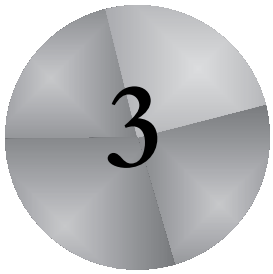

3 Resolver problemas mediante búsqueda

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.

FORMULACIÓN DEL OBJETIVO

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 ciudad[1] determinada.

FORMULACIÓN DEL PROBLEMA

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 este familiarizado con la geografía de Rumanía, no sabría qué carretera seguir[2]. 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.

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*.

BÚSQUEDA

SOLUCIÓN

EJECUCIÓN

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 ← ACTUALIZAR-ESTADO(estado, percepción)
  si sec está vacía entonces hacer
    objetivo ← FORMULAR-OBJETIVO(estado)
    problema ← FORMULAR-PROBLEMA(estado,objetivo)
    sec ← BÚSQUEDA(problema)
  acción ← PRIMERO(secuencia)
  sec ← 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.

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.

LAZO ABIERTO

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*).

FUNCIÓN SUCESOR

- Una descripción de las posibles **acciones** disponibles por el agente. La formulación[3] más común utiliza una **función sucesor**. Dado un estado particular x, SUCESOR-FN(x) devuelve un conjunto de pares ordenados $\langle$*acción, sucesor*$\rangle$, 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.

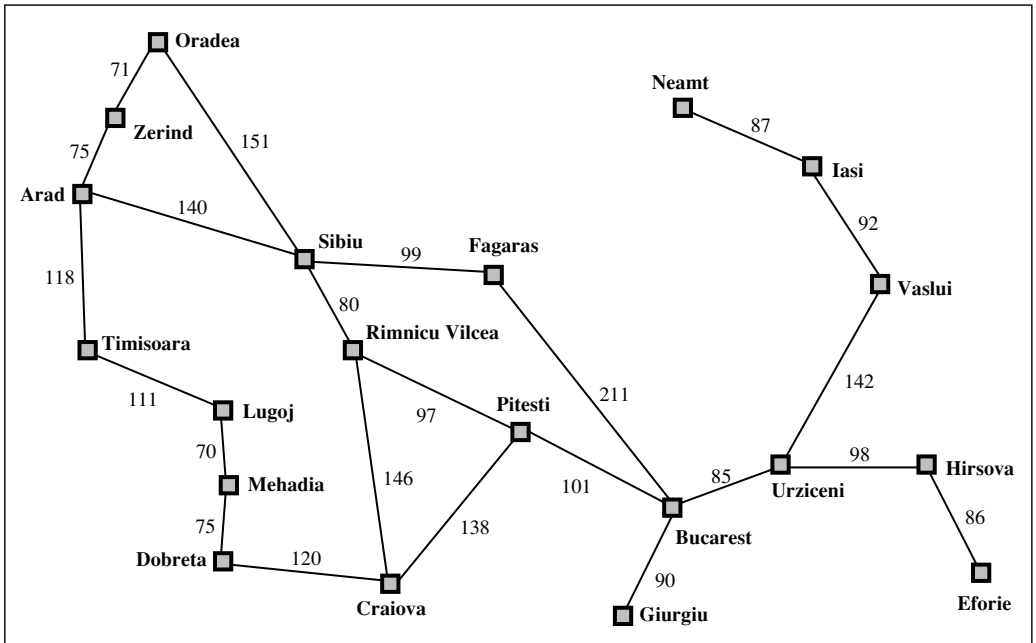

Figura 3.2 Un mapa de carreteras simplificado de parte de Rumanía.

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 negativos[4].

COSTO INDIVIDUAL

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.

SOLUCIÓN ÓPTIMA

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.

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**.

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

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 ejercitar 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.

PROBLEMA DE JUGUETE

PROBLEMA DEL MUNDO REAL

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 $n\ 2^n$ estados.

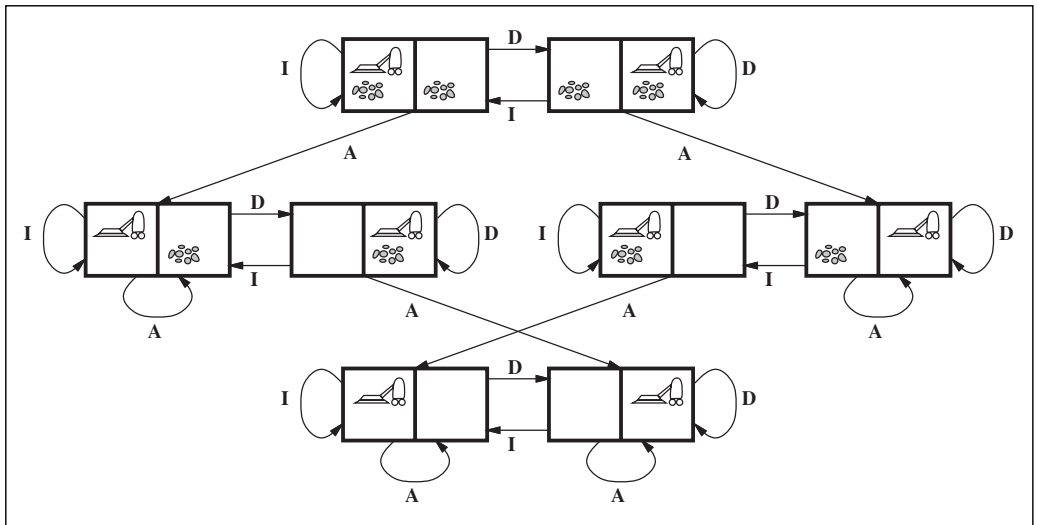

Figura 3.3 Espacio de estados para el mundo de la aspiradora. Los arcos denotan las acciones: I = *Izquierda*, D = *Derecha*, A = *Aspirar*.

8-PUZZLE

El **8-puzle**, 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 puzle que evitan todos los detalles de manipulaciones físicas.

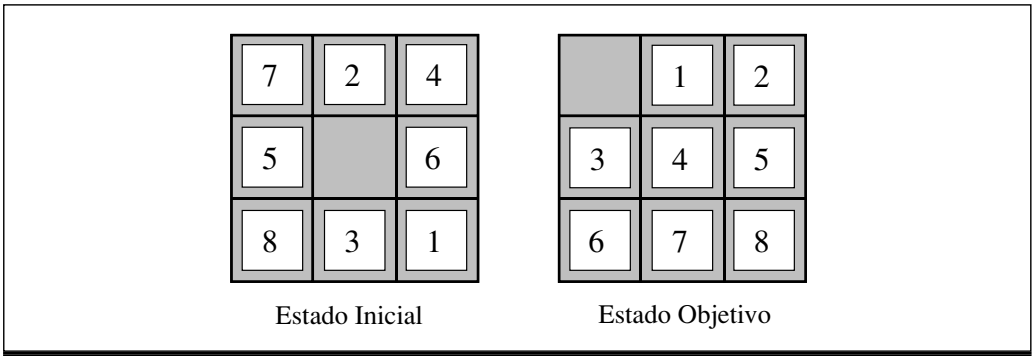

Figura 3.4 Un ejemplo típico del 8-puzle.

PIEZAS DESLIZANTES

El 8-puzle pertenece a la familia de puzles 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-puzle tiene $9!/2 = 181{,}440$ estados alcanzables y se resuelve fácilmente. El 15 puzle (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 puzle (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.

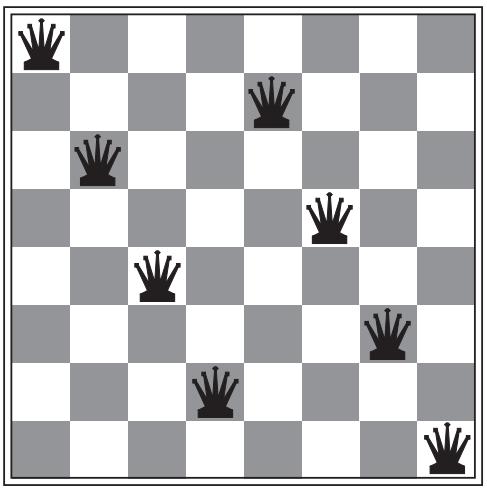

Figura 3.5 Casi una solución del problema de las 8-reinas. (La solución se deja como ejercicio.)

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:

FORMULACIÓN INCREMENTAL

FORMULACIÓN COMPLETA DE ESTADOS

- **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

BÚSQUEDA EN INTERNET

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.

3.3 Búsqueda de soluciones

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.

ÁRBOL DE BÚSQUEDA

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.

NODO DE BÚSQUEDA

EXPANDIR

GENERAR

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*(*RimnicuVilcea*). 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.

ESTRATEGIA DE BÚSQUEDA

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 los 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

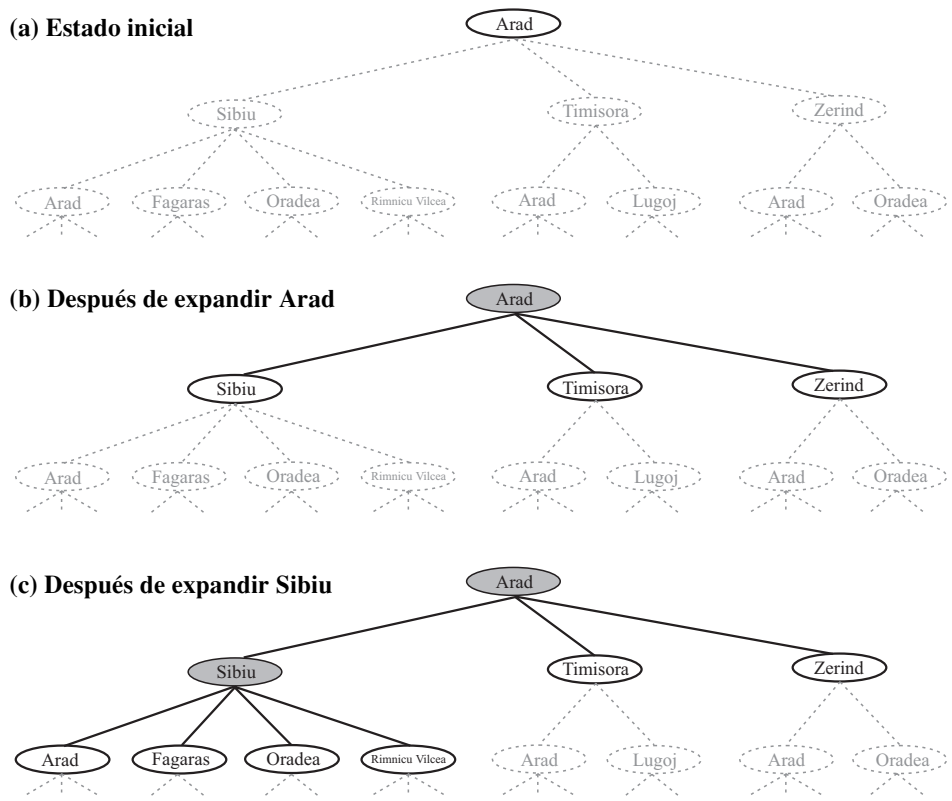

Figura 3.6 Árboles de búsqueda parciales para encontrar una ruta desde Arad hasta Bucarest. Se sombrean los nodos que se han expandido; los nodos que se han generado pero todavía no se han expandido se rodean en negrita; los nodos que todavía no se han generado se marcan con líneas discontinuas.

función BÚSQUEDA-ÁRBOLES(*problema,estrategia*) **devuelve** una solución o fallo
  inicializa 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.

FRONTERA

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

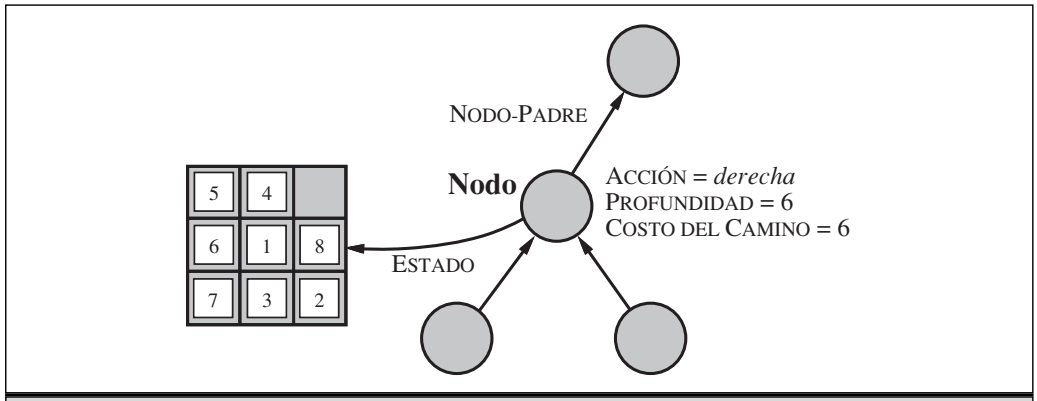

Figura 3.8 Los nodos son estructuras de datos a partir de los cuales se construye el árbol de búsqueda. Cada uno tiene un padre, un estado y varios campos. Las flechas señalan del hijo al padre.

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 quizá 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:

COLA

- HACER-COLA(*elemento*, …) crea una cola con el(los) 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

OPTIMIZACIÓN

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

```
función BÚSQUEDA-ÁRBOLES(problema,frontera) devuelve una solución o fallo
  frontera ← INSERTA(HACER-NODO(ESTADO-INICIAL[problema]), frontera)
  hacer bucle
    si VACIA?(frontera) entonces devolver fallo.
    nodo ← BORRAR-PRIMERO(frontera)
    si TEST-OBJETIVO[problema] aplicado al ESTADO[nodo] es cierto
      entonces devolver SOLUCIÓN(nodo)
    frontera ← INSERTAR-TODO(EXPANDIR(nodo,problema),frontera)

función EXPANDIR(nodo,problema) devuelve un conjunto de nodos
  sucesores ← conjunto vacío
  para cada (acción,resultado) en SUCESOR-FN[problema](ESTADO[nodo]) hacer
    s ← un nuevo NODO
    ESTADO[s] ← resultado
    NODO-PADRE[s] ← nodo
    ACCIÓN[s] ← acción
    COSTO-CAMINO[s] ← COSTO-CAMINO[nodo] + COSTO-INDIVIDUAL(nodo,acción,s)
    PROFUNDIDAD[s] ← 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

- **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.

FACTOR DE RAMIFICACIÓN

El tiempo a menudo se mide en términos de número de nodos generados[5] durante la búsqueda, y el espacio en términos de máximo número de nodos que se almacena en memoria.

COSTO DE LA BÚSQUEDA

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 quizá 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;

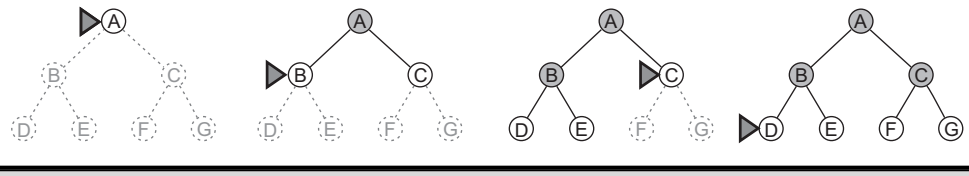

Figura 3.10 Búsqueda primero en anchura sobre un árbol binario sencillo. En cada etapa, el próximo nodo a expandir se indica con una marca.

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

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

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 profundi-

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

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.

BÚSQUEDA DE COSTO UNIFORME

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 $O(b^{\lceil C^*/\epsilon \rceil})$, 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 $b^{\lceil C^*/\epsilon \rceil}$ 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

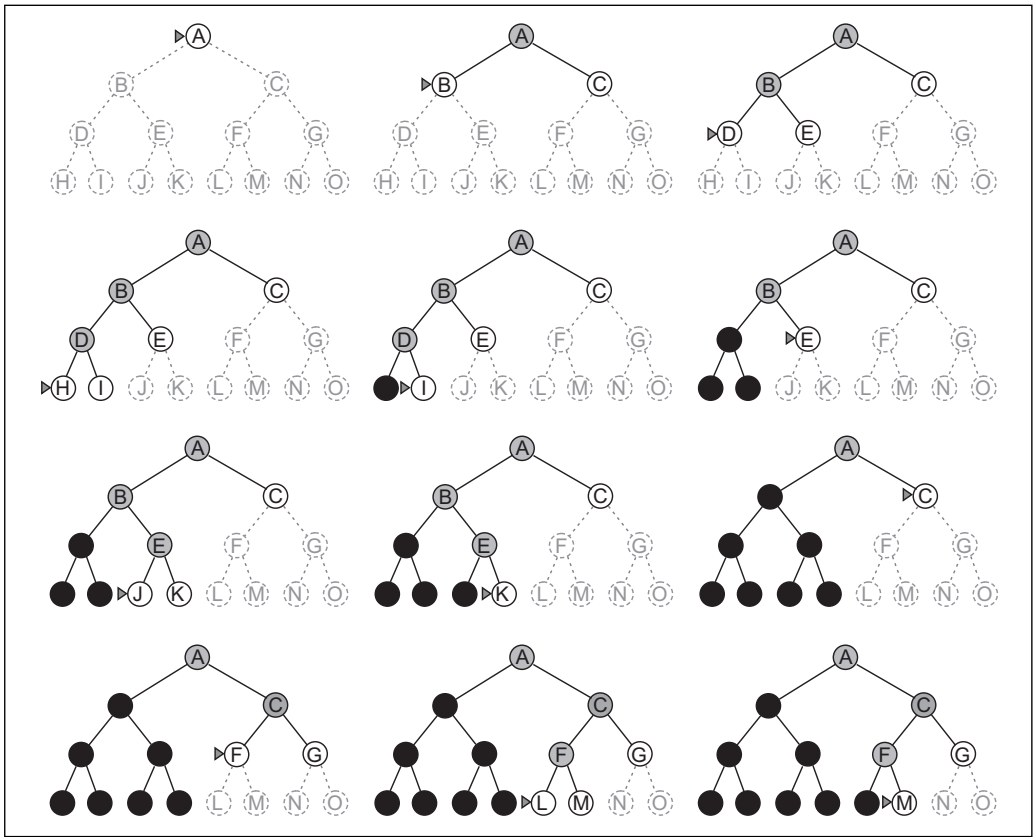

Figura 3.12 Búsqueda primero en profundidad sobre un árbol binario. Los nodos que se han expandido y no tienen descendientes en la frontera se pueden quitar de la memoria; estos se muestran en negro. Los nodos a profundidad 3 se suponen que no tienen sucesores y *M* es el nodo objetivo.

```
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 ← 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 ← BPL-RECURSIVO(sucesor,problema,límite)
    si resultado = corte entonces ocurrió un corte ← verdadero
    en otro caso si resultado ≠ 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 su 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.

BÚSQUEDA HACIA ATRÁS

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

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 $\ell = \infty$.

DIÁMETRO

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

BÚSQUEDA 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 ∞ 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

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

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

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

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,

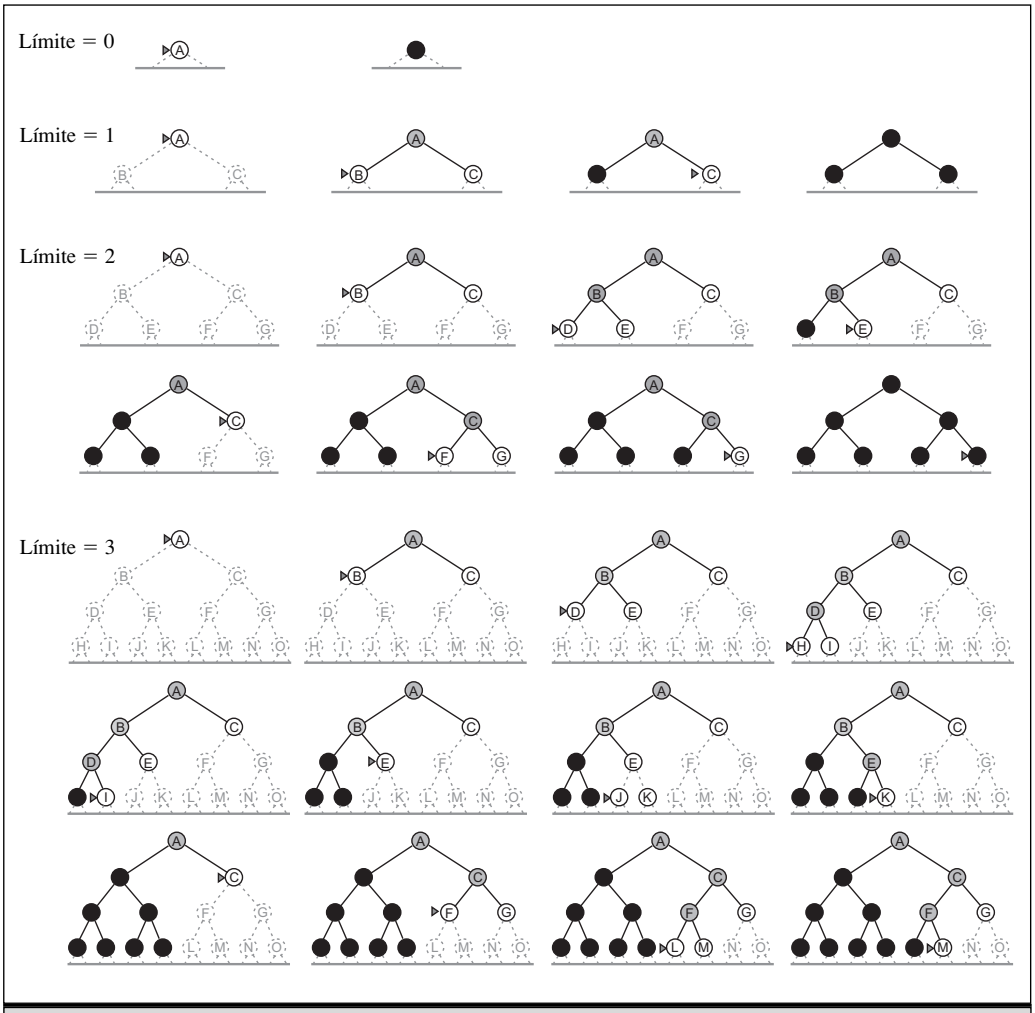

Figura 3.15 Cuatro iteraciones de la búsqueda de profundidad iterativa sobre un árbol binario.

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 comprueban antes de ser expandido si cada nodo está en la frontera del otro árbol de bús-

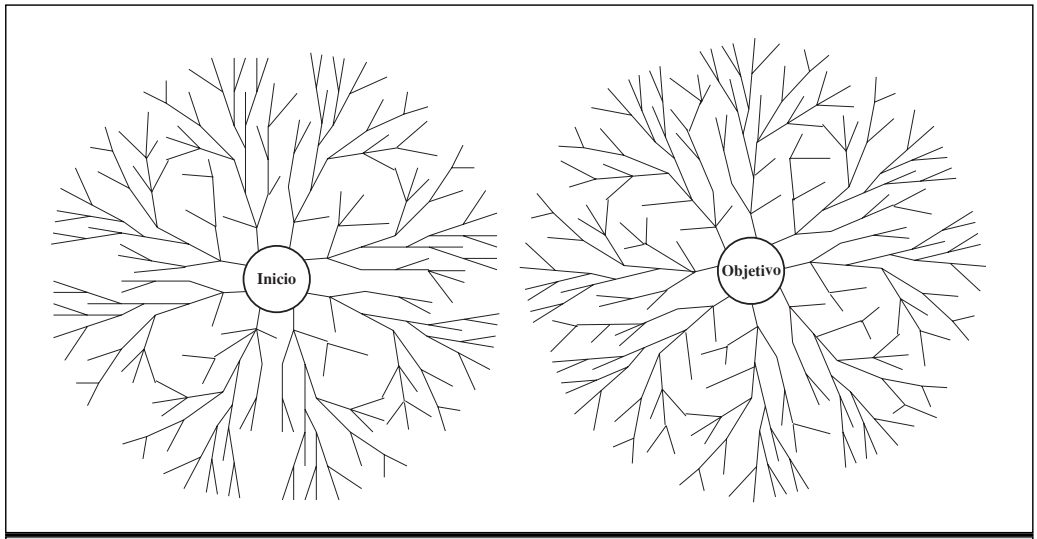

Figura 3.16 Un esquema de una búsqueda bidireccional que está a punto de tener éxito, cuando una rama desde el nodo inicio encuentra una rama desde el nodo objetivo.

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.

PREDECESORES

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-puzle 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	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Espacio	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
¿Optimal?	Sí[c]	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] optimal 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-

da de rutas y los puzles 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.

REJILLA RECTANGULAR

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 emparejamiento, 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*.

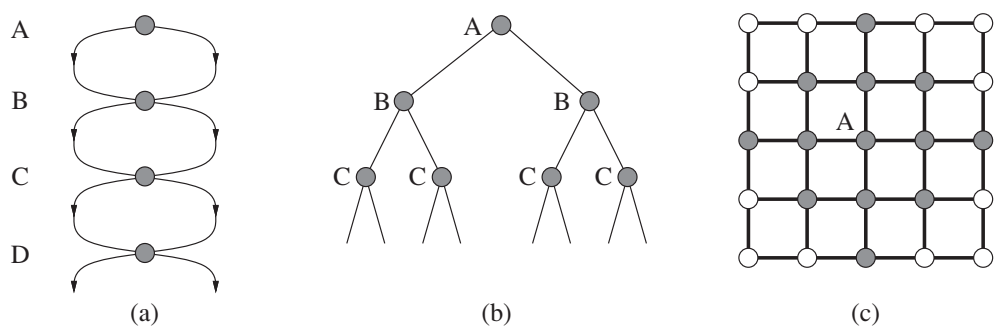

Figura 3.18 Espacios de estados que genera un árbol de búsqueda exponencialmente más grande. (a) Un espacio de estados en el cual hay dos acciones posibles que conducen de A a B, dos de B a C, etcétera. El espacio de estados contiene $d + 1$ estados, donde d es la profundidad máxima. (b) Correspondiente 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 ← conjunto vacío
  frontera ← INSERTAR(HACER-NODO(ESTADO-INICIAL[problema]), frontera)
  bucle hacer
    si VACIA?(frontera) entonces devolver fallo
    nodo ← 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 ← 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

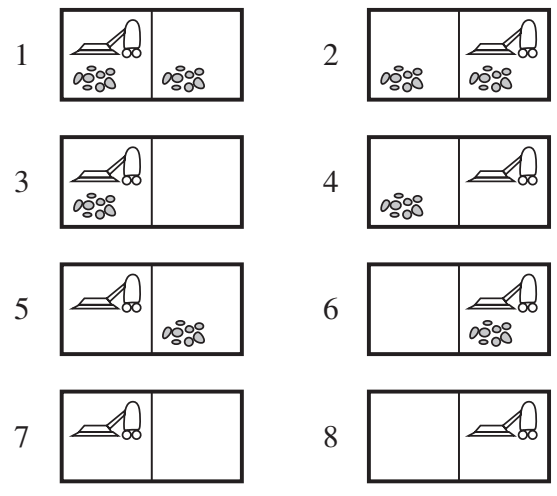

Figura 3.20 Los ocho posibles estados del mundo de la aspiradora.

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.)

COACCIÓN

ESTADO DE CREENCIA

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.

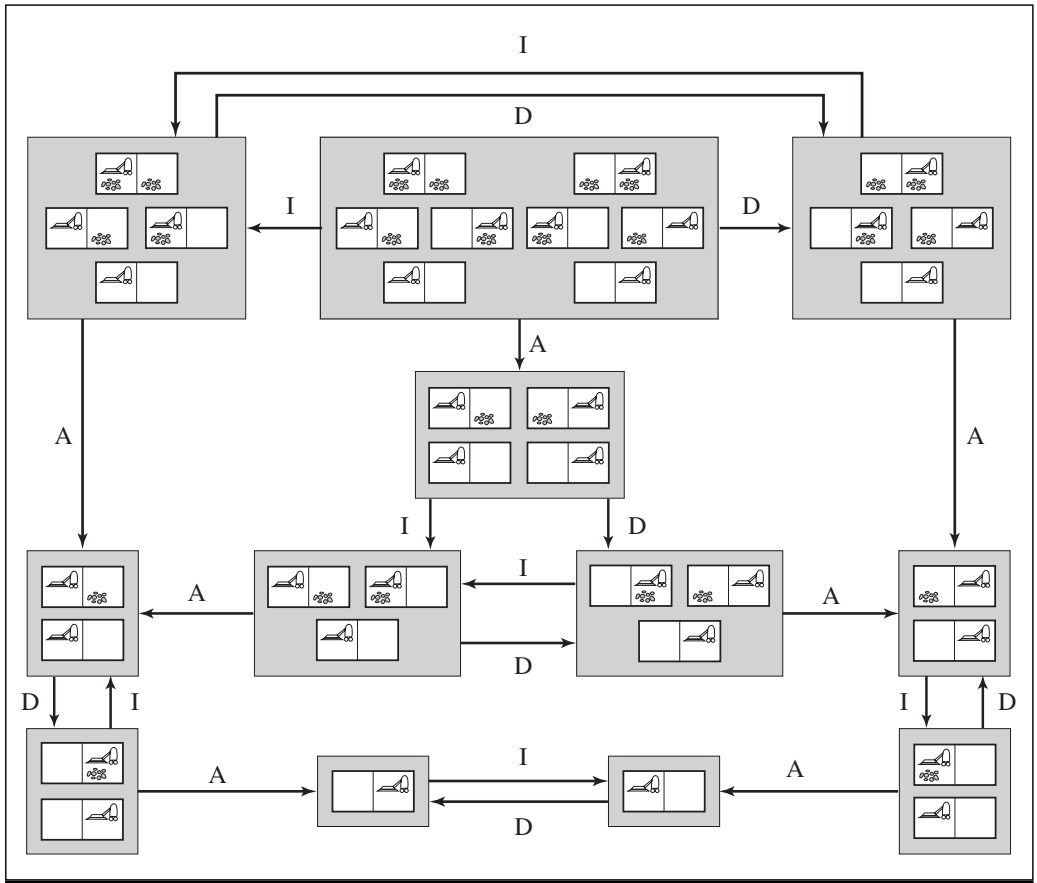

Figura 3.21 La parte accesible del espacio de estados de creencia para el mundo determinista de la aspiradora sin sensores. Cada caja sombreada corresponde a un estado de creencia simple. Desde cualquier punto, el agente está en un estado de creencia particular pero no sabe en qué estado físico está. 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

PROBLEMA 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 Newel (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-puzle es el primo más pequeño del 15-puzle, que fue inventado por el famoso americano diseñador de juegos Sam Loyd (1959) en la década de 1870. El 15-puzle 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-puzle 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-puzle). Un análisis exhaustivo del 8-puzle 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-puzle 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 eminente 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 *indecidible*. 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). Koening 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-puzle 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.