mismo tiempo, podemos exigir simplemente que ningún objeto pueda estar en dos lugares a la vez.

$$\forall p, x, y, t \ x \neq y \Rightarrow \neg(En(p, x)^t \wedge En(p, y)^t)$$

Este hecho, combinado con los axiomas de estado-sucesivo, implica que un avión no puede volar a dos aeropuertos al mismo tiempo. Hechos como este se conocen como **restricciones de estado**. En lógica proposicional, por supuesto, tenemos que escribir todas las instancias que cubren la restricción de cada estado. En el problema del aeropuerto, la restricción de estado basta para descartar todos los planes espurios. Las restricciones de estado frecuentemente son mucho más compactas que los axiomas de exclusión de acciones, pero no son siempre fáciles de derivar de la descripción STRIPS del problema.

Resumiendo, la planificación como satisfabilidad comprende la búsqueda de modelos de sentencia conteniendo el estado inicial, el objetivo, los axiomas de estado sucesivos, los axiomas de precondición, y por último la acción de exclusión de axiomas o las restricciones de estado. Puede mostrarse que esta colección de axiomas es suficiente, en el sentido de que no hay soluciones espurias. Cualquier modelo que satisfaga la sentencia

proposicional será un plan válido para el problema original, esto es, cada linealización del plan es una secuencia correcta de acciones que alcanza el objetivo.

Complejidad de codificaciones proposicionales

El principal inconveniente del enfoque proposicional es el tamaño ingente del conocimiento que se genera del problema a partir del problema de planificación original. Por ejemplo, el esquema de acción $Volar(p, a_1, a_2)$ se transforma en $T \times |Planes| \times |Aero-puertos|^2$ símbolos proposicionales diferentes. En general, el número total de símbolos de acción viene dado por $T \times |Act| \times |O|^P$, donde $|Act|$ es el número de los esquemas de acción, $|O|$ es el número de objetos en el dominio, y P es el máximo número de argumentos de cualquier esquema de acción. El número de cláusulas es aún elevado. Por ejemplo, con 10 escalones temporales, 12 aviones y 30 aeropuertos, el axioma de exclusión de acción completa tiene 583 millones de cláusulas.

Como el número de símbolos de acción es exponencial en el número de argumentos del esquema de acción, una solución podría ser el intento de reducir el número de argumentos. Podemos hacer esto tomando prestada una idea de las redes semánticas (Capítulo 10). Las redes semánticas usan sólo predicados binarios; predicados con un mayor número de argumentos serían reducidos a un conjunto de predicados binarios que describirían cada argumento por separado. Al aplicar esta idea a un símbolo de acción tal como $Volar(P_1, SFO, JFK)^0$, se obtienen tres nuevos símbolos:

- $Volar_1(P_1)^0$: el avión P_1 ha volado a tiempo 0
- $Volar_2(SFO)^0$: el origen del vuelo fue SFO
- $Volar_3(JFK)^0$: el destino del vuelo fue JFK

DIVISIÓN DE SÍMBOLOS

Este proceso, llamado «**división de símbolos**» elimina la necesidad de un número exponencial de símbolos. Ahora sólo necesitaremos $T \times |Act| \times P \times |O|$.

La división de símbolos, por ella misma, puede reducir el número de símbolos, pero no reduce automáticamente el número de axiomas en la Base de Conocimiento. Esto es, si cada símbolo de acción en cada cláusula fuera simplemente reemplazado por el conjunto de tres símbolos, entonces el tamaño total de la Base de Conocimiento podría permanecer prácticamente equivalente. La división de símbolos realmente reduce el tamaño de la Base de Conocimiento porque algunos de los símbolos escindidos son irrelevantes a ciertas acciones y pueden ser omitidos. Por ejemplo, consideremos el axioma de estado-sucesivo en la Ecuación 11.1, modificado para incluir LAX y para omitir precondiciones de acción (las cuales son cubiertas por axiomas de precondición separados):

$$\begin{aligned} En(P_1, JFK)^1 \Leftrightarrow & (En(P_1, JFK)^0 \wedge \neg Volar(P_1, JFK, SFO)^0 \wedge \neg Volar(P_1, JFK, LAX)^0) \\ & \vee (Volar(P_1, SFO, JFK)^0 \vee (Volar(P_1, LAX, JFK)^0) \end{aligned}$$

La primera condición dice que P_1 estará en JFK si estuviese allí a tiempo 0 y no volase desde JFK a cualquier otra ciudad, no importa cuál. Usando símbolos divididos, podemos simplemente omitir el argumento cuyo valor no nos interesa:

$$\begin{aligned} En(P_1, JFK)^1 \Leftrightarrow & (En(P_1, JFK)^0 \wedge \neg(Volar_1(P_1)^0 \wedge Volar_2(JFK)^0)) \\ & \vee (Volar_1(P_1)^0 \wedge Volar_3(JFK)^0) \end{aligned}$$

Notemos que *SFO* y *LAX* no son mencionados más en el axioma. De forma más general, la división de símbolos de acción permiten ahora que el tamaño de cada axioma de estado-sucesivo sea independiente del número de aeropuertos. Reducciones similares ocurren con los axiomas de precondición y los axiomas de exclusión de acción (véase Ejercicio 11.6). Para el caso descrito antes con 10 tramos temporales, 12 aviones y 30 aeropuertos, el axioma de exclusión completa es reducida de 583 millones de cláusulas a 9.360 cláusulas.

Existe un inconveniente: la representación de división-de-símbolos no permite acciones paralelas. Consideremos las dos acciones paralelas $Volar(P_1, SFO, JFK)^0$ y $Volar(P_2, JFK, SFO)^0$. Transformándolos a una representación dividida, tenemos:

$$\begin{aligned} Volar_1(P_1)^0 \wedge Volar_2(SFO)^0 \wedge Volar_3(JFK)^0 \wedge \\ Volar_1(P_2)^0 \wedge Volar_2(JFK)^0 \wedge Volar_3(SFO)^0 \end{aligned}$$

No es posible determinar lo que ha sucedido. Sabemos que P_1 y P_2 volaron, pero no podemos identificar el origen y el destino de cada vuelo. Esto significa que un axioma de exclusión completa debe ser usado, con el inconveniente mencionado previamente.

Los planificadores basados en la satisfactoriedad pueden abordar importantes problemas de planificación, por ejemplo, encontrar soluciones óptimas de 30 etapas para problemas de planificación en mundos de bloques con docenas de bloques. El tamaño de la codificación proposicional y el coste de la solución son problemas fuertemente dependientes, sin embargo en muchos casos la memoria requerida para almacenar los axiomas proposicionales se convierte en un cuello de botella. Una interesante forma de plantear este problema ha sido mediante algoritmos de retro-propagación, tales como DPLL, que son muchas veces mejores para la solución de problemas que los algoritmos de búsquedas locales como WALKSAT. Esto es porque la mayoría de los axiomas proposicionales son cláusulas de Horn, las cuales son tratadas de manera eficiente por la unidad de propagación técnica. Esta observación ha dirigido la investigación hacia el desarrollo de algoritmos híbridos combinando búsquedas aleatorias con mecanismos de retro-propagación.

11.6 Análisis de los enfoques de planificación

La planificación es, actualmente, un área de gran interés dentro de la IA. Una razón de esto es que combina las dos mayores áreas de IA que se han tratado hasta ahora: *búsquedas* y *lógica*. Esto es, un planificador puede ser visto tanto como un programa que busca la solución o como uno que (constructivamente) proporcione la existencia de una solución. Lo fértil del cruce de ideas desde ambas áreas ha llevado a un avance en la pasada década de varios órdenes de magnitud y un aumento del uso de planificadores en aplicaciones industriales. Desafortunadamente, no tenemos una clara comprensión de qué técnicas trabajan mejor en qué clase de problemas. Muy posiblemente, emergan nuevas técnicas que sobrepasarán a los actuales métodos.

La planificación es, en primer lugar, un ejercicio de control de la explosión combinatoria. Si tenemos un número p de proposiciones primitivas en un dominio, tendremos

2^p estados. Para dominios complejos, p puede crecer mucho. Consideremos que los objetos en el dominio tienen propiedades (*localización, color*) y relaciones (*en, sobre, entre*). Con d objetos en un dominio que mantengan relaciones cada tres, tendremos 2^{d^3} estados. Podremos concluir que, en el peor de los casos, la planificación es una tarea sin solución.

En contra de tal pesimismo, la estrategia divide-y-vencerás puede ser una poderosa arma. En el mejor caso (la descomposición completa del problema) divide-y-vencerás nos ofrece una agilización exponencial. La descomposibilidad no aporta ventajas, sin embargo, en caso de interacciones negativas entre acciones. Los planificadores de orden parcial afrontan estos problemas con enlaces causales, un poderoso enfoque de representación, pero desafortunadamente cada conflicto debe ser resuelto con una elección, y las elecciones pueden multiplicarse exponencialmente. GRAPHPLAN evita estas selecciones durante la fase de construcción del grafo, utilizando enlaces de exclusión mutua para registrar conflictos sin construir elecciones para resolverlos. SATPLAN representa un rango similar de relaciones mutuamente excluyentes, pero lo hace mediante el uso de la forma general CNF preferible a una estructura específica de datos. Lo bien o mal que vaya dependerá del solucionador SAT usado.

Algunas veces es posible resolver un problema eficientemente, aceptando que las interacciones negativas pueden ser descartadas. Decimos que un problema tiene **sub-objetivos serializables** si existe un orden de sub-objetivos tal que el planificador pueda alcanzarlos en ese orden, sin tener que anular ninguno de los sub-objetivos alcanzados previamente. Por ejemplo, en el mundo de bloques, si el objetivo es construir una torre (ejemplo: *A sobre B*, que se encuentra sobre *C*, que a su vez está sobre el *Tablero*), entonces los sub-objetivos son serializables de abajo-a-arriba: si primeramente situamos *C* sobre el *Tablero*, no tendremos que deshacer esta meta mientras estamos alcanzando el resto de sub-objetivos. Un planificador que usa la estrategia abajo-a-arriba puede solucionar cualquier problema en el dominio del mundo de bloques sin retro-propagación (aunque puede no encontrar siempre el mejor plan).

Presentemos un ejemplo complejo: para el agente planificador remoto que comandaba la misión espacial Deep Space One se determinó que las proposiciones vinculadas en la dirección de la misión eran serializables. Esto quizá no es demasiado sorprendente, porque una nave espacial es diseñada por ingenieros para que su control sea lo más sencillo posible (sujeto a otras restricciones). Tomando ventaja de la ordenación serializada de objetivos, el agente planificador remoto fue capaz de eliminar muchas de las búsquedas. Esto se tradujo en un control de la nave en tiempo real, algo que previamente se consideraba imposible.

Existe más de un modo de control de explosiones combinatorias. Vimos en el Capítulo 5 que hay muchas técnicas para el control de la retro-propagación en problemas que satisfagan restricciones (CSPs), tales como retro-propagación dependientemente dirigida. Todas estas técnicas pueden ser aplicadas a la planificación. Por ejemplo, extraer una solución de un grafo de planificación puede ser formulado como un problema CSP booleano cuyas variables establezcan si una acción dada debería ocurrir en un tiempo determinado. El CSP puede ser resuelto usando cualquiera de los algoritmos presentados en el Capítulo 5. Un método estrechamente relacionado, usado en sistemas BLACKBOX, convierte el grafo de planificación en una expresión CNF y posteriormente

te extrae un plan utilizando un solucionador SAT. Este enfoque parece trabajar mejor que SATPLAN, a priori porque el grafo de planificación ha eliminado ya muchos de los estados imposibles y acciones del problema. Funciona también mejor que GRAPHPLAN, presumiblemente porque una búsqueda de satisfabilidad tal como WALKSAT tiene mucha más flexibilidad que una simple búsqueda de retro-propagación que use GRAPHPLAN.

No hay duda de que planificadores tales como GRAPHPLAN, SATPLAN y BLACKBOX han hecho progresar el campo de la planificación, tanto por la versatilidad de los sistemas de planificación como por la clarificación a nivel representacional de los temas relacionados. Estos métodos son, por tanto, inherentemente proposicionales y están limitados a los dominios donde pueden expresarse. (Por ejemplo, problemas logísticos con una pequeña docena de objetos y de localizaciones, pueden requerir gigabytes de almacenamiento para las correspondientes expresiones CNF.) Parece probable que se requieran representaciones de primer orden y nuevos algoritmos que acompañen el progreso en el área, mientras las estructuras tales como grafos de planificación continuarán siendo útiles como fuente de heurísticas.

11.7 Resumen

En este capítulo, hemos definido el problema de planificación en entornos deterministas, y completamente observables. Describimos las representaciones principales usadas para problemas de planificación y varias estrategias algorítmicas para resolverlos. Los elementos que tenemos que recordar son:

- Los sistemas de planificación son algoritmos de resolución de problemas que operan con representaciones proposicionales explícitas (de primer orden) de estado y acciones. Estas representaciones hacen posible la obtención de heurísticas efectivas y el desarrollo de poderosos y flexibles algoritmos para la resolución de problemas.
- El lenguaje STRIPS describe acciones en términos de sus precondiciones y efectos y describe el estado inicial y objetivo como secuencias de conjunciones entre literales positivos. El lenguaje ADL relaja algunas de estas restricciones, permitiendo la disyunción, la negación y los cuantificadores.
- Las búsquedas en el espacio de estados pueden operar hacia delante (**progresión**) o hacia atrás (**regresión**). Heurísticas efectivas pueden obtenerse aceptando la hipótesis de sub-objetivos independientes y mediante varias aproximaciones del problema de planificación.
- Los algoritmos de Planificación de Orden Parcial (POP) exploran el espacio de planes sin comprometerse con una secuencia de acciones totalmente ordenada. Trabajando hacia atrás desde el objetivo y añadiendo acciones para planificar cómo alcanzar cada sub-objetivo. Es particularmente efectivo en problemas en los que se puede utilizar una estrategia divide-y-vencerás.
- Un **grafo de planificación** puede ser construido de manera incremental, partiendo del estado inicial. Cada capa contiene un súper-conjunto de todos los literales

que podrían ocurrir en cada tramo temporal y codifica las **relaciones de exclusión mutua**, las relaciones entre literales o entre acciones que no puedan darse a la vez. Los grafos de planificación nos proporcionan útiles heurísticas en el espacio de estados, para planificadores de orden parcial, y pueden ser usados directamente en el algoritmo GRAPHPLAN.

- El algoritmo GRAPHPLAN procesa el grafo de planificación, usando una búsqueda hacia atrás hasta extraer un plan. Esto tiene en cuenta ordenaciones parciales entre acciones.
- El algoritmo SATPLAN traduce un problema de planificación en axiomas proposicionales y aplica un algoritmo de satisfabilidad para encontrar un modelo que se corresponda con un plan válido. Diferentes representaciones proposicionales han sido desarrolladas variando el grado de concisión y eficiencia.
- Cada una de las estrategias de planificación tiene sus adeptos y no existen consensos sobre cuál es la mejor. La competencia entre planteamientos y lo fértil del cruce y combinación de diferentes enfoques se ha traducido en ganancias significativas en eficiencia para sistemas de planificación.



NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

La planificación en IA emerge de la investigación en áreas, como búsquedas en el espacio de estados, demostración de teoremas y teoría de control, y desde las necesidades prácticas en robótica, organización y otros dominios. STRIPS (Fikes y Nilsson, 1971), el primero de los grandes sistemas de planificación, ilustra la interacción de dichas influencias. STRIPS fue diseñado como un componente para la planificación *software* del robot Shakey proyectado en el SRI. Su estructura de control total fue modelada en GPS (*General Problem Solver* (Newell y Simon, 1961), un sistema de búsqueda en el espacio de estados que utilizaba un mecanismo de análisis de fines/medios. STRIPS usaba una versión del sistema de demostración de teoremas QA3 (Green, 1969b) como una subrutina para establecer la verdad de las precondiciones de las acciones. Lifschitz (1986) ofrece precisas definiciones y un análisis del lenguaje STRIPS. Bylander (1992) muestra simples planificaciones STRIPS para PSPACE-completos. Fikes y Nilsson (1993) nos ofrecen una retrospectiva histórica del proyecto STRIPS y una revisión de sus relaciones con los esfuerzos más recientes en planificación.

La representación de la acción usada por STRIPS ha sido más influyente que su estrategia algorítmica. Casi todos los sistemas de planificación desde entonces han usado una variante u otra del lenguaje STRIPS. Desafortunadamente, la proliferación de variantes ha hecho las comparaciones necesariamente difíciles. Una mejor comprensión de las limitaciones e intercambios entre formalismos necesitaría más tiempo. El Lenguaje de Descripción de Acciones, ADL, (Pednault, 1986) relajó algunas de las restricciones en el lenguaje STRIPS e hizo posible tratar problemas más complejos. Nebel (2000) explora esquemas para compilar ADL en STRIPS. El problema del Lenguaje de Descripción de dominios o PDL (Ghallab *et al.*, 1998) fue introducido como un sintaxis estandarizada para la representación de STRIPS, ADL y otros lenguajes. PDDL ha sido usado como el

lenguaje estándar para las competiciones de planificación en la conferencia AIPS, iniciadas el año 1998.

Los planificadores, a principios de la década de los 70, generalmente trabajaban con secuencias de acciones ordenadas. La descomposición de problemas fue alcanzada por la computación de un sub-plan para cada sub-objetivo y encadenando los sub-planes en algún orden. Este enfoque, llamado por Sacerdoti **planificación lineal** (1975), pronto se demostró que era incompleto. No podía resolver algunos problemas muy simples, tales como la anomalía de Sussman (*véase Ejercicio 11.11*), planteado por Allen Brown durante su experimentación con el sistema HACKER (Sussman, 1975). Un planificador completo debe permitir **intercalar** acciones de diferentes sub-planes dentro de una simple secuencia. La noción de sub-objetivos serializables (Korf, 1987) se corresponde exactamente con el conjunto de problemas para los que los planificadores no-intercalables son completos.

Una solución para problemas intercalados fue la planificación de regresión de objetivos, una técnica con la que se reordenan las fases de un plan totalmente ordenado, con el fin de evitar conflictos entre sub-objetivos. Esto fue introducido por Waldinger (1975) y también usado por el WARPLAN de Warren (1974). WARPLAN es también destacable porque fue el primer planificador en ser descrito un lenguaje de programación lógico (Prolog) y es uno de los mejores ejemplos de las posibilidades que pueden algún día abrirse con el uso de la programación lógica: WARPLAN tiene solamente 100 líneas de código, una pequeña fracción de tamaño en tiempo comparable con otros planificadores. INTERPLAN (Tate, 1975a, 1975b) también permitió el intercalado arbitrario de etapas de un plan para compensar la anomalía de Sussman y los problemas relacionados.

La idea sobre las que se fundamenta que la planificación de orden parcial incluye la detección de conflictos (Tate, 1975a), y la protección de condiciones alcanzadas desde la interferencia (Sussman, 1975). La construcción de planes parcialmente ordenados (en su momento llamados **redes de tareas**) fue precursora del planificador NOAH (Sacerdoti, 1975, 1977) y el sistema NONLIN de Tate (1975b, 1977)⁷.

La planificación de orden parcial dominó los 20 años siguientes de investigación, y durante gran parte de este tiempo, las aportaciones a esta área no fueron del todo entendidas. TWEAK (Chapman, 1987) fue una reconstrucción lógica y una simplificación del trabajo en planificación durante este tiempo; su formulación fue suficientemente clara para permitir demostraciones de completitud e intratabilidad (NP-completitud e indecidibilidad) o diversas formulaciones del problema de planificación. El trabajo de Chapman nos lleva hasta la que fue posiblemente la primera descripción legible de un planificador de orden parcial completo (McAllester y Rosenblitt, 1991). Una implementación del algoritmo de McAllester y Rosenblitt llamado SNLP (Soderland y Weld, 1991) fue ampliamente distribuido y permitió a muchos investigadores entender y experimentar con planificación de orden parcial por primera vez. El algoritmo POP descrito en este capítulo está basado en SNLP.

⁷ Puede existir alguna confusión con la terminología. Muchos autores usan el término **no-lineal** para referirse a parcialmente ordenado. Esto es significativamente diferente del uso original de Sacerdoti para referirse a planes intercambiables.

El grupo de Weld también desarrolló UCPOP (Penberthy y Weld, 1992), el primer planificador para problemas expresados en ADL. UCPOP incorporó la heurística de número de objetivos insatisfechos. Por primera vez se obtuvo un planificador más rápido que SNLP, pero casi nunca fue capaz de encontrar planes con más de una docena de pasos. Aunque heurísticas mejoradas fueron desarrolladas por UCPOP (Joslin y Pollack, 1994; Gereveni y Schubert, 1996), la planificación de orden parcial se vio interrumpida en la década de los 90 cuando emergieron métodos más rápidos. Nguyen y Kambhampati (2001) sugirieron que una renovación era necesaria: con heurísticas adecuadas extraídas de un grafo de planificación, y desarrollaron el planificador REPOP que avanzaba mucho mejor que GRAPHPLAN y era competitivo frente a planificadores de espacio de estados más rápidos.

Avrim Blum y Merrick Furst (1995, 1997) revitalizaron el campo de la planificación con su sistema GRAPHPLAN, que fue más rápido en varios órdenes de magnitud que los planificadores de orden parcial. Otros sistemas de planificación de grafos, tales como IPP (Koehler *et al.*, 1997), STAN (Fox y Long, 1998) y SGP (Weld *et al.* 1998), pronto se pondrían en práctica. Una estructura de datos estrechamente relacionada con el grafo de planificación ha sido desarrollado hace relativamente poco por Ghallab y Laruelle (1994), cuyo planificador de orden parcial IXTE_T se usa para derivar adecuadas heurísticas que guíen las búsquedas. Nuestra discusión sobre grafos de planificación está basada parcialmente en este trabajo y en los trabajos de Subbarao Kambhampati. Como hemos mencionado en este capítulo, un grafo de planificación puede ser usado de muchas formas diferentes para guiar la búsqueda de una solución. El vencedor de la competición de planificación AIPS en la edición de 2002, LPG (Gerevini y Serina, 2002), buscaba grafos de planificación usando una técnica de búsqueda local inspirada por WALKSAT.

La planificación como satisfabilidad y el algoritmo SATPLAN fueron propuestos por Kautz y Selman (1992), que estuvieron inspirados por el sorprendente éxito de las búsquedas locales para la satisfacción de problemas (véase Capítulo 7). Kautz *et al.* (1996) también investigaron diferentes formas de representaciones proposicionales utilizando axiomas STRIPS, llegando al resultado de que las formas más compactas necesariamente no llevaban a las soluciones más satisfactorias a tiempo. Un análisis sistemático fue realizado por Ernst *et al.* (1997), quien también desarrolló un «compilador» automático para la generación de representaciones proposicionales de problemas PDL. El planificador BLACKBOX, que combina ideas de GRAPHPLAN y SATPLAN, fue desarrollado por Kautz y Selman (1998).

El resurgimiento de interés por la planificación en espacio de estados fue promovido por Drew McDermott y su programa UNPOP (1996), y fue el primero en sugerir una distancia heurística basada en la aproximación de un problema con la eliminación de listas ignoradas. El nombre UNPOP fue una reacción a la insoportable concentración de planificadores de orden parcial; McDermott sospechaba que no se le había prestado la suficiente atención que merecían otros enfoques. El planificador de búsqueda heurística de Bonet y Geffner (HSP) y sus posteriores variaciones (Bonet y Geffner, 1999) fueron los primeros en plantear búsquedas prácticas en espacio de estados para problemas de planificación complejos. El buscador de espacio de estados más exitoso en la actualidad es FASTFORWARD o FF (2000) de Hoffmann, ganador en la competición de plani-

ficación AIPS 2000. FF usa una heurística de planificación de grafos simplificada con un algoritmo de búsqueda muy rápido que combina búsquedas locales y hacia delante, de un modo novedoso.

Más recientemente, ha habido interés en la representación de planes como **diagramas de decisión binaria**, una descripción compacta de autómatas finitos ampliamente estudiada en la comunidad *hardware* (Clarke y Grumberg, 1987; McMillan, 1993). Existen técnicas para proporcionar propiedades de diagramas de decisión binaria, incluyendo la propiedad de ser una solución a un problema de planificación. Cimatti *et al.* (1998) presenta un planificador basado en este enfoque. Otras representaciones también han sido usadas; por ejemplo, Vossen *et al.* (2001) examinan el uso de la programación entera para la planificación.

Existen, aunque no de manera definitiva, algunas nuevas e interesantes comparaciones entre los diferentes enfoques de planificación. Helmert (2001) analiza varias clases de problemas de planificación, y muestra que el enfoque basado en restricciones, tal como GRAPHPLAN y SATPLAN son los mejores para dominios NP-duros, mientras que enfoques basados en búsquedas funcionan mejor en dominios donde soluciones factibles puedan ser encontradas sin técnicas de retro-propagación. GRAPHPLAN y SATPLAN tienen problemas en dominios con varios objetos, porque pueden crear varias acciones. En algunos casos, el problema puede ser aplazado o evitado por la generación dinámica de acciones proposicionalizadas preferiblemente que mediante la instanciación antes de que la búsqueda comience.

Weld (1994, 1999) proporciona dos excelentes revisiones de los algoritmos de planificación modernos. Es interesante ver el cambio que se produce entre los cinco años que pasan entre ambos enfoques: el primero se concentra en planificación de orden parcial, y el segundo introduce GRAPHPLAN y SATPLAN. *Readings in Planning* (Allen *et al.*, 1990) es una antología exhaustiva de muchos de los mejores artículos recientes en el área, incluyendo buenas revisiones. Yang (1997) proporciona una perspectiva general de técnicas de planificación de orden parcial.

La planificación ha sido central en la IA desde su inicio, y los artículos sobre planificación son un ingrediente de las principales revistas especializadas y conferencias. Existen conferencias especializadas tales como la Conferencia Internacional de Sistemas de Planificación en IA (AIPS), el Workshop Internacional en Planificación y Organización Espacial, y la Conferencia Europea sobre Planificación.

EJERCICIOS



11.1 Describa las diferencias y similitudes entre la resolución de problemas y la planificación.

11.2 Dados los axiomas de la Figura 11.2 ¿Cuáles son las instancias concretas aplicables a *Volar(p, desde, hasta)* en el estado descrito por

$$\begin{aligned} & \text{En}(P_1, \text{JFK}) \wedge \text{En}(P_2, \text{SFO}) \wedge \text{Avión}(P_1) \wedge \text{Avión}(P_2) \\ & \wedge \text{Aeropuerto}(\text{JFK}) \wedge \text{Aeropuerto}(\text{SFO}) ? \end{aligned}$$

11.3 Consideremos cómo podríamos traducir un conjunto de esquemas STRIPS en un conjunto de axiomas de cálculo situado y de estado-sucesivo (véase Capítulo 10).

- Considérese el esquema para *Volar(p, desde, hasta)*. Escriba una definición lógica para el predicado *VolarPrecondición(p, desde, hasta, s)*, lo cual es correcto si las precondiciones para *Volar(p, desde, hasta)* son satisfechas en la situación *s*.
- En segundo lugar, y asumiendo que *Volar(p, desde, hasta)* es el único esquema de acción disponible al agente, escriba un axioma de estado-sucesivo para *En(p, x, s)* que capture la misma información que el esquema de acción.
- Ahora supongamos que existe un método de viaje adicional: *Transporte(p, desde, hasta)*. Se tiene la precondición adicional $\neg \text{Modificado}(p)$ y el efecto adicional *Modificado(p)*. Explique cómo debe ser modificada la base de conocimiento de cálculo situado.
- Finalmente, desarrolle un procedimiento general y específico precisamente para llevar a cabo el traslado desde un conjunto de esquemas STRIPS a un conjunto de axiomas de estado sucesivo.

11.4 El problema del mono y los plátanos viene representado por un mono en un laboratorio con algunas bananas colgadas del techo fuera de su alcance. Una caja se encuentra disponible y le capacitaría para alcanzar las bananas si el mono se sube encima. Inicialmente, el mono está sobre *A*, las bananas sobre *B*, y la caja en *C*. El mono y la caja tiene peso *Poco*, pero si el mono se sube a la caja, el conjunto adquiere un peso *Elevado*, de igual valor que las bananas. Las acciones disponibles para el mono incluyen *Ir* de un lado a otro, *Empujar* un objeto de un lugar a otro, *Subirse o Bajarse* de un objeto, y *Agarrar o Soltar* un objeto. *Agarrar* es el resultado de la manipulación de un objeto si el mono y el objeto están en el mismo lugar con el mismo peso.

- a) Escriba la descripción del estado inicial.
- b) Escriba las definiciones, en forma STRIPS, de las seis acciones.
- c) Supongamos que el mono quiere confundir al científico, que ha salido a tomar el té, agarrando las bananas pero colocando de nuevo la caja en su sitio inicial. Escriba este esquema como un objetivo general (es decir, sin asumir que la caja está necesariamente en *C*) en lenguaje de cálculo situado. ¿Podría este objetivo ser alcanzado por un sistema de tipo STRIPS?
- d) El axioma para *Empujar* es probablemente incorrecto, porque si el objeto es demasiado pesado, su posición permanecerá siendo la misma aún cuando el operador *Empujar* sea aplicado. ¿Es este un ejemplo de problema de ramificación o de cualificación? Asegúrese de que la descripción del problema sirva para objetos pesados.

11.5 Explique por qué el proceso para la generación de precedentes en búsquedas hacia-atrás no necesita añadir literales que representen los efectos negativos de la acción.

11.6 Explique por qué al eliminar los efectos negativos de cada esquema de acción en un problema STRIPS, se obtiene un problema aproximado.

11.7 Examine la definición de **búsqueda bidireccional** del Capítulo 3.

- Podría ser una búsqueda bidireccional en el espacio de estados una buena idea para la planificación?
- ¿Y una búsqueda bidireccional en el espacio de planes de orden parcial?
- Invente una versión de planificación de orden parcial en el que una acción pueda ser añadida a un plan si sus precondiciones pueden ser alcanzadas por los efectos de acciones ya existentes en el plan. Explique cómo tratar con conflictos y ordenamiento de restricciones. ¿Es el algoritmo esencialmente idéntico a una búsqueda hacia-delante en el espacio de estados?
- Consideremos un planificador de orden parcial que combine el método de la parte (c) con el método estándar de añadir acciones para alcanzar condiciones abiertas. ¿Podría ser el algoritmo resultante el mismo que en la parte (b)?

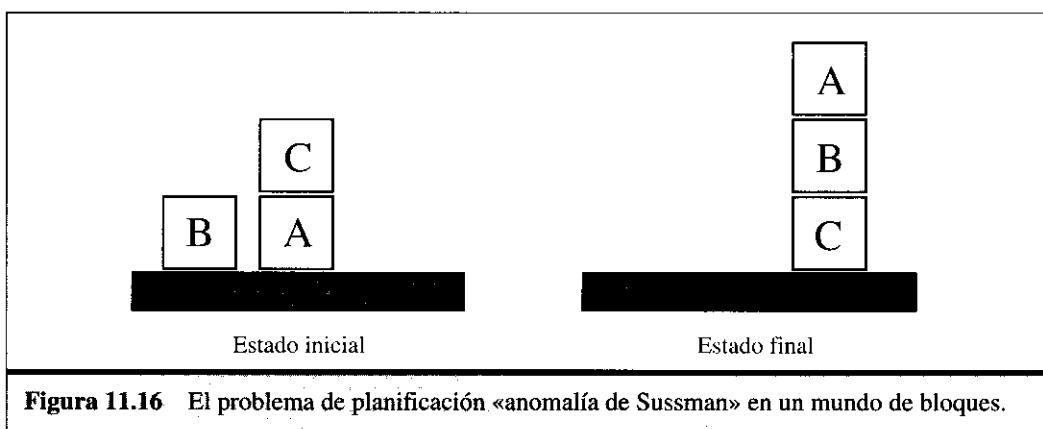
11.8 Construya niveles 0,1 y 2 del grafo de planificación para el problema de la Figura 11.2.

11.9 Demuestre las siguientes afirmaciones acerca de los grafos de planificación:

- Un literal que no aparece en el nivel final del grafo no puede ser alcanzado.
- El coste de nivel de un literal en un grafo serial no es mayor que el coste real de un plan óptimo para alcanzarlo.

11.10 Contrastamos planificadores de búsqueda hacia atrás y hacia delante en el espacio de estados frente a planificadores de orden parcial, diciendo que los últimos son buscadores en el espacio de planes. Explique cómo búsquedas hacia delante y hacia atrás en el espacio de estados pueden también ser consideradas como búsquedas en el espacio de planes, y diga cuáles son los operadores de refinamiento de planes.

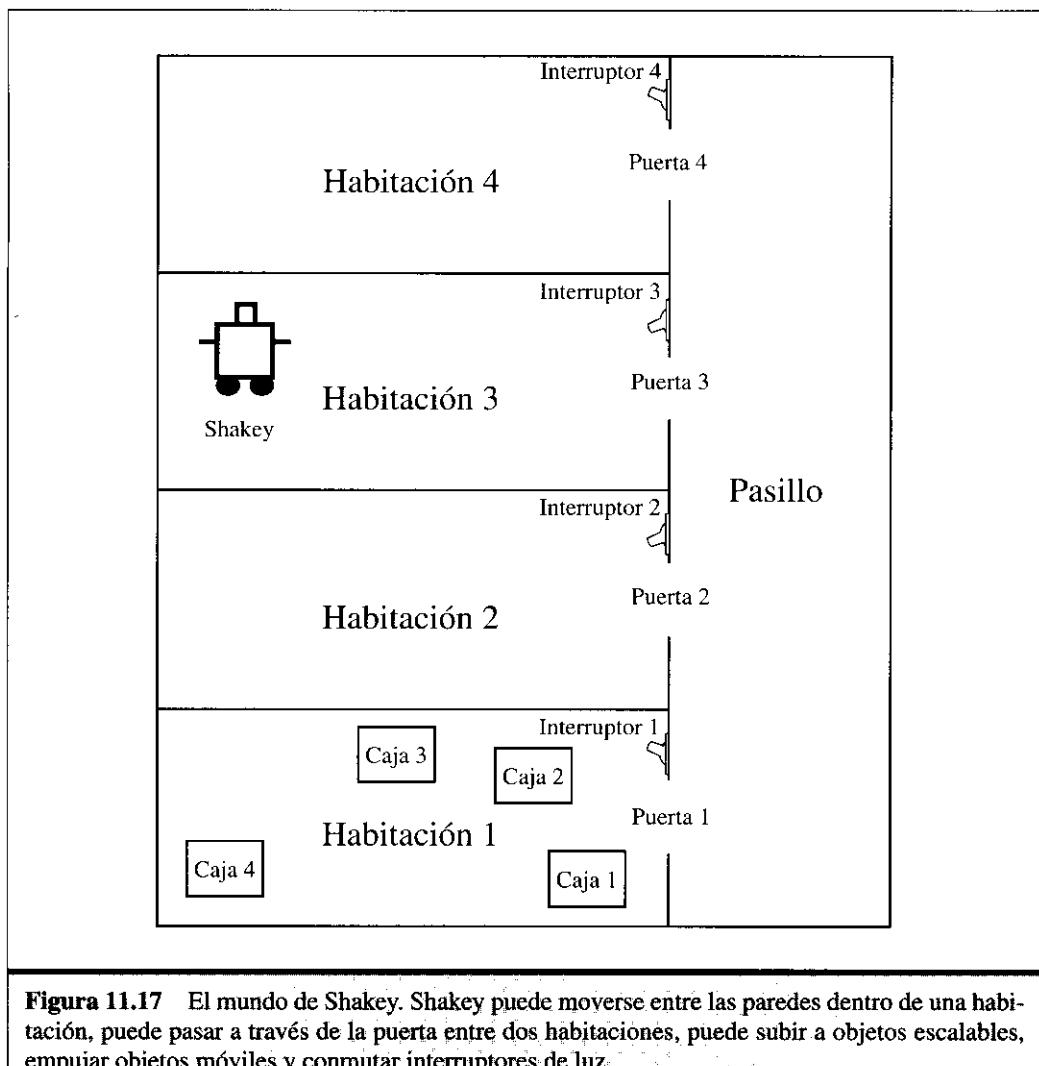
11.11 La Figura 11.16 muestra un problema en un mundo de bloques conocido como la **anomalía de Sussman**. El problema fue considerado anómalo porque los planificadores que no permiten intercalar fases, de principios de los 70, no podían resolverlo. Escriba una definición del problema en notación STRIPS y resuélvalo, tanto a mano como con un programa de planificación. Un planificador no-intercalador es un planificador que, dados dos sub-objetivos G_1 y G_2 , o produce un plan para G_1 concatenado con un



plan para G_2 , o viceversa. Explique por qué no se puede resolver este problema con un planificador de este tipo.

11.12 Consideremos el problema de ponerse uno mismo zapatos y calcetines, tal como definimos en la Sección 11.3. Aplique GRAPHPLAN a este problema y muestre la solución obtenida. Ahora añada acciones para ponerse un abrigo y un sombrero. Muestre el plan de orden parcial que es solución, y muestre que existen 180 linealizaciones diferentes del plan de orden parcial. ¿Cuál es el mínimo número de grafos de planificación diferentes y necesarios para representar las 180 linealizaciones?

11.13 El programa original STRIPS fue diseñado para controlar al robot Shakey. La Figura 11.17 muestra una versión del mundo de Shakey que consiste en cuatro habitaciones alineadas a lo largo de un pasillo; cada habitación tiene una puerta y un interruptor de luz.



Las acciones en el mundo de Shakey incluyen: moverse de un sitio a otro, empujar objetos móviles (tales como cajas), escalar y bajar de objetos rígidos (tales como cajas), y encender y apagar las luces. El robot por sí mismo nunca fue suficientemente hábil para subirse sobre una caja o comutar el interruptor para que la luz pasase de encendida a apagada y al contrario, pero el planificador STRIPS fue capaz de encontrar e imprimir planes que iban más allá de las habilidades del robot. Las seis acciones de Shakey eran las siguientes:

- $Ir(x, y)$, que requiere que Shakey esté en x , además de que x e y estén localizadas en la misma habitación. Por convención, una puerta entre dos habitaciones está en ambas.
- Empujar una caja b desde el lugar x al lugar y dentro de la misma habitación: $Empujar(b, x, y)$. Necesitaremos el predicado *Caja* y constantes para las cajas.
- Subirse a una caja: $Subir(b)$; Bajar de una caja: $Bajar(b)$. Necesitaremos el predicado *Sobre* y la constante *Suelo*.
- Encender y apagar una luz: $Encender(s)$, Apagar(s); para encender o apagar la luz, Shakey debe estar sobre una caja que se encuentre en la posición del interruptor.

Describa las seis acciones de Shakey y el estado inicial de la Figura 11.17 en notación STRIPS. Construya un plan para Shakey que logre que la *Caja*₂ esté en la *Habitación*₂.

11.14 Vimos que los grafos de planificación solamente pueden trabajar con acciones proposicionales. ¿Qué ocurre si queremos usar grafos de planificación para un problema con variables en el objetivo, tal como $En(P_1, x) \wedge En(P_2, x)$, donde x cubre el rango sobre un dominio finito de localizaciones? ¿Cómo podría codificarse tal problema para trabajar con grafos de planificación? (Recuerde la acción *Finalizar* del planificador POP. ¿Qué precondiciones debe tener?)

11.15 Hasta ahora hemos asumido que las acciones son sólo ejecutadas en situaciones apropiadas. Veamos lo que tienen que decir axiomas proposicionales de estado sucesivo, como en la Ecuación 11.1 acerca de acciones cuyas precondiciones no son satisfechas.

- a) Muestre que los axiomas predicen que no sucederá nada cuando una acción sea ejecutada en un estado donde sus precondiciones no son satisfechas.
- b) Considérese un plan p que contiene las acciones requeridas para alcanzar un objetivo pero que también incluye acciones no permitidas. ¿Es éste un caso como

$$Estado\;inicial \wedge axiomas\;de\;estado\;sucesivo \wedge p \models objetivo?$$

- c) Con axiomas de estado sucesor y de primer-orden en cálculo situado (como en el Capítulo 10), ¿es posible probar que un plan que contenga acciones ilegales alcanzará el objetivo?

11.16 Utilice el dominio del problema del aeropuerto, para dar ejemplos y explique cómo la división de símbolos reduce el tamaño de los axiomas de precondición y los axiomas de exclusión de acciones. Derive una fórmula general para el tamaño de cada conjunto de axiomas en términos del número de tramos temporales, el número de esquemas de acción y el número de objetos.

11.17 En el algoritmo SATPLAN de la Figura 11.15, cada llamada al algoritmo de satisfabilidad afirma un objetivo g^T , donde T cubre el rango de $T = 0$ a $T = T_{\max}$. Suponga, en lugar de esto, que el algoritmo de satisfabilidad es llamado sólo una vez, con el objetivo $g^0 \vee g^1 \vee \dots \vee g^{T_{\max}}$.

- a) ¿Devuelve siempre un plan si existe uno con longitud menor o igual que T_{\max} ?
- b) ¿Este enfoque introduce alguna nueva «solución» espuria?
- c) Discuta cómo se podría modificar un algoritmo de satisfabilidad tal como WALKSAT de modo que encontrase soluciones cortas (si existen) cuando se dé un objetivo disyuntivo de la forma indicada.