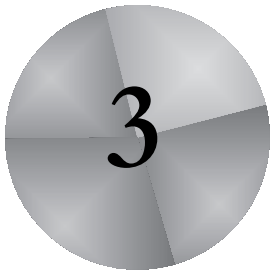


3 Resolver problemas mediante búsqueda

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.

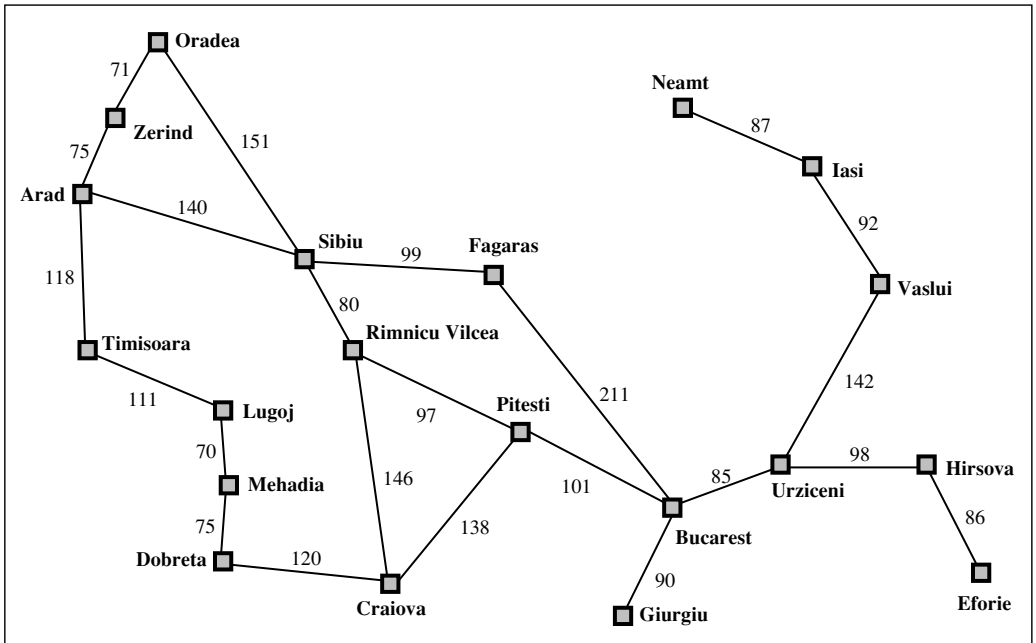


Figura 3.2 Un mapa de carreteras simplificado de parte de Rumanía.

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.

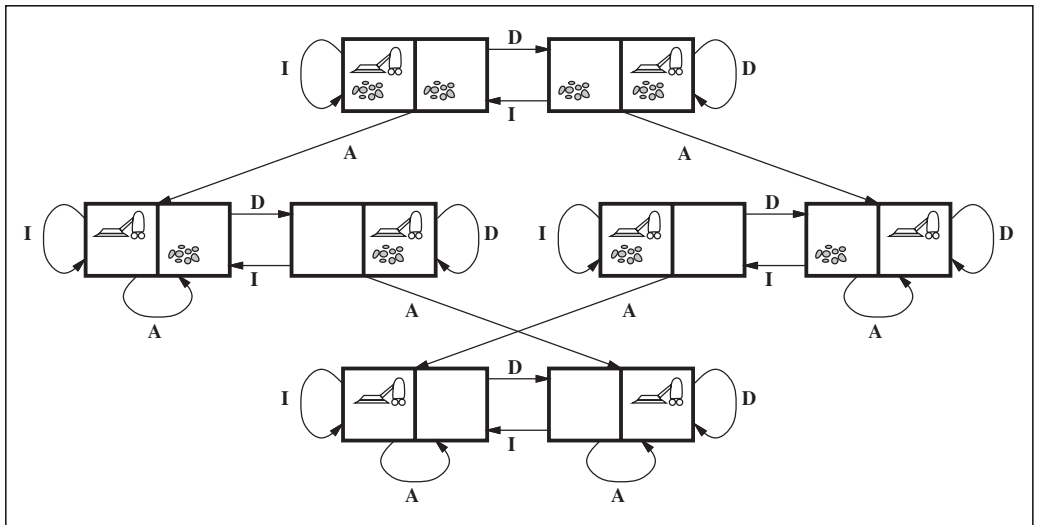


Figura 3.3 Espacio de estados para el mundo de la aspiradora. Los arcos denotan las acciones: I = *Izquierda*, D = *Derecha*, A = *Aspirar*.

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.

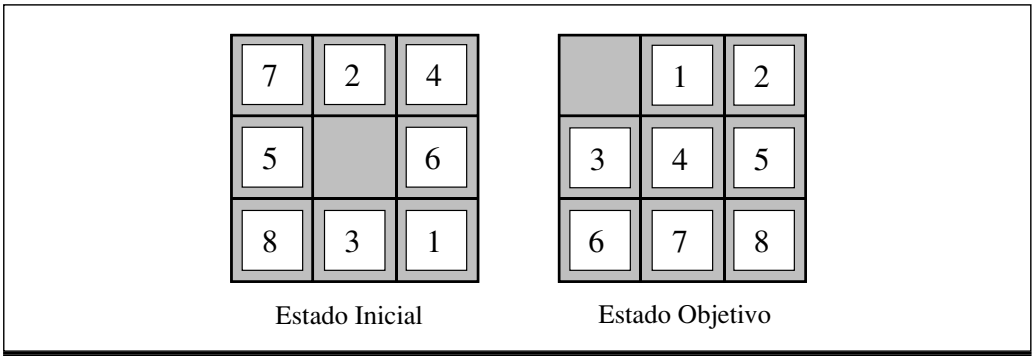


Figura 3.4 Un ejemplo típico del 8-puzle.

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.

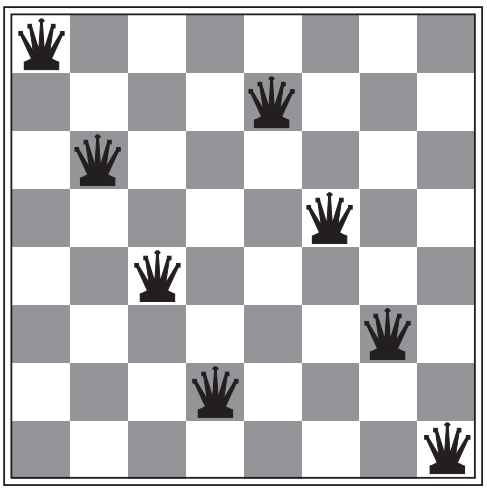


Figura 3.5 Casi una solución del problema de las 8-reinas. (La solución se deja como ejercicio.)

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

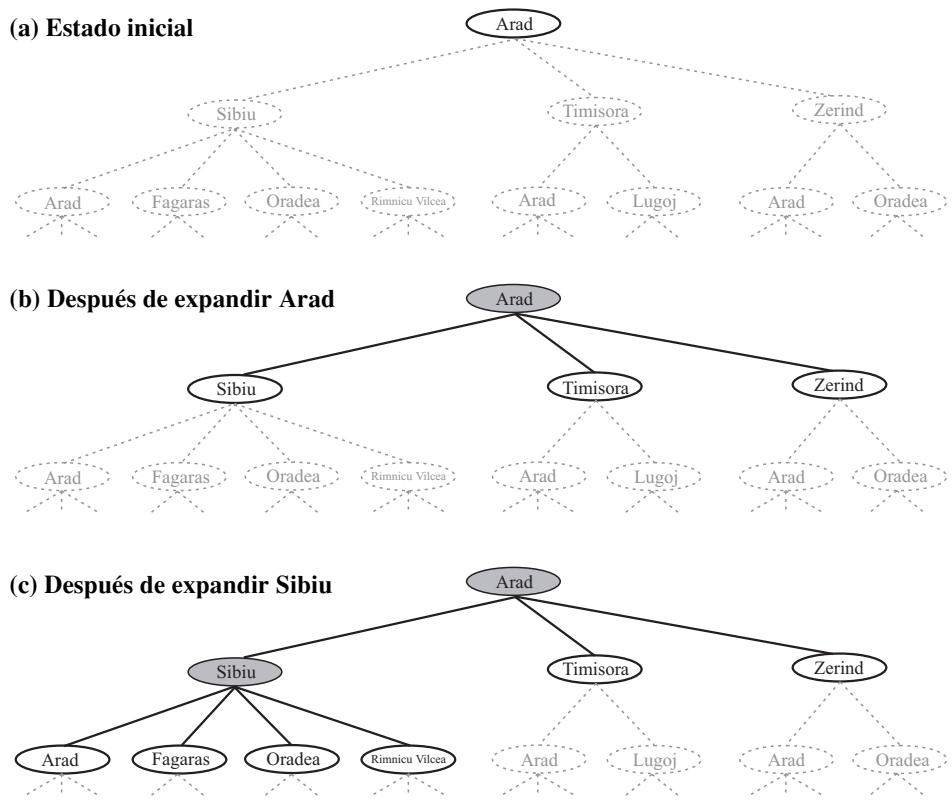


Figura 3.6 Árboles de búsqueda parciales para encontrar una ruta desde Arad hasta Bucarest. Se sombrean los nodos que se han expandido; los nodos que se han generado pero todavía no se han expandido se rodean en negrita; los nodos que todavía no se han generado se marcan con líneas discontinuas.

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

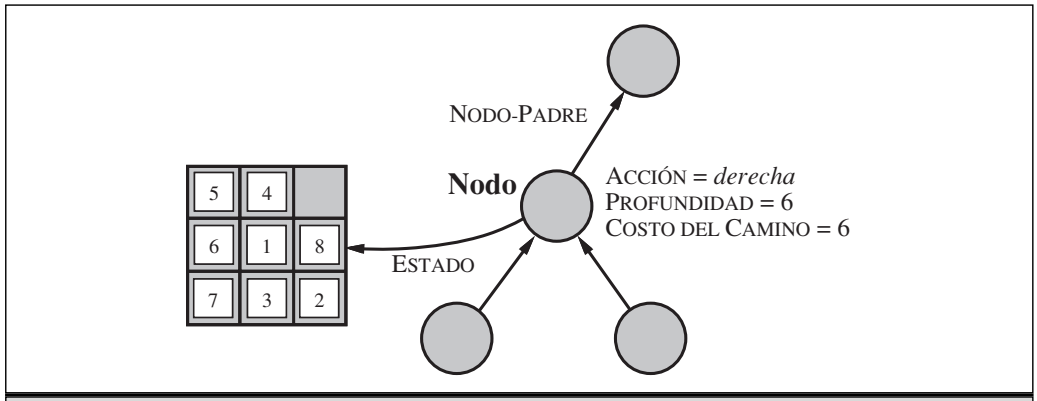


Figura 3.8 Los nodos son estructuras de datos a partir de los cuales se construye el árbol de búsqueda. Cada uno tiene un padre, un estado y varios campos. Las flechas señalan del hijo al padre.

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;

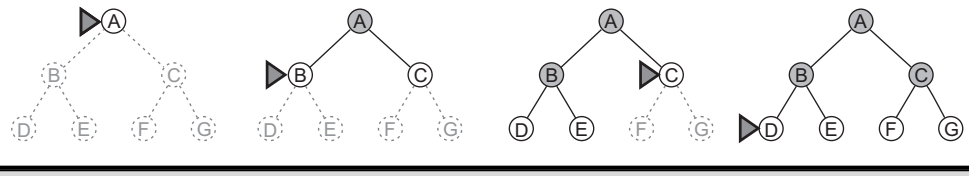


Figura 3.10 Búsqueda primero en anchura sobre un árbol binario sencillo. En cada etapa, el próximo nodo a expandir se indica con una marca.

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

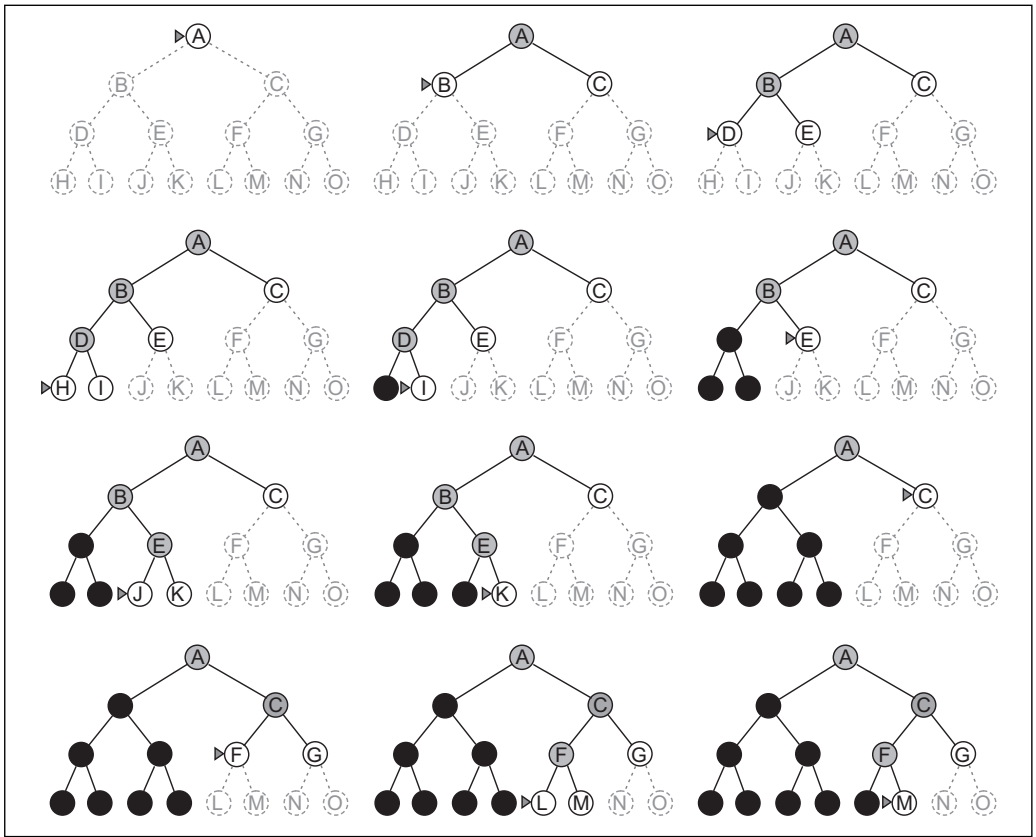


Figura 3.12 Búsqueda primero en profundidad sobre un árbol binario. Los nodos que se han expandido y no tienen descendientes en la frontera se pueden quitar de la memoria; estos se muestran en negro. Los nodos a profundidad 3 se suponen que no tienen sucesores y *M* es el nodo objetivo.

```
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,

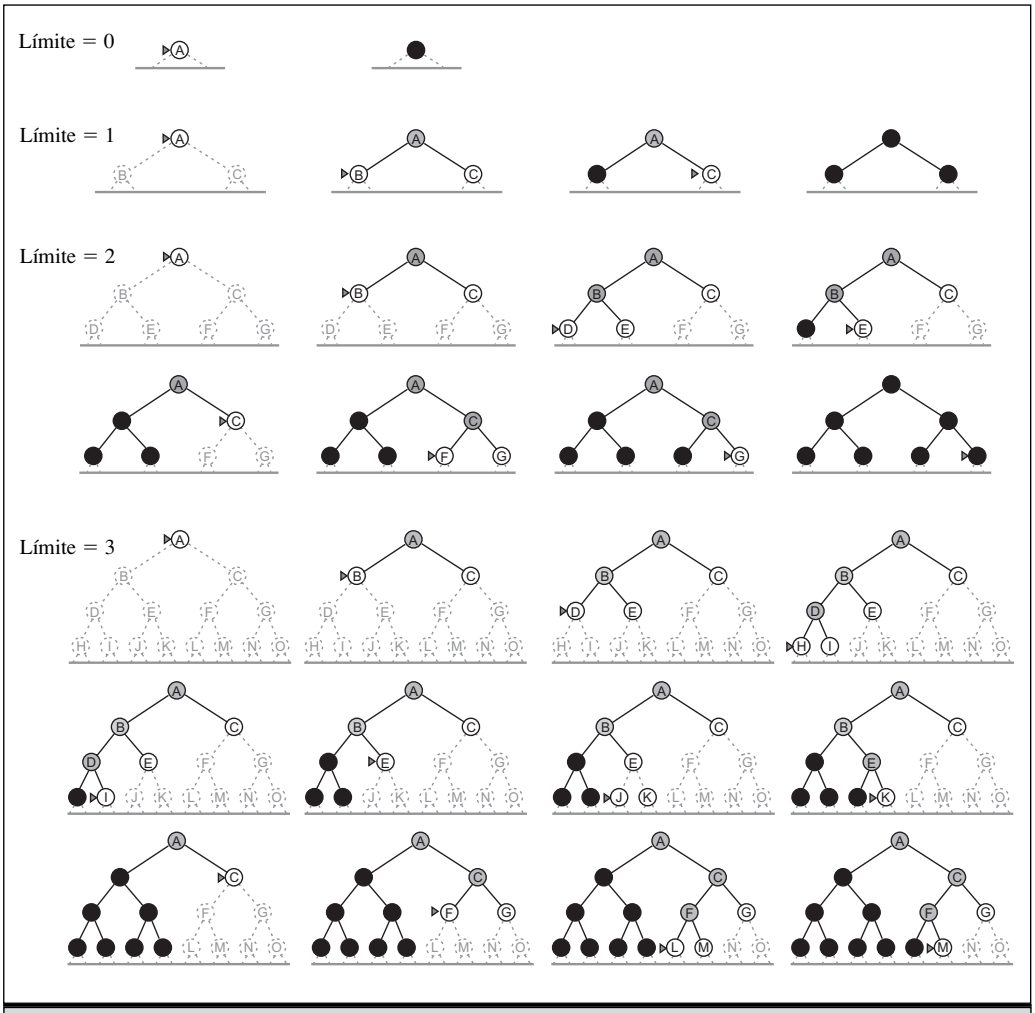


Figura 3.15 Cuatro iteraciones de la búsqueda de profundidad iterativa sobre un árbol binario.

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-

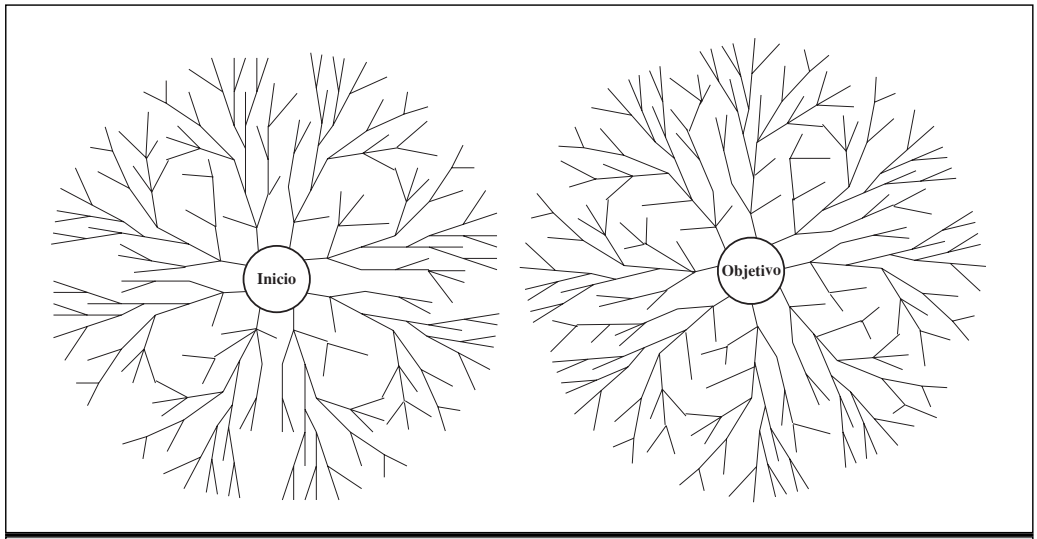


Figura 3.16 Un esquema de una búsqueda bidireccional que está a punto de tener éxito, cuando una rama desde el nodo inicio encuentra una rama desde el nodo objetivo.

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*.

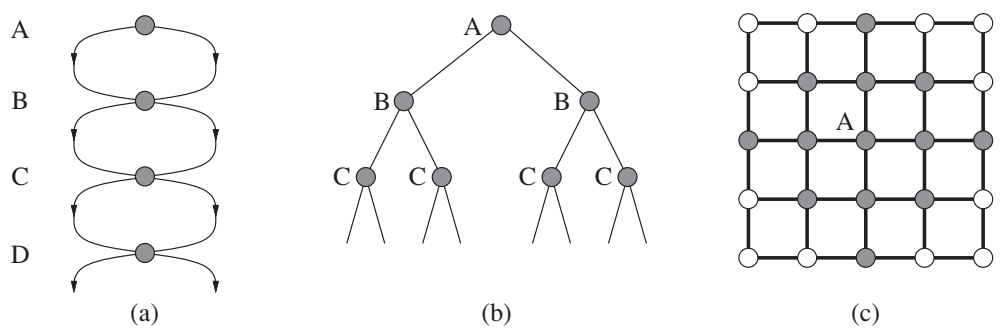


Figura 3.18 Espacios de estados que genera un árbol de búsqueda exponencialmente más grande. (a) Un espacio de estados en el cual hay dos acciones posibles que conducen de A a B, dos de B a C, etcétera. El espacio de estados contiene $d + 1$ estados, donde d es la profundidad máxima. (b) Correspondiente 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

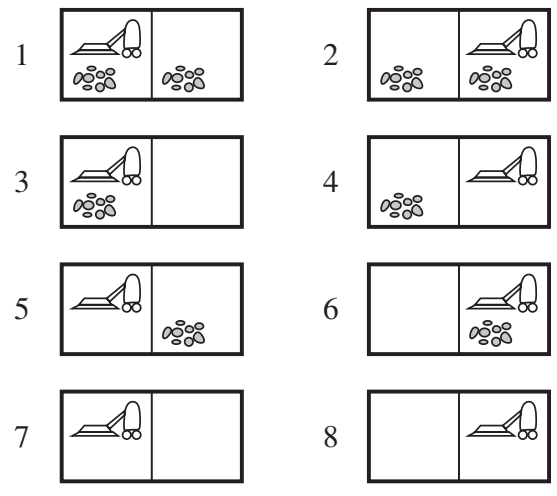


Figura 3.20 Los ocho posibles estados del mundo de la aspiradora.

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.

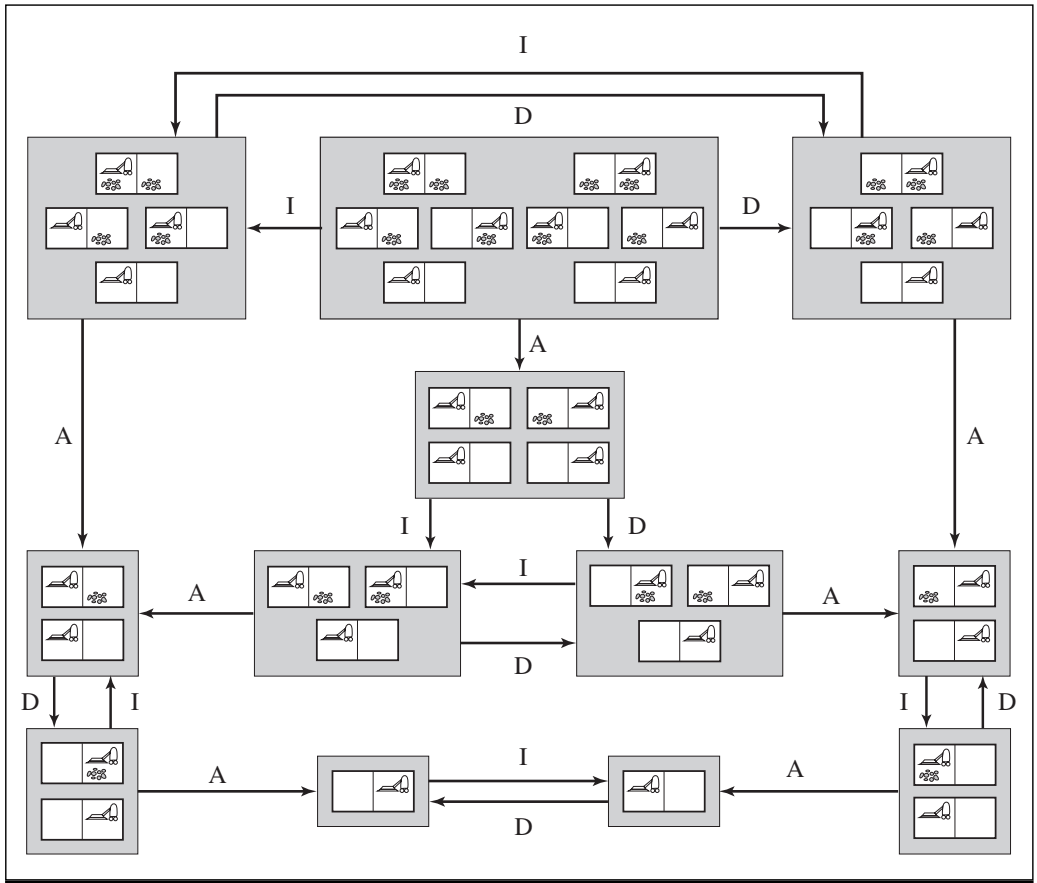


Figura 3.21 La parte accesible del espacio de estados de creencia para el mundo determinista de la aspiradora sin sensores. Cada caja sombreada corresponde a un estado de creencia simple. Desde cualquier punto, el agente está en un estado de creencia particular pero no sabe en qué estado físico está.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

3.5 Consideremos el problema de las 8-reinas usando la formulación «eficiente» incremental de la página 75. Explique por qué el tamaño del espacio de estados es al menos $\sqrt[3]{n!}$ y estime el valor más grande para n para el cual es factible la exploración exhaustiva. (*Consejo*: saque una cota inferior para el factor de ramificación considerando el número máximo de cuadrados que una reina puede atacar en cualquier columna.)

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

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

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

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

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

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

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

a) Formule el problema de forma precisa, haciendo sólo las distinciones necesarias para asegurar una solución válida. Dibujar un diagrama del espacio de estados completo.

b) Implemente y resuelva el problema de manera óptima utilizando un algoritmo apropiado de búsqueda. ¿Es una buena idea comprobar los estados repetidos?

c) ¿Por qué cree que la gente utiliza mucho tiempo para resolver este puzle, dado que el espacio de estados es tan simple?

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

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

a) Muestre que este algoritmo es óptimo para costos de camino generales.

b) Considere un árbol uniforme con factor de ramificación b, profundidad de solución d, y costos unidad. ¿Cuántas iteraciones requerirá la longitud iterativa?

c) Ahora considere los costos en el rango continuo [0,1] con un coste mínimo positivo ϵ. ¿Cuántas iteraciones requieren en el peor caso?

d) Implemente el algoritmo y aplíquelo a los casos de los problemas del 8-puzle y del viajante de comercio. Compare el funcionamiento del algoritmo con la búsqueda de costo uniforme, y comente sus resultados.

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

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

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

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

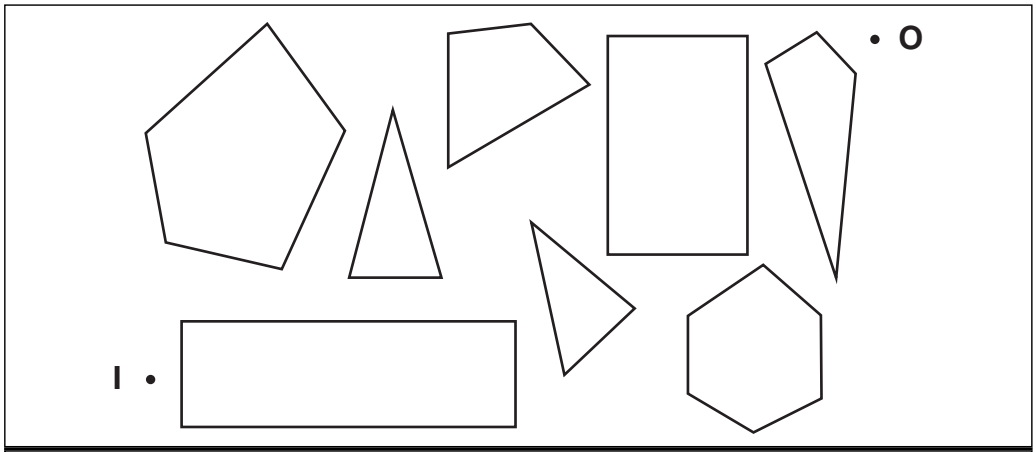


Figura 3.22 Una escena con obstáculos poligonales.

a) Suponga que el espacio de estados consiste en todas las posiciones (x, y) en el plano. ¿Cuántos estados hay? ¿Cuántos caminos hay al objetivo?

b) Explique brevemente por qué el camino más corto desde un vértice de un polígono a cualquier otro debe consistir en segmentos en línea recta que unen algunos vértices de los polígonos. Defina un espacio de estados bueno. ¿Cómo de grande es este espacio de estados?

c) Defina las funciones necesarias para implementar el problema de búsqueda, incluyendo una función sucesor que toma un vértice como entrada y devuelve el conjunto de vértices que pueden alcanzarse en línea recta desde el vértice dado. (No olvide los vecinos sobre el mismo polígono.) Utilice la distancia en línea recta para la función heurística.

d) Aplique uno o varios de los algoritmos de este capítulo para resolver un conjunto de problemas en el dominio, y comente su funcionamiento.

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

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

a) Implemente este entorno y un agente de resolución de problemas. El agente tendrá que formular un nuevo problema después de la colocación en otro lugar, que implicará el descubrimiento de su posición actual.

b) Documente el funcionamiento de su agente (teniendo que generar su agente el comentario conveniente de cómo se mueve a su alrededor) y el informe de su funcionamiento después de más de 100 episodios.

c) Modifique el entorno de modo que el 30 por ciento de veces el agente termine en un destino no planeado (escogido al azar de otros vértices visibles, o que no haya ningún movimiento). Esto es un modelo ordinario de los errores de movimiento de un robot real. Modifique al agente de modo que cuando se descubra tal error, averigüe dónde está y luego construya un plan para regresar donde estaba y resumir el viejo plan. ¡Recuerde que a veces recuperarse hacia donde estaba también podría fallar! Muestre un ejemplo satisfactorio del agente de modo que después de dos errores de movimientos sucesivos, todavía alcance el objetivo.

d) Ahora intente dos esquemas de recuperación diferentes después de un error: (1) diríjase al vértice más cercano sobre la ruta original; y (2) plantee una nueva ruta al objetivo desde la nueva posición. Compare el funcionamiento de estos tres esquemas de recuperación. ¿La inclusión de costos de la búsqueda afecta la comparación?

e) Ahora suponga que hay posiciones de las cuales la vista es idéntica (por ejemplo, suponga que el mundo es una rejilla con obstáculos cuadrados). ¿Qué tipo de problema afronta ahora el agente? ¿A qué se parecen las soluciones?

3.17 En la página 71, dijimos que no consideraríamos problemas con caminos de costos negativos. En este ejercicio, lo exploramos con más profundidad.

a) Suponga que las acciones pueden tener costos negativos arbitrariamente grandes; explique por qué esta posibilidad forzaría a cualquier algoritmo óptimo a explorar entero el espacio de estados.

b) ¿Ayuda si insistimos que los costos puedan ser mayores o iguales a alguna constante negativa c? Considere tanto árboles como grafos.

c) Suponga que hay un conjunto de operadores que forman un ciclo, de modo que la ejecución del conjunto en algún orden no cause ningún cambio neto al estado. Si todos estos operadores tienen el coste negativo, ¿qué implica sobre el comportamiento óptimo para un agente en tal entorno?

d) Fácilmente pueden imaginarse operadores con alto coste negativo, incluso sobre dominios como la búsqueda de rutas. Por ejemplo, algunos caminos podrían tener un paisaje hermoso cuanto más lejano esté y pese más que los gastos normales en términos de tiempo y combustible. Explique, en términos exactos, dentro del contexto del espacio de estados de búsqueda, por qué la gente no conduce sobre ciclos pintorescos indefinidamente, y explique cómo definir el espacio de estados y operadores para la búsqueda de rutas de modo que agentes artificiales también puedan evitar la formación de ciclos.

e) ¿Puede usted pensar en un dominio real en el cual los costos son la causa de la formación de ciclos?

3.18 Considere el mundo de la aspiradora de dos posiciones sin sensores conforme a la ley de Murphy. Dibuje el espacio de estados de creencia accesible desde el estado de creencia inicial $\{1, 2, 3, 4, 5, 6, 7, 8\}$, y explique por qué el problema es irresoluble. Mues-

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

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

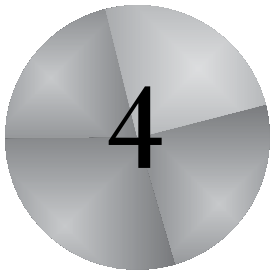
a) ¿Cuál de los algoritmos definidos en este capítulo parece apropiado para este problema? ¿Debería el algoritmo comprobar los estados repetidos?

b) Aplique el algoritmo escogido para obtener una secuencia de acciones óptima para un mundo de 3×3 cuyo estado inicial tiene la suciedad en los tres cuadrados superiores y el agente está en el centro.

c) Construya un agente de búsqueda para el mundo de la aspiradora, y evalúe su rendimiento en un conjunto de 3×3 con probabilidad de 0,2 de suciedad en cada cuadrado.

d) Compare su mejor agente de búsqueda con un agente reactivo sencillo aleatorio que aspira si hay suciedad y en otro caso se mueve aleatoriamente.

e) Considere lo que sucedería si el mundo se ampliase a $n \times n$. ¿Cómo se modificaría el rendimiento del agente de búsqueda y del agente reactivo variando el n?



4 Búsqueda informada y exploración

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

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

4.1 Estrategias de búsqueda informada (heurísticas)

BÚSQUEDA INFORMADA

Esta sección muestra cómo una estrategia de **búsqueda informada** (la que utiliza el conocimiento específico del problema más allá de la definición del problema en sí mismo) puede encontrar soluciones de una manera más eficiente que una estrategia no informada.

BÚSQUEDA PRIMERO EL MEJOR

FUNCIÓN DE EVALUACIÓN

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

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

FUNCIÓN HEURÍSTICA

Hay una familia entera de algoritmos de BÚSQUEDA-PRIMERO-MEJOR con funciones[1] de evaluación diferentes. Una componente clave de estos algoritmos es una **función heurística**[2], denotada $h(n)$:

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

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

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

Búsqueda voraz primero el mejor

BÚSQUEDA VORAZ PRIMERO EL MEJOR

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

DISTANCIA EN LÍNEA RECTA

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

[1] El Ejercicio 4.3 le pide demostrar que esta familia incluye varios algoritmos familiares no informados.
[2] Una función heurística $h(n)$ toma un *nodo* como entrada, pero depende sólo del *estado* en ese nodo.
[3] Nuestra primera edición la llamó **búsqueda avara (voraz)**; otros autores la han llamado **búsqueda primero el mejor**. Nuestro uso más general del término se sigue de Pearl (1984).

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

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

Figura 4.1 Valores de h_{DLR}. Distancias en línea recta a Bucarest.

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

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

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

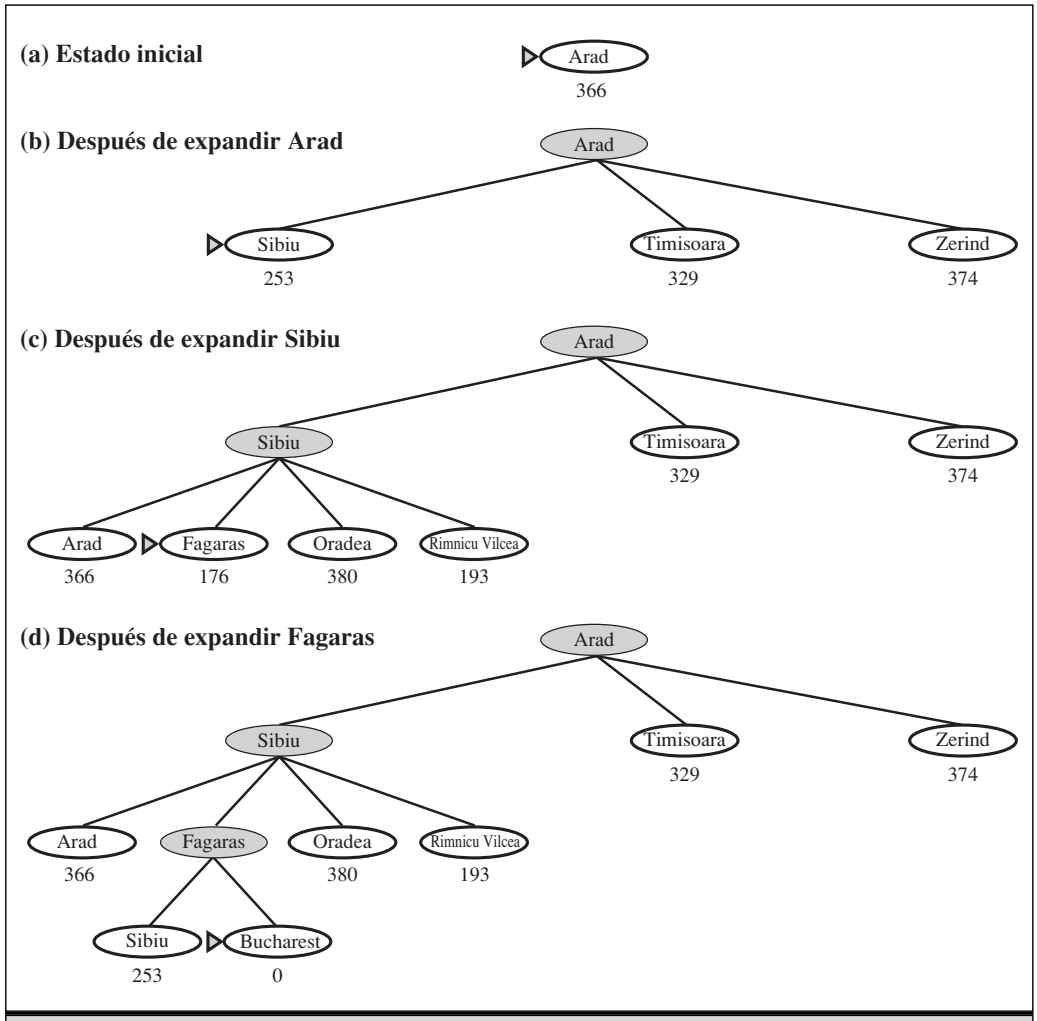


Figura 4.2 Etapas en una búsqueda primero el mejor avara para Bucarest utilizando la heurística distancia en línea recta h_{DLR}. Etiquetamos lo nodos con sus h-valores.

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

BÚSQUEDA A*

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

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

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

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

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

HEURÍSTICA ADMISIBLE

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

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

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

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

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

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

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

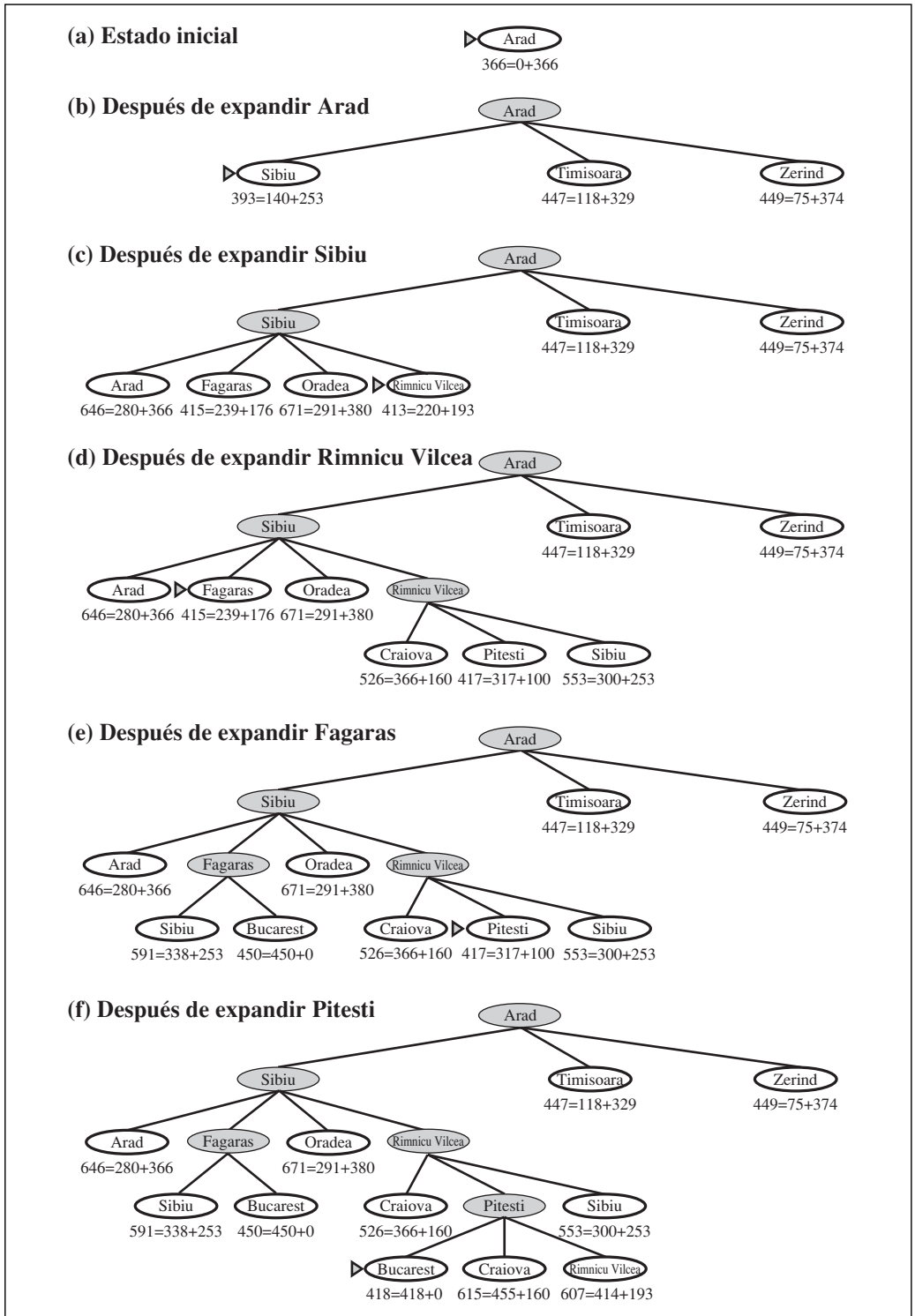


Figura 4.3 Etapas en una búsqueda A* para Bucarest. Etiquetamos lo nodos con $f = g + h$. Los valores h son las distancias en línea recta a Bucarest tomadas de la Figura 4.1.

CONSISTENCIA

MONOTONÍA

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

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

DESIGUALDAD TRIANGULAR

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

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

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

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

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

CURVAS DE NIVEL

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

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

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

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

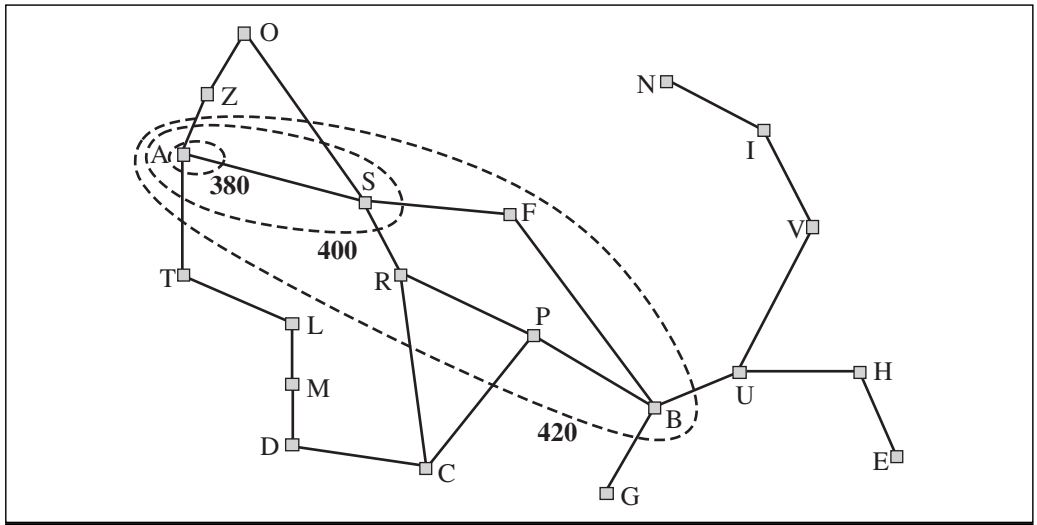


Figura 4.4 Mapa de Rumanía mostrando curvas de nivel en $f = 380$, $f = 400$ y $f = 420$, con Arad como estado inicial. Los nodos dentro de cada curva de nivel tienen f-costos menores o iguales al valor de la curva de nivel.

el g-coste más alto (porque todos los nodos de objetivo tienen $h(n) = 0$). Por intuición, es también obvio que la búsqueda A* es completa. Como añadimos las bandas de f creciente, al final debemos alcanzar una banda donde f sea igual al coste del camino a un estado objetivo[4].

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

PODA

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

ÓPTIMAMENTE EFICIENTE

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

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

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

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

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

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

Búsqueda heurística con memoria acotada

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

BÚSQUEDA RECURSIVA DEL PRIMERO MEJOR

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

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

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

Figura 4.5 Algoritmo para la búsqueda primero el mejor recursiva.

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

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

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

A*M

A*MS

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

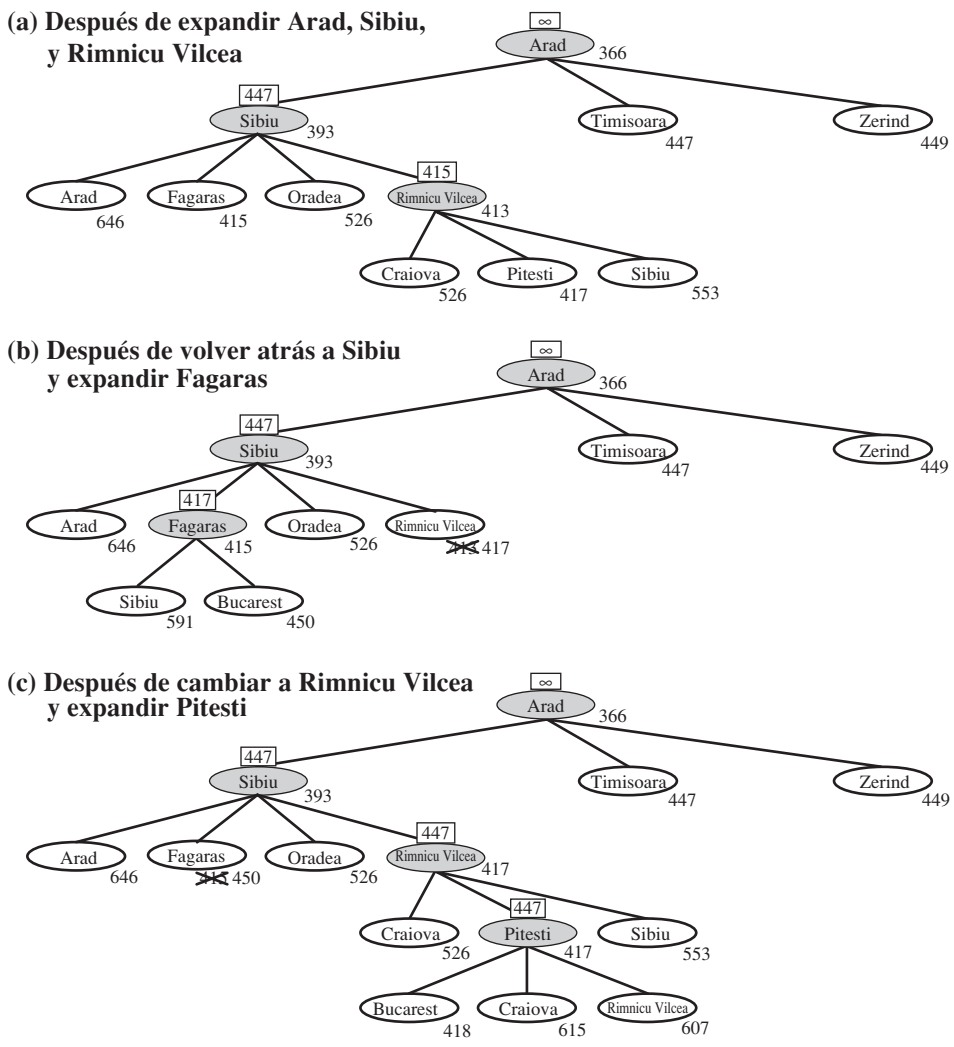


Figura 4.6 Etapas en una búsqueda BRPM para la ruta más corta a Bucarest. Se muestra el valor del *f*-límite para cada llamada recursiva sobre cada nodo actual. (a) Se sigue el camino vía Rimnicu Vilcea hasta que la mejor hoja actual (Pitesti) tenga un valor que es peor que el mejor camino alternativo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

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

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

El algoritmo completo es demasiado complicado para reproducirse aquí[5], pero hay un matiz digno de mencionar. Dijimos que A*MS expande la mejor hoja y suprime la peor hoja. ¿Y si *todos* los nodos hoja tienen el mismo *f*-valor? Entonces el algoritmo podría seleccionar el mismo nodo para eliminar y expandir. A*MS soluciona este problema expandiendo la mejor hoja *más nueva* y suprimiendo la peor hoja *más vieja*. Estos pueden ser el mismo nodo sólo si hay una sola hoja; en ese caso, el árbol actual de búsqueda debe ser un camino sólo desde la raíz a la hoja llenando toda la memoria. Si la hoja no es un nodo objetivo, entonces, *incluso si está sobre un camino solución óptimo*, esa solución no es accesible con la memoria disponible. Por lo tanto, el nodo puede descartarse como si no tuviera ningún sucesor.

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

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

THRASHING

Aprender a buscar mejor

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

ESPACIO DE ESTADOS METANIVEL

ESPACIO DE ESTADOS A NIVEL DE OBJETO

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

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

APRENDIZAJE METANIVEL

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

4.2 Funciones heurísticas

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

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

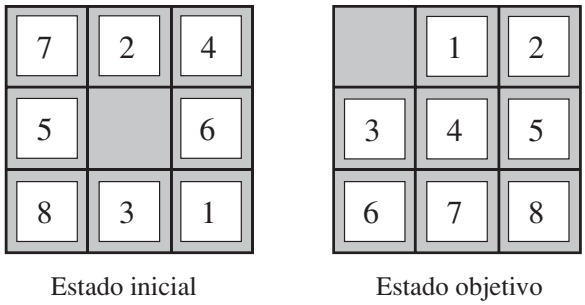


Figura 4.7 Un caso típico del 8-puzle. La solución tiene 26 pasos.

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

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

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

DISTANCIA DE MANHATTAN

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

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

El efecto de la precisión heurística en el rendimiento

FACTOR DE RAMIFICACIÓN EFICAZ

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

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

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

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

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

DOMINACIÓN

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

	Costo de la búsqueda			Factor de ramificación eficaz		
d	BPI	A*(h_1)	A*(h_2)	BPI	A*(h_1)	A*(h_2)
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	3644035	227	73	2,78	1,42	1,24
14	—	539	113	—	1,44	1,23
16	—	1301	211	—	1,45	1,25
18	—	3056	363	—	1,46	1,26
20	—	7276	676	—	1,47	1,27
22	—	18094	1219	—	1,48	1,28
24	—	39135	1641	—	1,48	1,26

Figura 4.8 Comparación de los costos de la búsqueda y factores de ramificación eficaces para la BÚSQUEDA-PROFUNDIDAD-ITERATIVA y los algoritmos A* con h_1 y h_2. Los datos son la media de 100 ejemplos del puzle-8, para soluciones de varias longitudes.

Inventar funciones heurísticas admisibles

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

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

PROBLEMA RELAJADO

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

Si la definición de un problema está escrita en un lenguaje formal, es posible construir problemas relajados automáticamente[6]. Por ejemplo, si las acciones del 8-puzle están descritas como

> Una ficha puede moverse del cuadrado A al cuadrado B si
> A es horizontalmente o verticalmente adyacente a B **y** B es la vacía

podemos generar tres problemas relajados quitando una o ambas condiciones:

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

De (a), podemos obtener h_2 (distancia de Manhattan). El razonamiento es que h_2 sería el resultado apropiado si moviéramos cada ficha en dirección a su destino. La heurística obtenida de (b) se discute en el Ejercicio 4.9. De (c), podemos obtener h_1 (fichas mal colocadas), porque sería el resultado apropiado si las fichas pudieran moverse a su destino en un paso. Notemos que es crucial que los problemas relajados generados por esta técnica puedan resolverse esencialmente sin búsqueda, porque las reglas relajadas permiten que el problema sea descompuesto en ocho subproblemas independientes. Si el problema relajado es difícil de resolver, entonces los valores de la correspondencia heurística serán costosos de obtener[7].

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

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

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

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

---

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

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

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

SUBPROBLEMA

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

MODELO DE BASES DE DATOS

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

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

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

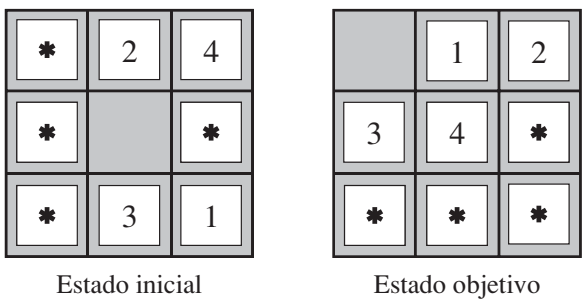


Figura 4.9 Un subproblema del puzle-8 dado en la Figura 4.7. La tarea es conseguir las fichas 1, 2, 3 y 4 en sus posiciones correctas, sin preocuparse de lo que le pasan a las otras fichas.

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

MODELO DE BASES DE DATOS DISJUNTAS

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

Aprendizaje de heurísticas desde la experiencia

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

CARACTERÍSTICAS

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

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

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

4.3 Algoritmos de búsqueda local y problemas de optimización

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

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

BÚSQUEDA LOCAL

ESTADO ACTUAL

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

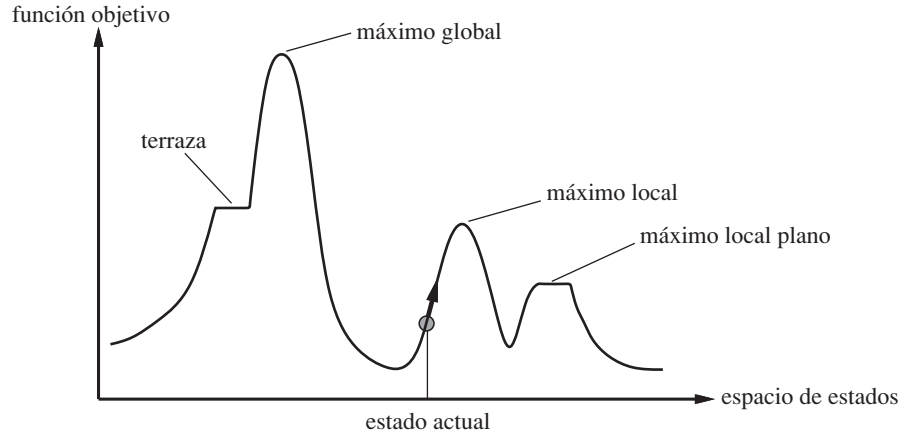


Figura 4.10 Un paisaje del espacio de estados unidimensional en el cual la elevación corresponde a la función objetivo. El objetivo es encontrar el máximo global. La búsqueda de ascensión de colinas modifica el estado actual para tratar de mejorarlo, como se muestra con la flecha. Se definen en el texto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

PROBLEMAS DE OPTIMIZACIÓN

FUNCIÓN OBJETIVO

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

PAISAJE DEL ESPACIO DE ESTADOS

MÍNIMO GLOBAL

MÁXIMO GLOBAL

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

Búsqueda de ascensión de colinas

ASCENSIÓN DE COLINAS

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

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

```
función ASCENSIÓN-COLINAS(problema) devuelve un estado que es un máximo local
  entradas: problema, un problema
  variables locales: actual, un nodo
                     vecino, un nodo

  actual ← HACER-NODO(ESTADO-INICIAL[problema])
  bucle hacer
    vecino ← sucesor de valor más alto de actual
    si VALOR[vecino] ≤ VALOR[actual] entonces devolver ESTADO[actual]
    actual ← vecino
```

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

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

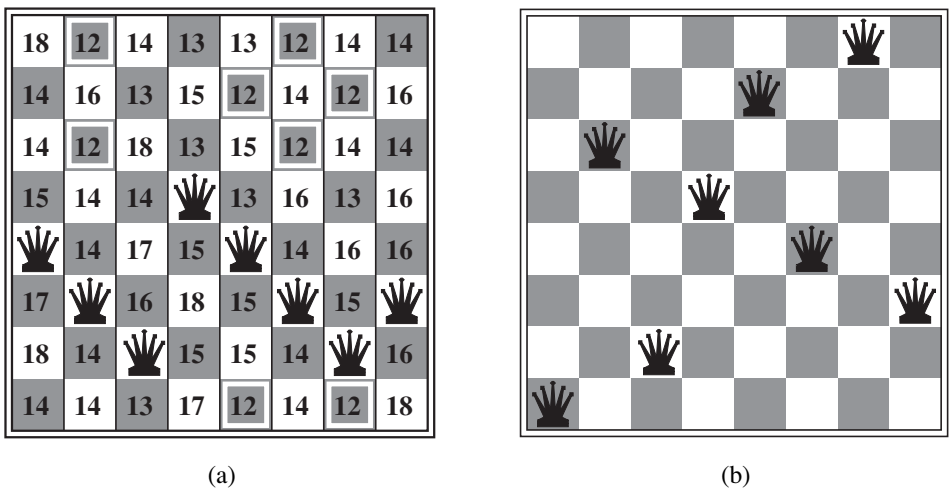


Figura 4.12 (a) Un estado de 8-reinas con una heurística de estimación de costos $h = 17$, mostrando al valor de h para cada sucesor posible obtenido al mover una reina dentro de su columna. Los mejores movimientos están marcados. (b) Un mínimo local en el espacio de estados de las 8-reinas; el estado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

BÚSQUEDA LOCAL VORAZ

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

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

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

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

TERRAZA

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

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

MOVIMIENTO LATERAL

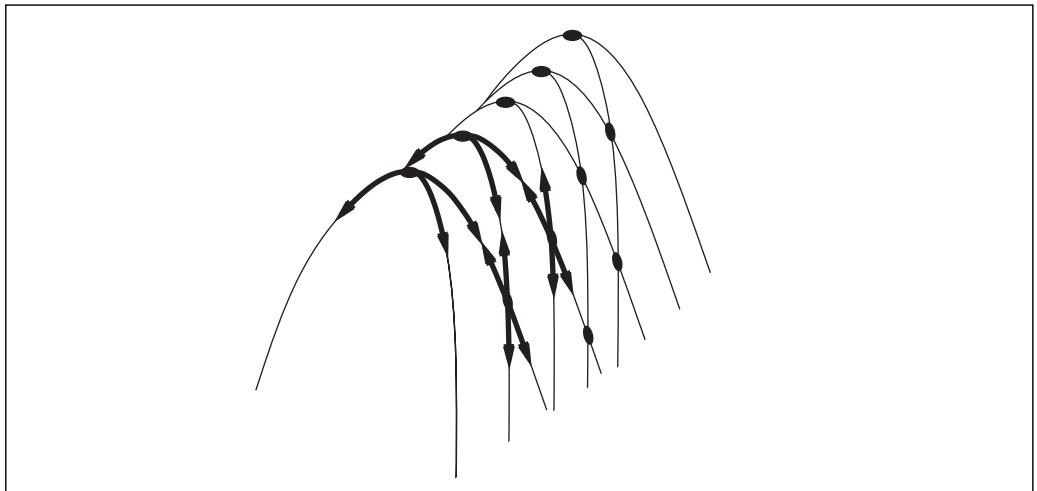


Figura 4.13 Ilustración de por qué las crestas causan dificultades para la ascensión de colinas. La rejilla de estados (círculos oscuros) se pone sobre una cresta que se eleva de izquierda a derecha y crea una secuencia de máximos locales que no están directamente relacionados el uno con el otro. De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

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

ASCENSIÓN DE COLINAS ESTOCÁSTICA

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

ASCENSIÓN DE COLINAS DE PRIMERA OPCIÓN

ASCENSIÓN DE COLINAS DE REINICIO ALEATORIO

Los algoritmos de ascensión de colinas descritos hasta ahora son incompletos, a menudo dejan de encontrar un objetivo, cuando éste existe, debido a que pueden estancarse sobre máximos locales. **La ascensión de colinas de reinicio aleatorio** adopta el refrán conocido, «si al principio usted no tiene éxito, intente, intente otra vez». Esto conduce a una serie de búsquedas en ascensión de colinas desde los estados iniciales generados aleatoriamente[8], parándose cuando se encuentra un objetivo. Es completa con probabilidad acercándose a 1, por la razón trivial de que generará finalmente un estado objetivo como el estado inicial. Si cada búsqueda por ascensión de colinas tiene una probabilidad p de éxito, entonces el número esperado de reinicios requerido es $1/p$. Para ejemplos de ocho reinas sin permitir movimientos laterales, $p \approx 0{,}14$, entonces necesitamos aproximadamente siete iteraciones para encontrar un objetivo (seis fracasos y un éxito). El número esperado de pasos es el coste de una iteración acertada más $(1 - p)/p$ veces el coste de fracaso, o aproximadamente 22 pasos. Cuando permitimos movimientos laterales, son necesarios $1/0{,}94 \approx 1{,}06$ iteraciones por regla general y $(1 \times 21) + (0{,}06/0{,}94) \times 64 \approx 25$ pasos. Para las ocho reinas, entonces, la ascensión de colinas de reinicio aleatorio es muy eficaz. Incluso para tres millones de reinas, la aproximación puede encontrar soluciones en menos de un minuto[9].

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

Búsqueda de temple simulado

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

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

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

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

TEMPLE SIMULADO

GRADIENTE DESCENDENTE

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

```
función TEMPLE-SIMULADO(problema, esquema) devuelve un estado solución
  entradas: problema, un problema
            esquema, una aplicación desde el tiempo a «temperatura»
  variables locales: actual, un nodo
                     siguiente, un nodo
                     T, una «temperatura» controla la probabilidad de un paso hacia abajo

  actual ← HACER-NODO(ESTADO-INICIAL[problema])
  para t ← 1 a ∞ hacer
    T ← esquema[t]
    si T = 0 entonces devolver actual
    siguiente ← un sucesor seleccionado aleatoriamente de actual
    ΔE ← VALOR[siguiente] − VALOR[actual]
    si ΔE > 0 entonces actual ← siguiente
    en caso contrario actual ← siguiente sólo con probabilidad e^(ΔE/T)
```

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

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

Búsqueda por haz local

BÚSQUEDA POR HAZ LOCAL

Guardar solamente un nodo en la memoria podría parecer una reacción extrema para el problema de limitaciones de memoria. El algoritmo[10] de **búsqueda por haz local** guarda la pista de *k* estados (no sólo uno). Comienza con estados generados aleatoriamente. En cada paso, se generan todos los sucesores de los *k* estados. Si alguno es un objetivo, paramos el algoritmo. Por otra parte, se seleccionan los *k* mejores sucesores de la lista completa y repetimos.

A primera vista, una búsqueda por haz local con *k* estados podría parecerse a ejecutar *k* reinicios aleatorios en paralelo en vez de en secuencia. De hecho, los dos algoritmos son bastantes diferentes. En una búsqueda de reinicio aleatorio, cada proceso de búsqueda se ejecuta independientemente de los demás. *En una búsqueda por haz local, la información útil es pasada entre los k hilos paralelos de búsqueda*. Por ejemplo, si un estado genera varios sucesores buenos y los otros $k - 1$ estados generan sucesores malos, entonces el efecto es que el primer estado dice a los demás, «¡Venid aquí, la hierba es más verde!» El algoritmo rápidamente abandona las búsquedas infructuosas y mueve sus recursos a donde se hace la mayor parte del progreso.

En su forma más simple, la búsqueda de haz local puede sufrir una carencia de diversidad entre los *k* estados (se pueden concentrar rápidamente en una pequeña región del espacio de estados, haciendo de la búsqueda un poco más que una versión cara de la ascensión de colinas). Una variante llamada **búsqueda de haz estocástica**, análoga a la ascensión de colinas estocástica, ayuda a aliviar este problema. En vez de elegir los *k* mejores del conjunto de sucesores candidatos, la búsqueda de haz estocástica escoge a *k* sucesores aleatoriamente, con la probabilidad de elegir a un sucesor como una función creciente de su valor. La búsqueda de haz estocástica muestra algún parecido con el proceso de selección natural, por lo cual los «sucesores» (descendientes) de un «estado» (organismo) pueblan la siguiente generación según su «valor» (idoneidad o salud).

BÚSQUEDA DE HAZ ESTOCÁSTICA

Algoritmos genéticos

ALGORITMO GENÉTICO

Un **algoritmo genético** (o AG) es una variante de la búsqueda de haz estocástica en la que los estados sucesores se generan combinando *dos* estados padres, más que modificar un solo estado. La analogía a la selección natural es la misma que con la búsqueda de haz estocástica, excepto que ahora tratamos con reproducción sexual más que con la reproducción asexual.

[10] Búsqueda por haz local es una adaptación de la **búsqueda de haz**, que es un algoritmo basado en camino.

POBLACIÓN

INDIVIDUO

Como en la búsqueda de haz, los AGs comienzan con un conjunto de k estados generados aletaoriamente, llamados **población**. Cada estado, o **individuo**, está representado como una cadena sobre un alfabeto finito (el más común, una cadenas de 0s y 1s). Por ejemplo, un estado de las ocho reinas debe especificar las posiciones de las ocho reinas, cada una en una columna de ocho cuadrados, y se requieren $8 \times \log_2 8 = 24$ bits. O bien, el estado podría representarse como ocho dígitos, cada uno en el rango de uno a ocho (veremos más tarde que las dos codificaciones se comportan de forman diferente). La Figura 4.15(a) muestra una población de cuatro cadenas de ocho dígitos que representan estados de ocho reinas.

FUNCIÓN IDONEIDAD

En la Figura 4.15(b)-(e) se muestra la producción de la siguiente generación de estados. En (b) cada estado se tasa con la función de evaluación o (en terminología AG) la **función idoneidad**. Una función de idoneidad debería devolver valores más altos para estados mejores, así que, para el problema de las 8-reinas utilizaremos el número de pares de reinas no atacadas, que tiene un valor de 28 para una solución. Los valores de los cuatro estados son 24, 23, 20 y 11. En esta variante particular del algoritmo genético, la probabilidad de ser elegido para la reproducción es directamente proporcional al resultado de idoneidad, y los porcentajes se muestran junto a los tanteos.

CRUCE

En (c), se seleccionan dos pares, de manera aleatoria, para la reproducción, de acuerdo con las probabilidades en (b). Notemos que un individuo se selecciona dos veces y uno ninguna[11]. Para que cada par se aparee, se elige aleatoriamente un punto de **cruce** de las posiciones en la cadena. En la Figura 4.15 los puntos de cruce están después del tercer dígito en el primer par y después del quinto dígito en el segundo par[12].

En (d), los descendientes se crean cruzando las cadenas paternales en el punto de cruce. Por ejemplo, el primer hijo del primer par consigue los tres primeros dígitos del

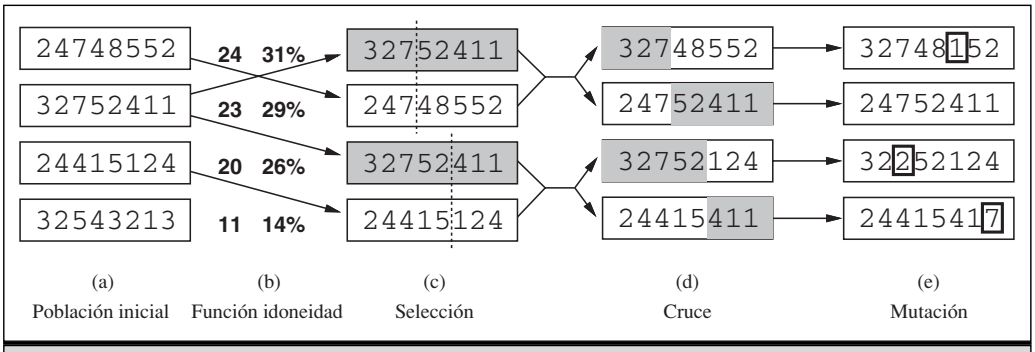


Figura 4.15 Algoritmo genético. La población inicial en (a) es ordenada con la función idoneidad en (b), y resultan pares para acoplamiento en (c). Ellos producen los descendientes en (d), que están sujetos a mutación en (e).

[11] Hay muchas variantes de esta regla de selección. Puede demostrarse que el método **selectivo**, en el que se desechan todos los individuos debajo de un umbral dado, converge más rápido que la versión aleatoria (Basum *et al*., 1995).

[12] Son aquí los asuntos de codificación. Si se usa una codificación de 24 bit en vez de ocho dígitos, entonces el punto de cruce tiene 2/3 de posibilidad de estar en medio de un dígito, que resulta en una mutación esencialmente arbitraria de ese dígito.

primer padre y los dígitos restantes del segundo padre, mientras que el segundo hijo consigue los tres primeros dígitos del segundo padre y el resto del primer padre. En la Figura 4.16 se muestran los estados de las ocho reinas implicados en este paso de reproducción. El ejemplo ilustra el hecho de que, cuando dos estados padres son bastante diferentes, la operación de cruce puede producir un estado que está lejos de cualquiera de los estados padre. Esto es, a menudo, lo que ocurre al principio del proceso en el que la población es bastante diversa, así que el cruce (como en el temple simulado) con frecuencia realiza pasos grandes, al principio, en el espacio de estados en el proceso de búsqueda y pasos más pequeños, más tarde, cuando la mayor parte de individuos son bastante similares.

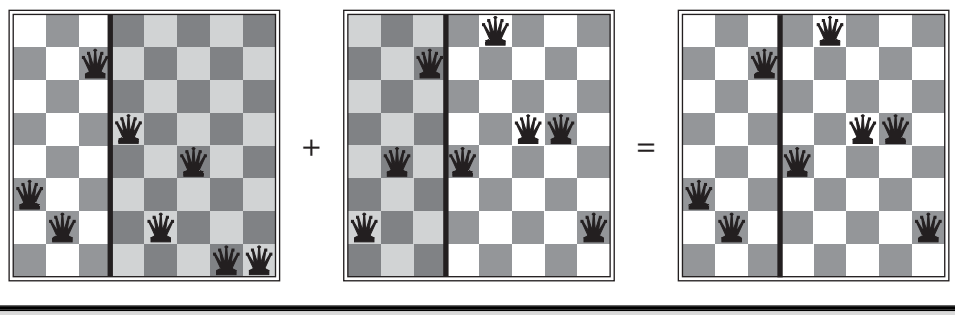


Figura 4.16 Los estados de las ocho reinas correspondientes a los dos primeros padres de la Figura 4.15(c) y el primer descendiente de Figura 4.15(d). Las columnas sombreadas se pierden en el paso de la transición y las columnas no sombreadas se mantienen.

MUTACIÓN

Finalmente, en (e), cada posición está sujeta a la **mutación** aleatoria con una pequeña probabilidad independiente. Un dígito fue transformado en el primer, tercer, y cuarto descendiente. El problema de las 8-reinas corresponde a escoger una reina aleatoriamente y moverla a un cuadrado aleatorio en su columna. La figura 4.17 describe un algoritmo que implementa todos estos pasos.

Como en la búsqueda por haz estocástica, los algoritmos genéticos combinan una tendencia ascendente con exploración aleatoria y cambian la información entre los hilos paralelos de búsqueda. La ventaja primera, si hay alguna, del algoritmo genético viene de la operación de cruce. Aún puede demostrarse matemáticamente que, si las posiciones del código genético se permutan al principio en un orden aleatorio, el cruce no comunica ninguna ventaja. Intuitivamente, la ventaja viene de la capacidad del cruce para combinar bloques grandes de letras que han evolucionado independientemente para así realizar funciones útiles, de modo que se aumente el nivel de granularidad en el que funciona la búsqueda. Por ejemplo, podría ser que poner las tres primeras reinas en posiciones 2, 4 y 6 (donde ellas no se atacan las unas a las otras) constituya un bloque útil que pueda combinarse con otros bloques para construir una solución.

ESQUEMA

La teoría de los algoritmos genéticos explica cómo esta teoría trabaja utilizando la idea de un **esquema**, una subcadena en la cual algunas de las posiciones se pueden dejar inespecíficas. Por ejemplo, el esquema 246***** describe todos los estados de

```
función ALGORITMO-GENÉTICO(población,IDONEIDAD) devuelve un individuo
  entradas: población, un conjunto de individuos
            IDONEIDAD, una función que mide la capacidad de un individuo

  repetir
    nueva_población ← conjunto vacío
    bucle para i desde 1 hasta TAMAÑO(población) hacer
      x ← SELECCIÓN-ALEATORIA(población, IDONEIDAD)
      y ← SELECCIÓN-ALEATORIA(población, IDONEIDAD)
      hijo ← REPRODUCIR(x,y)
      si (probabilidad aleatoria pequeña) entonces hijo ← MUTAR(hijo)
      añadir hijo a nueva_población
    población ← nueva_población
  hasta que algún individuo es bastante adecuado, o ha pasado bastante tiempo
  devolver el mejor individuo en la población, de acuerdo con la IDONEIDAD

función REPRODUCIR(x,y) devuelve un individuo
  entradas: x,y, padres individuales

  n ← LONGITUD(x)
  c ← número aleatorio de 1 a n
  devolver AÑADIR(SUBCADENA(x, 1, c),SUBCADENA(y, c + 1, n))
```

Figura 4.17 Algoritmo genético. El algoritmo es el mismo que el de la Figura 4.15, con una variación: es la versión más popular; cada cruce de dos padres produce sólo un descendiente, no dos.

ocho reinas en los cuales las tres primeras reinas están en posiciones 2, 4 y 6 respectivamente. A las cadenas que emparejan con el esquema (tal como 24613578) se les llaman **instancias** del esquema. Se puede demostrar que, si la idoneidad media de las instancias de un esquema está por encima de la media, entonces el número de instancias del esquema dentro de la población crecerá con el tiempo. Claramente, este efecto improbablemente será significativo si los bits adyacentes están totalmente no relacionados uno al otro, porque entonces habrá pocos bloques contiguos que proporcionen un beneficio consistente. Los algoritmos genéticos trabajan mejor cuando los esquemas corresponden a componentes significativos de una solución. Por ejemplo, si la cadenas son una representación de una antena, entonces los esquemas pueden representar los componentes de la antena, tal como reflectores y deflectores. Un componente bueno probablemente estará bien en una variedad de diseños diferentes. Esto sugiere que el uso acertado de algoritmos genéticos requiere la ingeniería cuidadosa de la representación.

En la práctica, los algoritmos genéticos han tenido un impacto extendido sobre problemas de optimización, como disposición de circuitos y el programado del trabajo en tiendas. Actualmente, no está claro si lo solicitado de los algoritmos genéticos proviene de su funcionamiento o de sus orígenes estéticamente agradables de la teoría de la evolución. Se han hecho muchos trabajos para identificar las condiciones bajo las cuales los algoritmos genéticos funcionan bien.

EVOLUCIÓN Y BÚSQUEDA

La teoría de la **evolución** fue desarrollada por Charles Darwin (1859) en *El Origen de Especies por medio de la Selección Natural*. La idea central es simple: las variaciones (conocidas como **mutaciones**) ocurren en la reproducción y serán conservadas en generaciones sucesivas aproximadamente en la proporción de su efecto sobre la idoneidad reproductiva.

La teoría de Darwin fue desarrollada sin el conocimiento de cómo los rasgos de los organismos se pueden heredar y modificar. Las leyes probabilísticas que gobiernan estos procesos fueron identificadas primero por Gregor Mendel (1866), un monje que experimentó con guisantes dulces usando lo que él llamó la fertilización artificial. Mucho más tarde, Watson y Crick (1953) identificaron la estructura de la molécula de ADN y su alfabeto, AGTC (adenina, guanina, timina, citocina). En el modelo estándar, la variación ocurre tanto por mutaciones en la secuencia de letras como por «el cruce» (en el que el ADN de un descendiente se genera combinando secciones largas del ADN de cada padre).

Ya se ha descrito la analogía con algoritmos de búsqueda local; la diferencia principal entre la búsqueda de haz estocástica y la evolución es el uso de la reproducción sexual, en donde los sucesores se generan a partir de *múltiples* organismos más que de solamente uno. Los mecanismos actuales de la evolución son, sin embargo, mucho más ricos de lo que permiten la mayoría de los algoritmos genéticos. Por ejemplo, las mutaciones pueden implicar inversiones, copias y movimientos de trozos grandes de ADN; algunos virus toman prestado el ADN de un organismo y lo insertan en otro; y hay genes reemplazables que no hacen nada pero se copian miles de veces dentro del genoma. Hay hasta genes que envenenan células de compañeros potenciales que no llevan el gen, bajando el aumento de sus posibilidades de réplica. Lo más importante es el hecho de que los *genes codifican los mecanismos* por los cuales se reproduce y traslada el genoma en un organismo. En algoritmos genéticos, esos mecanismos son un programa separado que no está representado dentro de las cadenas manipuladas.

La evolución Darwiniana podría parecer más bien un mecanismo ineficaz, y ha generado ciegamente aproximadamente 10^{45} organismos sin mejorar su búsqueda heurística un ápice. 50 años antes de Darwin, sin embargo, el gran naturalista francés Jean Lamarck (1809) propuso una teoría de evolución por la cual los rasgos *adquiridos por la adaptación durante la vida de un organismo* serían pasados a su descendiente. Tal proceso sería eficaz, pero no parece ocurrir en la naturaleza. Mucho más tarde, James Baldwin (1896) propuso una teoría superficialmente similar: aquel comportamiento aprendido durante la vida de un organismo podría acelerar la evolución. A diferencia de la de Lamarck, la teoría de Baldwin es completamente consecuente con la evolución Darwiniana, porque confía en presiones de selección que funcionan sobre individuos que han encontrado óptimos locales entre el conjunto de comportamientos posibles permitidos por su estructura genética. Las simulaciones por computadores modernos confirman que «el efecto de Baldwin» es real, a condición de que la evolución «ordinaria» pueda crear organismos cuya medida de rendimiento está, de alguna manera, correlacionada con la idoneidad actual.

4.4 Búsqueda local en espacios continuos

En el Capítulo 2, explicamos la diferencia entre entornos discretos y continuos, señalando que la mayor parte de los entornos del mundo real son continuos. Aún ninguno de los algoritmos descritos puede manejar espacios de estados continuos, ¡la función sucesor en la mayor parte de casos devuelve infinitamente muchos estados! Esta sección proporciona una *muy breve* introducción a técnicas de búsqueda local para encontrar soluciones óptimas en espacios continuos. La literatura sobre este tema es enorme; muchas de las técnicas básicas se originaron en el siglo XVII, después del desarrollo de cálculo Newton y Leibniz[13]. Encontraremos usos para estas técnicas en varios lugares del libro, incluso en los capítulos sobre aprendizaje, visión y robótica. En resumen, cualquier cosa que trata con el mundo real.

Comencemos con un ejemplo. Supongamos que queremos colocar tres nuevos aeropuertos en cualquier lugar de Rumanía, de forma tal que la suma de las distancias al cuadrado de cada ciudad sobre el mapa (Figura 3.2) a su aeropuerto más cercano sea mínima. Entonces el espacio de estados está definido por las coordenadas de los aeropuertos: (x_1, y_1), (x_2, y_2), y (x_3, y_3). Es un espacio *seis-dimensional*; también decimos que los estados están definidos por seis **variables** (en general, los estados están definidos por un vector n-dimensional de variables, $\mathbf{x}$). Moverse sobre este espacio se corresponde a movimientos de uno o varios de los aeropuertos sobre el mapa. La función objetivo $f(x_1, y_1, x_2, y_2, x_3, y_3)$ es relativamente fácil calcularla para cualquier estado particular una vez que tenemos las ciudades más cercanas, pero bastante complicado anotar en general.

Un modo de evitar problemas continuos es simplemente discretizar la vecindad de cada estado. Por ejemplo, podemos movernos sólo sobre un aeropuerto a la vez, en la dirección x o y, en una cantidad fija $\pm\delta$. Con seis variables, nos da 12 sucesores para cada estado. Podemos aplicar entonces cualquiera de los algoritmos de búsqueda local descritos anteriormente. Uno puede aplicar también la ascensión de colinas estocástica y el temple simulado directamente, sin discretizar el espacio. Estos algoritmos eligen a los sucesores aleatoriamente, que pueden hacerse por la generación de vectores aleatorios de longitud δ.

GRADIENTE

Hay muchos métodos que intentan usar el **gradiente** del paisaje para encontrar un máximo. El gradiente de la función objetivo es un vector ∇f que nos da la magnitud y la dirección de la inclinación más escarpada. Para nuestro problema, tenemos

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

En algunos casos, podemos encontrar un máximo resolviendo la ecuación $\nabla f = 0$ (esto podría hacerse, por ejemplo, si estamos colocando solamente un aeropuerto; la solución es la media aritmética de todas las coordenadas de las ciudades). En muchos casos, sin embargo, esta ecuación no puede resolverse de forma directa. Por ejemplo, con tres aeropuertos, la expresión para el gradiente depende de qué ciudades son las más cercanas a cada aeropuerto en el estado actual. Esto significa que podemos calcular el gradiente

[13] Un conocimiento básico de cálculo multivariante y aritmética vectorial es útil cuando uno lee esta sección.

localmente pero no *globalmente*. Incluso, podemos realizar todavía la ascensión de colinas por la subida más escarpada poniendo al día el estado actual con la fórmula

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

donde α es una constante pequeña. En otros casos, la función objetivo podría no estar disponible de una forma diferenciable, por ejemplo, el valor de un conjunto particular de posiciones de los aeropuertos puede determinarse ejecutando algún paquete de simulación económica a gran escala. En esos casos, el llamado **gradiente empírico** puede determinarse evaluando la respuesta a pequeños incrementos y decrecimientos en cada coordenada. La búsqueda de gradiente empírico es la misma que la ascensión de colinas con subida más escarpada en una versión discretizada del espacio de estados.

GRADIENTE EMPÍRICO

Bajo la frase «α es una constante pequeña» se encuentra una enorme variedad de métodos ajustando α. El problema básico es que, si α es demasiado pequeña, necesitamos demasiados pasos; si α es demasiado grande, la búsqueda podría pasarse del máximo. La técnica de **línea de búsqueda** trata de vencer este dilema ampliando la dirección del gradiente actual (por lo general duplicando repetidamente α) hasta que f comience a disminuir otra vez. El punto en el cual esto ocurre se convierte en el nuevo estado actual. Hay varias escuelas de pensamiento sobre cómo debe elegirse la nueva dirección en este punto.

LÍNEA DE BÚSQUEDA

Para muchos problemas, el algoritmo más eficaz es el venerable método de **Newton-Raphson** (Newton, 1671; Raphson, 1690). Es una técnica general para encontrar raíces de funciones, es decir la solución de ecuaciones de la forma $g(x) = 0$. Trabaja calculando una nueva estimación para la raíz x según la fórmula de Newton.

NEWTON-RAPHSON

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

Para encontrar un máximo o mínimo de f, tenemos que encontrar x tal que el gradiente es cero (es decir, $\nabla f(\mathbf{x}) = \mathbf{0}$). Así $g(x)$, en la fórmula de Newton, se transforma en $\nabla f(\mathbf{x})$, y la ecuación de actualización puede escribirse en forma de vector-matriz como

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

donde $\mathbf{H}_f(\mathbf{x})$ es la matriz **Hesiana** de segundas derivadas, cuyo los elementos H_{ij} están descritos por $\partial^2 f/\partial x_i \partial x_j$. Ya que el Hesiano tiene n^2 entradas, Newton-Raphson se hace costoso en espacios dimensionalmente altos, y por tanto, se han desarrollado muchas aproximaciones.

HESIANA

Los métodos locales de búsqueda sufren de máximos locales, crestas, y mesetas tanto en espacios de estados continuos como en espacios discretos. Se pueden utilizar el reinicio aleatorio y el temple simulado y son a menudo provechosos. Los espacios continuos dimensionalmente altos son, sin embargo, lugares grandes en los que es fácil perderse.

Un tema final, que veremos de pasada, es la **optimización con restricciones**. Un problema de optimización está restringido si las soluciones debieran satisfacer algunas restricciones sobre los valores de cada variable. Por ejemplo, en nuestro problema de situar aeropuertos, podría restringir los lugares para estar dentro de Rumanía y sobre la tierra firme (más que en medio de lagos). La dificultad de los problemas de optimización con restricciones depende de la naturaleza de las restricciones y la función objetivo. La categoría más conocida es la de los problemas de **programación lineal**, en los cuales las restricciones deben ser desigualdades lineales formando una región *convexa* y la función

OPTIMIZACIÓN CON RESTRICCIONES

PROGRAMACIÓN LINEAL

objetiva es también lineal. Los problemas de programación lineal pueden resolverse en tiempo polinomial en el número de variables. También se han estudiado problemas con tipos diferentes de restricciones y funciones objetivo (programación cuadrática, programación cónica de segundo orden, etcétera).

4.5 Agentes de búsqueda *online* y ambientes desconocidos

BÚSQUEDA *OFFLINE*

Hasta ahora nos hemos centrado en agentes que usan algoritmos de **búsqueda *offline***. Ellos calculan una solución completa antes de poner un pie en el mundo real (*véase* la Figura 3.1), y luego ejecutan la solución sin recurrir a su percepciones. En contraste, un

BÚSQUEDA *ONLINE*

agente de **búsqueda en línea (*online*)**[14] funciona **intercalando** el cálculo y la acción: primero toma una acción, entonces observa el entorno y calcula la siguiente acción. La búsqueda *online* es una buena idea en dominios dinámicos o semidinámicos (dominios donde hay una penalización por holgazanear y por utilizar demasiado tiempo para calcular). La búsqueda *online* es una idea incluso mejor para dominios estocásticos. En general, una búsqueda *offline* debería presentar un plan de contingencia exponencialmente grande que considere todos los acontecimientos posibles, mientras que una búsqueda *online* necesita sólo considerar lo que realmente pasa. Por ejemplo, a un agente que juega al ajedrez se le aconseja que haga su primer movimiento mucho antes de que se haya resuelto el curso completo del juego.

PROBLEMA DE EXPLORACIÓN

La búsqueda online es una idea *necesaria* para un **problema de exploración**, donde los estados y las acciones son desconocidos por el agente; un agente en este estado de ignorancia debe usar sus acciones como experimentos para determinar qué hacer después, y a partir de ahí debe intercalar el cálculo y la acción.

El ejemplo básico de búsqueda *online* es un robot que se coloca en un edificio nuevo y lo debe explorar para construir un mapa, que puede utilizar para ir desde A a B. Los métodos para salir de laberintos (conocimiento requerido para aspirar a ser héroe de la Antigüedad) son también ejemplos de algoritmos de búsqueda *online*. Sin embargo, la exploración espacial no es la única forma de la exploración. Considere a un bebé recién nacido: tiene muchas acciones posibles, pero no sabe los resultados de ellas, y ha experimentado sólo algunos de los estados posibles que puede alcanzar. El descubrimiento gradual del bebé de cómo trabaja el mundo es, en parte, un proceso de búsqueda *online*.

Problemas de búsqueda en línea (*online*)

Un problema de búsqueda *online* puede resolverse solamente por un agente que ejecute acciones, más que por un proceso puramente computacional. Asumiremos que el agente sabe lo siguiente:

[14] El término «en línea (*online*)» es comúnmente utilizado en informática para referirse a algoritmos que deben tratar con los datos de entrada cuando se reciben, más que esperar a que esté disponible el conjunto entero de datos de entrada.

- ACCIONES (s), que devuelve una lista de acciones permitidas en el estado s;
- Funciones de coste individual $c(s, a, s')$ (notar que no puede usarse hasta que el agente sepa que s' es el resultado); y
- TEST-OBJETIVO(s).

Notemos en particular que el agente *no puede* tener acceso a los sucesores de un estado excepto si intenta realmente todas las acciones en ese estado. Por ejemplo, en el problema del laberinto de la Figura 4.18, el agente no sabe que *Subir* desde (1,1) conduce a (1,2); ni, habiendo hecho esto, sabe que *Bajar* lo devolverá a (1,1). Este grado de ignorancia puede reducirse en algunas aplicaciones (para ejemplo, un robot explorador podría saber cómo trabajan sus acciones de movimiento y ser ignorante sólo de las posiciones de los obstáculos).

Asumiremos que el agente puede reconocer siempre un estado que ha visitado anteriormente, y asumiremos que las acciones son deterministas (en el Capítulo 17, se relajarán estos dos últimos axiomas). Finalmente, el agente podría tener acceso a una función heurística admisible $h(s)$ que estime la distancia del estado actual a un estado objetivo. Por ejemplo, en la Figura 4.18, el agente podría saber la posición del objetivo y ser capaz de usar la distancia heurística de Manhattan.

Típicamente, el objetivo del agente es alcanzar un estado objetivo minimizando el coste (otro objetivo posible es explorar simplemente el entorno entero). El costo es el costo total del camino por el que el agente viaja realmente. Es común comparar este costo con el costo del camino que el agente seguiría *si supiera el espacio de búsqueda de antemano,* es decir, el camino más corto actual (o la exploración completa más corta). En el lenguaje de algoritmos *online*, se llama **proporción competitiva**; nos gustaría que fuera tan pequeña como sea posible.

PROPORCIÓN COMPETITIVA

Aunque ésta suene como una petición razonable, es fácil ver que la mejor proporción alcanzable competitiva es infinita en algunos casos. Por ejemplo, si algunas acciones son irreversibles, la búsqueda *online* podría alcanzar, por casualidad, un estado sin salida del cual no es accesible ningún estado objetivo.

Quizás encuentre el término «por casualidad» poco convincente (después de todo, podría haber un algoritmo que no tome el camino sin salida mientras explora). Nuestra

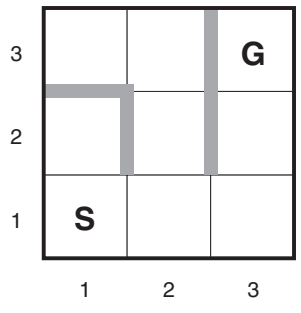


Figura 4.18 Un problema sencillo de un laberinto. El agente comienza en S y debe alcanzar G, pero no sabe nada del entorno.

reclamación, para ser más precisos, consiste en que *ningún algoritmo puede evitar callejones sin salida en todos los espacios de estados*. Considere los dos espacios de estados sin salida de la Figura 4.19 (a). A un algoritmo de búsqueda *online* que haya visitado los estados *S* y *A*, los dos espacios de estados parecen *idénticos*, entonces debe tomar la misma decisión en ambos. Por lo tanto, fallará en uno de ellos. Es un ejemplo de un **argumento de adversario** (podemos imaginar un adversario que construye el espacio de estados, mientras el agente lo explora, y puede poner el objetivo y callejones sin salida donde le guste).

ARGUMENTO DE ADVERSARIO

Los callejones sin salida son una verdadera dificultad para la exploración de un robot (escaleras, rampas, acantilados, y todas las clases de posibilidades presentes en terrenos naturales de acciones irreversibles). Para avanzar, asumiremos simplemente que el espacio de estados es **seguramente explorable**, es decir, algún estado objetivo es alcanzable desde cualquier estado alcanzable. Los espacios de estados con acciones reversibles, como laberintos y 8-puzles, pueden verse como grafos no-dirigidos y son claramente explorables.

SEGURAMENTE EXPLORABLE

Incluso en entornos seguramente explorables no se puede garantizar ninguna proporción competitiva acotada si hay caminos de costo ilimitado. Esto es fácil de demostrar en entornos con acciones irreversibles, pero de hecho permanece cierto también para el caso reversible, como se muestra en la Figura 4.19(b). Por esta razón, es común describir el funcionamiento de los algoritmos de búsqueda *online* en términos del tamaño del espacio de estados entero más que, solamente, por la profundidad del objetivo más superficial.

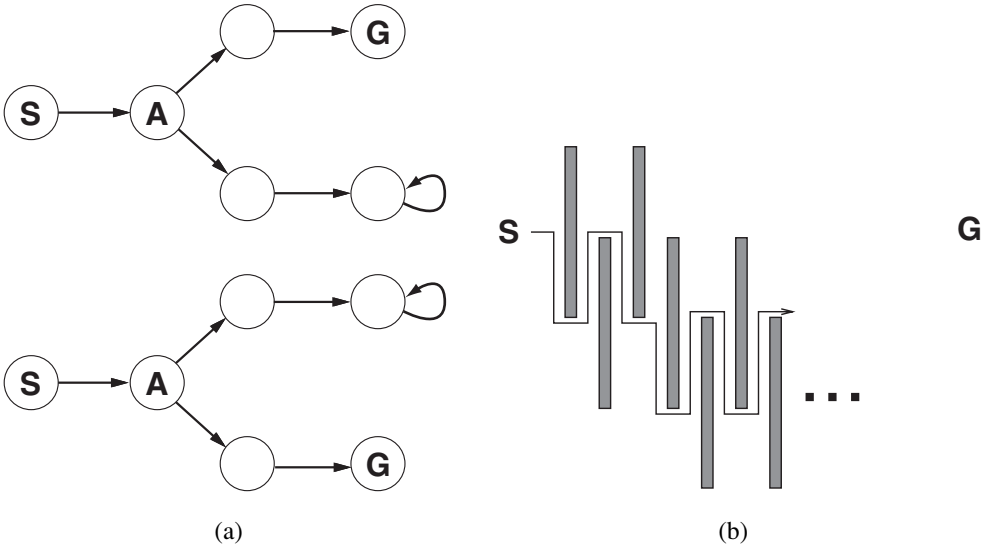


Figura 4.19 (a) Dos espacios de estados que podrían conducir a un agente de búsqueda *online* a un callejón sin salida. Cualquier agente fallará en al menos uno de estos espacios. (b) Un entorno de dos-dimensiones que puede hacer que un agente de búsqueda *online* siga una ruta arbitrariamente ineficaz al objetivo. Ante cualquier opción que tome el agente, el adversario bloquea esa ruta con otra pared larga y delgada, de modo que el camino seguido sea mucho más largo que el camino mejor posible.

Agentes de búsqueda en línea (*online*)

Después de cada acción, un agente *online* recibe una percepción al decirle que estado ha alcanzado; de esta información, puede aumentar su mapa del entorno. El mapa actual se usa para decidir dónde ir después. Esta intercalación de planificación y acción significa que los algoritmos de búsqueda *online* son bastante diferentes de los algoritmos de búsqueda *offline* vistos anteriormente. Por ejemplo, los algoritmos *offline* como A* tienen la capacidad de expandir un nodo en una parte del espacio y luego inmediatamente expandir un nodo en otra parte del espacio, porque la expansión de un nodo implica simulación más que verdaderas acciones. Un algoritmo *online*, por otra parte, puede expandir sólo el nodo que ocupa físicamente. Para evitar viajar a través de todo el árbol para expandir el siguiente nodo, parece mejor expandir los nodos en un orden *local*. La búsqueda primero en profundidad tiene exactamente esta propiedad, porque (menos cuando volvemos hacia atrás) el siguiente nodo a expandir es un hijo del nodo anteriormente expandido.

En la Figura 4.20 se muestra un agente de búsqueda primero en profundidad *online*. Este agente almacena su mapa en una tabla, *resultado*[*a*,*s*], que registra el estado que resulta de ejecutar la acción *a* en el estado *s*. Siempre que una acción del estado actual no haya sido explorada, el agente intenta esa acción. La dificultad viene cuando el agente ha intentado todas las acciones en un estado. En la búsqueda primero en profundidad *offline*, el estado es simplemente quitado de la cola; en una búsqueda *online*, el agente tiene que volver atrás físicamente. La búsqueda primero en profundidad, significa volver al estado el cual el agente incorporó el estado actual más recientemente. Esto se con-

```
función AGENTE-BPP-ONLINE(s′) devuelve una acción
  entradas: s′, una percepción que identifica el estado actual
  estático: resultado, una tabla, indexada por la acción y el estado, inicialmente vacía
            noexplorados, una tabla que enumera, para cada estado visitado, las acciones
              todavía no intentadas
            nohaciatras, una tabla que enumera, para cada estado visitado, los nodos hacia
              atrás todavía no intentados
            s,a, el estado y acción previa, inicialmente nula

  si TEST-OBJETIVO(s′) entonces devolver parar
  si s′ es un nuevo estado entonces noexplorados[s′] ← ACCIONES(s′)
  si s es no nulo entonces hacer
    resultado[a,s] ← s′
    añadir s al frente de nohaciatras[s′]
  si noexplorados[s′] esta vacío entonces
    si nohaciatras[s′] esta vacío entonces devolver parar
    en caso contrario a ← una acción b tal que resultado[b, s′] = POP(nohaciatras[s′])
  en caso contrario a ← POP(noexplorados[s′])
  s ← s′
  devolver a
```

Figura 4.20 Un agente de búsqueda *online* que utiliza la exploración primero en profundidad. El agente es aplicable, solamente, en espacios de búsqueda bidireccionales.

sigue guardando una tabla que pone en una lista, para cada estado, los estados predecesores a los cuales aún no ha vuelto. Si el agente se ha quedado sin estados a los que volver, entonces su búsqueda se ha completado.

Recomendamos al lector que compruebe el progreso del AGENTE-BPP-ONLINE cuando se aplica al laberinto de la Figura 4.18. Es bastante fácil ver que el agente, en el caso peor, terminará por cruzar cada enlace en el espacio de estados exactamente dos veces. Por exploración, esto es lo óptimo; para encontrar un objetivo, por otra parte, la proporción competitiva del agente podría ser arbitrariamente mala si se viaja sobre un camino largo cuando hay un objetivo directamente al lado del estado inicial. Una variante *online* de la profundidad iterativa resuelve este problema; para un entorno representado por un árbol uniforme, la proporción competitiva del agente es una constante pequeña.

A causa de su método de vuelta atrás, el AGENTE-BPP-ONLINE trabaja sólo en espacios de estados donde las acciones son reversibles. Hay algoritmos ligeramente más complejos que trabajan en espacios de estados generales, pero ninguno de estos algoritmos tiene una proporción competitiva acotada.

Búsqueda local en línea (*online*)

Como la búsqueda primero en profundidad, la **búsqueda de ascensión de colinas** tiene la propiedad de localidad en sus expansiones de los nodos. ¡De hecho, porque mantiene un estado actual en memoria, la búsqueda de ascensión de colinas es ya un algoritmo de búsqueda *online*! Desafortunadamente, no es muy útil en su forma más simple porque deja al agente que se sitúe en máximos locales con ningún movimiento que hacer. Por otra parte, los reinicios aleatorios no pueden utilizarse, porque el agente no puede moverse a un nuevo estado.

CAMINO ALEATORIO

En vez de reinicios aleatorios, podemos considerar el uso de un **camino aleatorio** para explorar el entorno. Un camino aleatorio selecciona simplemente al azar una de las acciones disponibles del estado actual; se puede dar preferencia a las acciones que todavía no se han intentado. Es fácil probar que un camino aleatorio encontrará *al final* un objetivo o termina su exploración, a condición de que el espacio sea finito[15]. Por otra parte, el proceso puede ser muy lento. La Figura 4.21 muestra un entorno en el que un

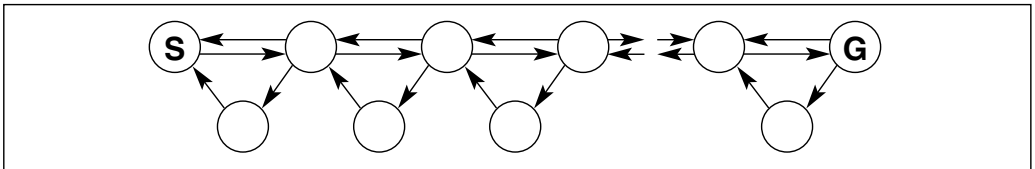


Figura 4.21 Un entorno en el cual un camino aleatorio utilizará exponencialmente muchos pasos para encontrar el objetivo.

[15] El caso infinito es mucho más difícil. ¡Los caminos aleatorios son completos en rejillas unidimensionales y de dos dimensiones infinitas, pero no en rejillas tridimensionales! En el último caso, la probabilidad de que el camino vuelva siempre al punto de partida es alrededor de 0,3405 (*véase* a Hughes, 1995, para una introducción general).

camino aleatorio utilizará un número exponencial de pasos para encontrar el objetivo, porque, en cada paso, el progreso hacia atrás es dos veces más probable que el progreso hacia delante. El ejemplo es artificial, por supuesto, pero hay muchos espacios de estados del mundo real cuya topología causa estas clases de «trampas» para los caminos aleatorios.

Aumentar a la ascensión de colinas con *memoria* más que aleatoriedad, resulta ser una aproximación más eficaz. La idea básica es almacenar una «mejor estimación actual» $H(s)$ del coste para alcanzar el objetivo desde cada estado que se ha visitado. El comienzo de $H(s)$ es justo la estimación heurística $h(s)$ y se actualiza mientras que el agente gana experiencia en el espacio de estados. La Figura 4.22 muestra un ejemplo sencillo en un espacio de estados unidimensional. En (a), el agente parece estar estancado en un mínimo local plano en el estado sombreado. Más que permanecer donde está, el agente debe seguir por donde parece ser la mejor trayectoria al objetivo, basada en las estimaciones de los costes actuales para sus vecinos. El coste estimado para alcanzar el objetivo a través de un vecino s' es el coste para conseguir s' más el coste estimado para conseguir un objetivo desde ahí, es decir, $c(s, a, s') + H(s')$. En el ejemplo, hay dos acciones con costos estimados $1 + 9$ y $1 + 2$, así parece que lo mejor posible es moverse a la derecha. Ahora, está claro que la estimación del costo de dos para el estado sombreado fue demasiado optimista. Puesto que el mejor movimiento costó uno y condujo a un estado que está al menos a dos pasos de un objetivo, el estado sombreado debe estar por lo menos a tres pasos de un objetivo, así que su H debe actualizarse adecuadamente, como se muestra en la figura 4.22(b). Continuando este proceso, el agente se moverá hacia delante y hacia atrás dos veces más, actualizando H cada vez y «apartando» el mínimo local hasta que se escapa a la derecha.

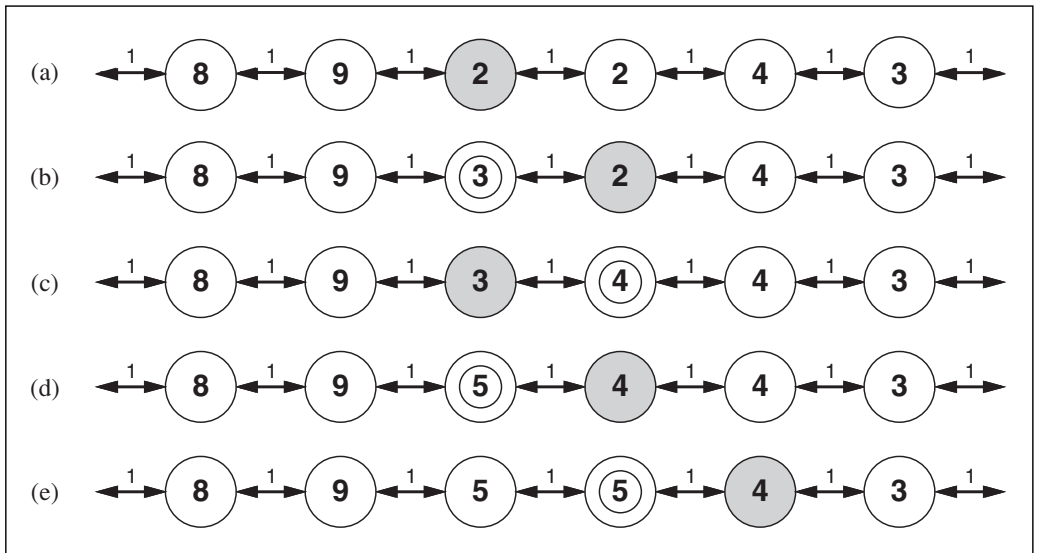


Figura 4.22 Cinco iteraciones de AA*TR en un espacio de estados unidimensional. Cada estado se marca con $H(s)$, la estimación del costo actual para alcanzar un objetivo, y cada arco se marca con su costo. El estado sombreado marca la ubicación del agente, y se rodean los valores actualizados en cada iteración.

AA*TR

En la Figura 4.23 se muestra un agente que implementa este esquema, llamado aprendiendo A* en tiempo real (**AA*TR**). Como el AGENTE-BPP-ONLINE, éste construye un mapa del entorno usando la tabla *resultado*. Actualiza el costo estimado para el estado que acaba de dejar y entonces escoge el movimiento «aparentemente mejor» según sus costos estimados actuales. Un detalle importante es que las acciones que todavía no se han intentado en un estado s siempre se supone que dirigen inmediatamente al objetivo con el costo menor posible, $h(s)$. Este **optimismo bajo la incertidumbre** anima al agente a explorar nuevos y posiblemente caminos prometedores.

OPTIMISMO BAJO INCERTIDUMBRE

función AGENTE-AA*TR(s') **devuelve** una acción
  entradas: s', una percepción que identifica el estado actual
  estático: *resultado*, una tabla, indexada por la acción y el estado, inicialmente vacía
    H, una tabla de costos estimados indexada por estado, inicialmente vacía
    s, a, el estado y acción previa, inicialmente nula

  si TEST-OBJETIVO(s') **entonces devolver** *parar*
  si s' es un nuevo estado (no en H) **entonces** $H[s'] \leftarrow h(s')$
  a menos que s sea nulo
    $resultado[a, s] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACCIONES}(s)}$ COSTO-AA*TR($s, b, resultado[b, s], H$)
  $a \leftarrow$ una acción b de ACCIONES(s') que minimiza COSTO-AA*TR($s', b, resultado[b, s'], H$)
  $s \leftarrow s'$
  devolver a

función COSTO-AA*TR(s, a, s', H) **devuelve** un costo estimado
  si s' está indefinido **entonces devolver** $h(s)$
  en otro caso devolver $c(s, a, s') + H[s']$

Figura 4.23 El AGENTE-AA*TR escoge una acción según los valores de los estados vecinos, que se actualizan conforme el agente se mueve sobre el espacio de estados.

Un agente AA*TR garantiza encontrar un objetivo en un entorno seguramente explorable y finito. A diferencia de A*, sin embargo, no es completo para espacios de estados infinitos (hay casos donde se puede dirigir infinitamente por mal camino). Puede explorar un entorno de n estados en $O(n^2)$ pasos, en el caso peor, pero a menudo lo hace mejor. El agente AA*TR es sólo uno de una gran familia de agentes *online* que pueden definirse especificando la regla de la selección de la acción y que actualiza la regla de maneras diferentes. Discutiremos esta familia, que fue desarrollada originalmente para entornos estocásticos, en el Capítulo 21.

Aprendizaje en la búsqueda en línea (*online*)

La ignorancia inicial de los agentes de búsqueda *online* proporciona varias oportunidades para aprender. Primero, los agentes aprenden un «mapa» del entorno (más precisamente, el resultado de cada acción en cada estado) simplemente registrando cada una de sus experiencias (notemos que la suposición de entornos deterministas quiere decir que

una experiencia es suficiente para cada acción). Segundo, los agentes de búsqueda locales adquieren estimaciones más exactas del valor de cada estado utilizando las reglas de actualización local, como en AA*TR. En el Capítulo 21 veremos que éstas actualizaciones convergen finalmente a valores *exactos* para cada estado, con tal de que el agente explore el espacio de estados de manera correcta. Una vez que se conocen los valores exactos, se pueden tomar las decisiones óptimas simplemente moviéndose al sucesor con el valor más alto (es decir, la ascensión de colinas pura es entonces una estrategia óptima).

Si usted siguió nuestra sugerencia de comprobar el comportamiento del AGENTE-BPP-ONLINE en el entorno de la Figura 4.18, habrá advertido que el agente no es muy brillante. Por ejemplo, después de ver que la acción *Arriba* va de (1,1) a (1,2), el agente no tiene la menor idea todavía que la acción *Abajo* vuelve a (1,1), o que la acción *Arriba* va también de (2,1) a (2,2), de (2,2) a (2,3), etcétera. En general, nos gustaría que el agente aprendiera que *Arriba* aumenta la coordenada *y* a menos que haya una pared en el camino, que hacia *Abajo* la reduce, etcétera. Para que esto suceda, necesitamos dos cosas: primero, necesitamos una representación formal y explícitamente manipulable para estas clases de reglas generales; hasta ahora, hemos escondido la información dentro de la caja negra llamada función sucesor. La Parte III está dedicada a este tema. Segundo, necesitamos algoritmos que puedan construir reglas generales adecuadas a partir de la observación específica hecha por el agente. Estos algoritmos se tratan en el Capítulo 18.

4.6 Resumen

Este capítulo ha examinado la aplicación de **heurísticas** para reducir los costos de la búsqueda. Hemos mirado varios algoritmos que utilizan heurísticas y encontramos que la optimalidad tiene un precio excesivo en términos del costo de búsqueda, aún con heurísticas buenas.

- **Búsqueda primero el mejor** es una BÚSQUEDA-GRAFO donde los nodos no expandidos de costo mínimo (según alguna medida) se escogen para la expansión. Los algoritmos primero el mejor utilizan típicamente una función heurística $h(n)$ que estima el costo de una solución desde n.
- **Búsqueda primero el mejor avara** expande nodos con $h(n)$ mínima. No es óptima, pero es a menudo eficiente.
- **Búsqueda A*** expande nodos con mínimo $f(n) = g(n) + h(n)$. A* es completa y óptima, con tal que garanticemos que $h(n)$ sea admisible (para BÚSQUEDA-ARBOL) o consistente (para BÚSQUEDA-GRAFO). La complejidad en espacio de A* es todavía prohibitiva.
- El rendimiento de los algoritmos de búsqueda heurística depende de la calidad de la función heurística. Las heurísticas buenas pueden construirse a veces relajando la definición del problema, por costos de solución precalculados para sub-problemas en un modelo de bases de datos, o aprendiendo de la experiencia con clases de problemas.

- **BRPM** y **A*MS** son algoritmos de búsqueda robustos y óptimos que utilizan cantidades limitadas de memoria; con suficiente tiempo, pueden resolver los problemas que A* no puede resolver porque se queda sin memoria.
- Los métodos de *búsqueda local,* como la **ascensión de colinas,** operan en formulaciones completas de estados, manteniendo sólo un número pequeño de nodos en memoria. Se han desarrollado varios algoritmos estocásticos, inclusive el **temple simulado**, que devuelven soluciones óptimas cuando se da un apropiado programa de enfriamiento. Muchos métodos de búsqueda local se pueden utilizar también para resolver problemas en espacios continuos.
- Un **algoritmo genético** es una búsqueda de ascensión de colinas estocástica en la que se mantiene una población grande de estados. Los estados nuevos se generan por **mutación** y por **cruce**, combinando pares de estados de la población.
- Los **problemas de exploración** surgen cuando el agente no tiene la menor idea acerca de los estados y acciones de su entorno. Para entornos seguramente explorables, los agentes de **búsqueda en línea** pueden construir un mapa y encontrar un objetivo si existe. Las estimaciones de las heurística, que se actualizan por la experiencia, proporcionan un método efectivo para escapar de mínimos locales.

Notas bibliográficas e históricas

El uso de información heurística en la resolución de problemas aparece en un artículo de Simon y Newell (1958), pero la frase «búsqueda heurística» y el uso de las funciones heurísticas que estiman la distancia al objetivo llegaron algo más tarde (Newell y Ernst, 1965; Lin, 1965). Doran y Michie (1966) dirigieron muchos estudios experimentales de búsqueda heurística aplicados a varios problemas, especialmente al 8-puzle y 15-puzle. Aunque Doran y Michie llevaran a cabo un análisis teórico de la longitud del camino y «penetrancia» (proporción entre la longitud del camino y el número total de nodos examinados hasta el momento) en la búsqueda heurística, parecen haber ignorado la información proporcionada por la longitud actual del camino. El algoritmo A*, incorporando la longitud actual del camino en la búsqueda heurística, fue desarrollado por Hart, Nilsson y Raphael (1968), con algunas correcciones posteriores (Hart *et al*., 1972). Dechter y Pearl (1985) demostraron la eficiencia óptima de A*.

El artículo original de A* introdujo la condición de consistencia en funciones heurísticas. La condición de monotonía fue introducida por Pohl (1977) como un sustituto más sencillo, pero Pearl (1984) demostró que las dos eran equivalentes. Varios algoritmos precedentes de A* utilizaron el equivalente de listas abiertas y cerradas; éstos incluyen la búsqueda primero en anchura, primero en profundidad, y costo uniforme (Bellman, 1957; Dijkstra, 1959). El trabajo de Bellman en particular mostró la importancia de añadir los costos de los caminos para simplificar los algoritmos de optimización.

Pohl (1970, 1977) fue el pionero en el estudio de la relación entre el error en las funciones heurísticas y la complejidad en tiempo de A*. La demostración de que A* se ejecuta en un tiempo lineal si el error de la función heurística está acotado por una constante puede encontrarse en Pohl (1977) y en Gaschnig (1979). Pearl (1984) reforzó este re-

sultado para permitir un crecimiento logarítmico en el error. El «factor de ramificación eficaz», medida de la eficiencia de la búsqueda heurística, fue propuesto por Nilsson (1971).

Hay muchas variaciones del algoritmo A*. Pohl (1973) propuso el uso del *ponderado dinámico*, el cual utiliza una suma ponderada $f_w(n) = w_g g(n) + w_h h(n)$ de la longitud del camino actual y de la función heurística como una función de evaluación, más que la suma sencilla $f(n) = g(n) + h(n)$ que utilizó A*. Los pesos w_g y w_h se ajustan dinámicamente con el progreso de la búsqueda. Se puede demostrar que el algoritmo de Pohl es ϵ-admisible (es decir, garantiza encontrar las soluciones dentro de un factor $1 + \epsilon$ de la solución óptima) donde ϵ es un parámetro suministrado al algoritmo. La misma propiedad es exhibida por el algoritmo A^*_ϵ (Pearl, 1984), el cual puede escoger cualquier nodo de la franja tal que su *f*-costo esté dentro de un factor $1 + \epsilon$ del nodo de la franja de *f*-costo más pequeño. La selección se puede hacer para minimizar el costo de la búsqueda.

A* y otros algoritmos de búsqueda en espacio de estados están estrechamente relacionados con las técnicas de *ramificar-y-acotar* ampliamente utilizadas en investigación operativa (Lawler y Wood, 1966). Las relaciones entre la búsqueda en espacio de estados y ramificar-y-acotar se han investigado en profundidad (Kumar y Kanal, 1983; Nau *et al*., 1984; Kumas *et al*., 1988). Martelli y Montanari (1978) demostraron una conexión entre la programación dinámica (*véase* el Capítulo 17) y cierto tipo de búsqueda en espacio de estados. Kumar y Kanal (1988) intentan una «ambiciosa unificación» de la búsqueda heurística, programación dinámica, y técnicas de ramifica-y-acotar bajo el nombre de PDC (el «proceso de decisión compuesto»).

Como los computadores a finales de los años 1950 y principios de los años 1960 tenían como máximo unas miles de palabras de memoria principal, la búsqueda heurística con memoria-acotada fue un tema de investigación. El Grafo Atravesado (Doran y Michie, 1966), uno de los programas de búsqueda más antiguos, compromete a un operador después de realizar una búsqueda primero el mejor hasta el límite de memoria. A*PI (Korf, 1985a, 1985b) fue el primero que usó un algoritmo de búsqueda heurística, óptima, con memoria-acotada y de la que se han desarrollado un número grande de variantes. Un análisis de la eficiencia de A*PI y de sus dificultades con las heurística real-valoradas aparece en Patrick *et al*. (1992).

El BRPM (Korf, 1991, 1993) es realmente algo más complicado que el algoritmo mostrado en la Figura 4.5, el cual está más cercano a un algoritmo, desarrollado independientemente, llamado **extensión iterativa**, o EI (Russell, 1992). BRPM usa una cota inferior y una cota superior; los dos algoritmos se comportan idénticamente con heurísticas admisibles, pero BRPM expande nodos en orden primero el mejor hasta con una heurística inadmisible. La idea de guardar la pista del mejor camino alternativo apareció en la implementación elegante, en Prolog, de A* realizada por Bratko (1986) y en el algoritmo DTA* (Russell y Wefald, 1991). El trabajo posterior también habló de espacios de estado meta nivel y aprendizaje meta nivel.

EXTENSIÓN ITERATIVA

El algoritmo A*M apareció en Chakrabarti *et al*. (1989). A*MS, o A*M simplificado, surgió de una tentativa de implementación de A*M como un algoritmo de comparación para IE (Russell, 1992). Kaindl y Khorsand (1994) han aplicado A*MS para producir un algoritmo de búsqueda bidireccional considerablemente más rápido que los

algoritmos anteriores. Korf y Zhang (2000) describen una aproximación divide-y-vencerás, y Zhou y Hansen (2002) introducen una búsqueda A* en un grafo de memoria-acotada. Korf (1995) revisa las técnicas de búsqueda de memoria-acotada.

La idea de que las heurísticas admisibles pueden obtenerse por relajación del problema aparece en el trabajo seminal de Held y Karp (1970), quien utilizó la heurística del mínimo-atravesando el árbol para resolver el PVC (*véase* el Ejercicio 4.8).

La automatización del proceso de relajación fue implementado con éxito por Prieditis (1993), construido sobre el trabajo previo de Mostow (Mostow y Prieditis, 1989). El uso del modelo de bases de datos para obtener heurísticas admisibles se debe a Gasser (1995) y Culberson y Schaeffer (1998); el modelo de bases de datos disjuntas está descrito por Korf y Felner (2002). La interpretación probabilística de las heurística fue investigada en profundidad por Pearl (1984) y Hansson y Mayer (1989).

La fuente bibliográfica más comprensiva sobre heurísticas y algoritmos de búsqueda heurísticos está en el texto *Heuristics* de Pearl (1984). Este libro cubre de una manera especialmente buena la gran variedad de ramificaciones y variaciones de A*, incluyendo demostraciones rigurosas de sus propiedades formales. Kanal y Kumar (1988) presentan una antología de artículos importantes sobre la búsqueda heurística. Los nuevos resultados sobre algoritmos de búsqueda aparecen con regularidad en la revista *Artificial Intelligence*.

Las técnicas locales de búsqueda tienen una larga historia en matemáticas y en informática. En efecto, el método de Newton-Raphson (Newton, 1671; Raphson, 1690) puede verse como un método de búsqueda local muy eficiente para espacios continuos en los cuales está disponible la información del gradiente. Brent (1973) es una referencia clásica para algoritmos de optimización que no requieren tal información. La búsqueda de haz, que hemos presentado como un algoritmo de búsqueda local, se originó como una variante de anchura-acotada de la programación dinámica para el reconocimiento de la voz en el sistema HARPY (Lowerre, 1976). En Pearl (1984, el Capítulo 5) se analiza en profundidad un algoritmo relacionado.

El tema de la búsqueda local se ha fortalecido en los últimos años por los resultados sorprendentemente buenos en problemas de satisfacción de grandes restricciones como las *n*-reinas (Minton *et al*., 1992) y de razonamiento lógico (Selman *et al*., 1992) y por la incorporación de aleatoriedad, múltiples búsquedas simultáneas, y otras mejoras. Este renacimiento, de lo que Christos Papadimitriou ha llamado algoritmos de la «Nueva Era», ha provocado también el interés entre los informáticos teóricos (Koutsoupias y Papadimitriou, 1992; Aldous y Vazirani, 1994). En el campo de la investigación operativa, una variante de la ascensión de colinas, llamada **búsqueda tabú,** ha ganado popularidad (Glover, 1989; Glover y Laguna, 1997). Realizado sobre modelos de memoria limitada a corto plazo de los humanos, este algoritmo mantienen una lista tabú de *k* estados, previamente visitados, que no pueden visitarse de nuevo; así se mejora la eficiencia cuando se busca en grafos y además puede permitir que el algoritmo se escape de algunos mínimos locales. Otra mejora útil sobre la ascensión de colinas es el algoritmo de STAGE (Boyan y Moore, 1998). La idea es usar los máximos locales encontrados por la ascensión de colinas de reinicio aleatorio para conseguir una idea de la forma total del paisaje. El algoritmo adapta una superficie suave al conjunto de máximos locales y luego calcula analíticamente el máximo global de esa superficie. Éste se convierte en el nuevo punto de

BÚSQUEDA TABÚ

reinicio. Se ha demostrado que este algoritmo trabaja, en la práctica, sobre problemas difíciles. (Gomes *et al*., 1998) mostraron que las distribuciones, en tiempo de ejecución, de los algoritmos de vuelta atrás sistemáticos a menudo tienen una **distribución de cola pesada**, la cual significa que la probabilidad de un tiempo de ejecución muy largo es mayor que lo que sería predicho si los tiempos de ejecución fueran normalmente distribuidos. Esto proporciona una justificación teórica para los reinicios aleatorios.

DISTRIBUCIÓN DE COLA PESADA

El temple simulado fue inicialmente descrito por Kirkpatrick *et al*. (1983), el cual se basó directamente en el **algoritmo de Metrópolis** (usado para simular sistemas complejos en la física (Metrópolis *et al*., 1953) y fue supuestamente inventado en la cena Los Alamos). El temple simulado es ahora un campo en sí mismo, con cien trabajos publicados cada año.

Encontrar soluciones óptimas en espacios continuos es la materia de varios campos, incluyendo la **teoría de optimización**, **teoría de control óptima**, y el **cálculo de variaciones**. Los convenientes (y prácticos) puntos de entrada son proporcionados por Press *et al*. (2002) y Bishop (1995). La **programación lineal** (PL) fue una de las primeras aplicaciones de computadores; el **algoritmo simplex** (Wood y Dantzig, 1949; Dantzig, 1949) todavía se utiliza a pesar de la complejidad exponencial, en el peor caso. Karmarkar (1984) desarrolló un algoritmo de tiempo polinomial práctico para PL.

El trabajo de Sewal Wright (1931), sobre el concepto de la **idoneidad de un paisaje**, fue un precursor importante para el desarrollo de los algoritmos genéticos. En los años 50, varios estadísticos, incluyendo Box (1957) y Friedman (1959), usaron técnicas evolutivas para problemas de optimización, pero no fue hasta que Rechenberg (1965, 1973) introdujera las **estrategias de evolución** para resolver problemas de optimización para planos aerodinámicos en la que esta aproximación ganó popularidad. En los años 60 y 70, John Holland (1975) defendió los algoritmos genéticos, como un instrumento útil y como un método para ampliar nuestra comprensión de la adaptación, biológica o de otra forma (Holland, 1995). El movimiento de **vida artificial** (Langton, 1995) lleva esta idea un poco más lejos, viendo los productos de los algoritmos genéticos como *organismos* más que como soluciones de problemas. El trabajo de Hinton y Nowlan (1987) y Ackley y Littman (1991) en este campo ha hecho mucho para clarificar las implicaciones del efecto de Baldwin. Para un tratamiento más a fondo y general sobre la evolución, recomendamos a Smith y Szathmáry (1999).

ESTRATEGIAS DE EVOLUCIÓN

VIDA ARTIFICIAL

La mayor parte de comparaciones de los algoritmos genéticos con otras aproximaciones (especialmente ascensión de colinas estocástica) han encontrado que los algoritmos genéticos son más lentos en converger (O'Reilly y Oppacher, 1994; Mitchell *et al*., 1996; Juels y Watternberg, 1996; Baluja, 1997). Tales conclusiones no son universalmente populares dentro de la comunidad de AG, pero tentativas recientes dentro de esa comunidad, para entender la búsqueda basada en la población como una forma aproximada de aprendizaje Bayesiano (*véase* el Capítulo 20), quizás ayude a cerrar el hueco entre el campo y sus críticas (Pelikan *et al*., 1999). La teoría de **sistemas dinámicos cuadráticos** puede explicar también el funcionamiento de AGs (Rabani *et al*., 1998). *Véase* Lohn *et al*. (2001) para un ejemplo de AGs aplicado al diseño de antenas, y Larrañaga *et al*. (1999) para una aplicación al problema de viajante de comercio.

El campo de la **programación genética** está estrechamente relacionado con los algoritmos genéticos. La diferencia principal es que las representaciones, que son muta-

PROGRAMACIÓN GENÉTICA

das y combinadas, son programas más que cadenas de bits. Los programas se representan en forma de árboles de expresión; las expresiones pueden estar en un lenguaje estándar como Lisp o pueden estar especialmente diseñadas para representar circuitos, controladores del robot, etcétera. Los cruces implican unir los subárboles más que las subcadenas. De esta forma, la mutación garantiza que los descendientes son expresiones gramaticalmente correctas, que no lo serían si los programas fueran manipulados como cadenas.

El interés reciente en la programación genética fue estimulado por el trabajo de John Koza (Koza, 1992, 1994), pero va por detrás de los experimentos con código máquina de Friedberg (1958) y con autómatas de estado finito de Fogel *et al.* (1966). Como con los algoritmos genéticos, hay un debate sobre la eficacia de la técnica. Koza *et al.* (1999) describen una variedad de experimentos sobre el diseño automatizado de circuitos de dispositivos utilizando la programación genética.

Las revistas *Evolutionary Computation* y *IEEE Transactions on Evolutionary Computation* cubren los algoritmos genéticos y la programación genética; también se encuentran artículos en *Complex Systems*, *Adaptative Behavior*, y *Artificial Life*. Las conferencias principales son la *International Conference on Genetic Algorithms* y la *Conference on Genetic Programming*, recientemente unidas para formar la *Genetic and Evolutionary Computation Conference*. Los textos de Melanie Mitchell (1996) y David Fogel (2000) dan descripciones buenas del campo.

Los algoritmos para explorar espacios de estados desconocidos han sido de interés durante muchos siglos. La búsqueda primero en profundidad en un laberinto puede implementarse manteniendo la mano izquierda sobre la pared; los bucles pueden evitarse marcando cada unión. La búsqueda primero en profundidad falla con acciones irreversibles; el problema más general de exploración de **grafos Eulerianos** (es decir, grafos en los cuales cada nodo tiene un número igual de arcos entrantes y salientes) fue resuelto por un algoritmo debido a Hierholzer (1873). El primer estudio cuidadoso algorítmico del problema de exploración para grafos arbitrarios fue realizado por Deng y Papadimitriou (1990), quienes desarrollaron un algoritmo completamente general, pero demostraron que no era posible una proporción competitiva acotada para explorar un grafo general. Papadimitriou y Yannakakis (1991) examinaron la cuestión de encontrar caminos a un objetivo en entornos geométricos de planificación de caminos (donde todas las acciones son reversibles). Ellos demostraron que es alcanzable una pequeña proporción competitiva con obstáculos cuadrados, pero no se puede conseguir una proporción acotada con obstáculos generales rectangulares (*véase* la Figura 4.19).

GRAFOS EULERIANOS

El algoritmo LRTA* fue desarrollado por Korf (1990) como parte de una investigación en la **búsqueda en tiempo real** para entornos en los cuales el agente debe actuar después de buscar en sólo una cantidad fija del tiempo (una situación mucho más común en juegos de dos jugadores). El LRTA* es, de hecho, un caso especial de algoritmos de aprendizaje por refuerzo para entornos estocásticos (Barto *et al.*, 1995). Su política de optimismo bajo incertidumbre (siempre se dirige al estado no visitado más cercano) puede causar un modelo de exploración que es menos eficiente, en el caso sin información, que la búsqueda primero en profundidad simple (Koening, 2000). Dasgupta *et al.* (1994) mostraron que la búsqueda en profundidad iterativa *online* es óptimamente eficiente para encontrar un objetivo en un árbol uniforme sin la información heurís-

BÚSQUEDA EN TIEMPO REAL

tica. Algunas variantes informadas sobre el tema LRTA* se han desarrollado con métodos diferentes para buscar y actualizar dentro de la parte conocida del grafo (Pemberton y Korf, 1992). Todavía no hay una buena comprensión de cómo encontrar los objetivos con eficiencia óptima cuando se usa información heurística.

BÚSQUEDA PARALELA

El tema de los algoritmos de **búsqueda paralela** no se ha tratado en el capítulo, en parte porque requiere una discusión larga de arquitecturas paralelas de computadores. La búsqueda paralela llega a ser un tema importante en IA y en informática teórica. Una introducción breve a la literatura de IA se puede encontrar en Mahanti y Daniels (1993).

EJERCICIOS

4.1 Trace cómo opera la búsqueda A* aplicada al problema de alcanzar Bucarest desde Lugoj utilizando la heurística distancia en línea recta. Es decir, muestre la secuencia de nodos que considerará el algoritmo y los valores f, g, y h para cada nodo.

4.2 El **algoritmo de camino heurístico** es una búsqueda primero el mejor en la cual la función objetivo es $f(n) = (2 - w)g(n) + wh(n)$. ¿Para qué valores del w está garantizado que el algoritmo sea óptimo? ¿Qué tipo de búsqueda realiza cuando $w = 0$?, ¿cuándo $w = 1$? y ¿cuándo $w = 2$?

4.3 Demuestre cada una de las declaraciones siguientes:

a) La búsqueda primero en anchura es un caso especial de la búsqueda de coste uniforme.

b) La búsqueda primero en anchura, búsqueda primero en profundidad, y la búsqueda de coste uniforme son casos especiales de la búsqueda primero el mejor.

c) La búsqueda de coste uniforme es un caso especial de la búsqueda A*.

4.4 Idee un espacio de estados en el cual A*, utilizando la BÚSQUEDA-GRAFO, devuelva una solución sub-óptima con una función $h(n)$ admisible pero inconsistente.

4.5 Vimos en la página 109 que la heurística de distancia en línea recta dirige la búsqueda primero el mejor voraz por mal camino en el problema de ir de Iasi a Fagaras. Sin embargo, la heurística es perfecta en el problema opuesto: ir de Fagaras a Iasi. ¿Hay problemas para los cuales la heurística engaña en ambas direcciones?

4.6 Invente una función heurística para el 8-puzle que a veces sobrestime, y muestre cómo puede conducir a una solución subóptima sobre un problema particular (puede utilizar un computador para ayudarse). Demuestre que, si h nunca sobrestima en más de c, A*, usando h, devuelve una solución cuyo coste excede de la solución óptima en no más de c.

4.7 Demuestre que si una heurística es consistente, debe ser admisible. Construya una heurística admisible que no sea consistente.

4.8 El problema del viajante de comercio (PVC) puede resolverse con la heurística del árbol mínimo (AM), utilizado para estimar el coste de completar un viaje, dado que ya se ha construido un viaje parcial. El coste de AM del conjunto de ciudades es

la suma más pequeña de los costos de los arcos de cualquier árbol que une todas las ciudades.

a) Muestre cómo puede obtenerse esta heurística a partir de una versión relajada del PVC.
b) Muestre que la heurística AM domina la distancia en línea recta.
c) Escriba un generador de problemas para ejemplos del PVC donde las ciudades están representadas por puntos aleatorios en el cuadrado unidad.
d) Encuentre un algoritmo eficiente en la literatura para construir el AM, y úselo con un algoritmo de búsqueda admisible para resolver los ejemplos del PVC.

4.9 En la página 122, definimos la relajación del 8-puzle en el cual una ficha podía moverse del cuadrado A al cuadrado B si B era el blanco. La solución exacta de este problema define la **heurística de Gaschnig** (Gaschnig, 1979). Explique por qué la heurística de Gaschnig es al menos tan exacta como h_1 (fichas mal colocadas), y muestre casos donde es más exacta que h_1 y que h_2 (distancia de Manhattan). ¿Puede sugerir un modo de calcular la heurística de Gaschnig de manera eficiente?

4.10 Dimos dos heurísticas sencillas para el 8-puzle: distancia de Manhattan y fichas mal colocadas. Varias heurísticas en la literatura pretendieron mejorarlas, *véase*, por ejemplo, Nilsson (1971), Mostow y Prieditis (1989), y Hansson *et al.* (1992). Pruebe estas mejoras, implementando las heurísticas y comparando el funcionamiento de los algoritmos que resultan.

4.11 Dé el nombre del algoritmo que resulta de cada uno de los casos siguientes:

a) Búsqueda de haz local con $k = 1$.
b) Búsqueda de haz local con $k = \infty$.
c) Temple simulado con $T = 0$ en cualquier momento.
d) Algoritmo genético con tamaño de la población $N = 1$.

4.12 A veces no hay una función de evaluación buena para un problema, pero hay un método de comparación bueno: un modo de decir si un nodo es mejor que el otro, sin adjudicar valores numéricos. Muestre que esto es suficiente para hacer una búsqueda primero el mejor. ¿Hay un análogo de A*?

4.13 Relacione la complejidad en tiempo de LRTA* con su complejidad en espacio.

4.14 Suponga que un agente está en un laberinto de 3x3 como el de la Figura 4.18. El agente sabe que su posición inicial es (1,1), que el objetivo está en (3,3), y que las cuatro acciones *Arriba, Abajo, Izquierda, Derecha* tienen sus efectos habituales a menos que estén bloqueadas por una pared. El agente *no* sabe dónde están las paredes internas. En cualquier estado, el agente percibe el conjunto de acciones legales; puede saber también si el estado ha sido visitado antes o si es un nuevo estado.

a) Explique cómo este problema de búsqueda *online* puede verse como una búsqueda *offline* en el espacio de estados de creencia, donde el estado de creencia inicial incluye todas las posibles configuraciones del entorno. ¿Cómo de grande es el estado de creencia inicial? ¿Cómo de grande es el espacio de estados de creencia?
b) ¿Cuántas percepciones distintas son posibles en el estado inicial?

c) Describa las primeras ramas de un plan de contingencia para este problema. ¿Cómo de grande (aproximadamente) es el plan completo?

Nótese que este plan de contingencia es una solución para *todos los entornos posibles* que encajan con la descripción dada. Por lo tanto, intercalar la búsqueda y la ejecución no es estrictamente necesario hasta en entornos desconocidos.

4.15 En este ejercicio, exploraremos el uso de los métodos de búsqueda local para resolver los PVCs del tipo definido en el Ejercicio 4.8.

a) Idee una aproximación de la ascensión de colinas para resolver los PVSs. Compare los resultados con soluciones óptimas obtenidas con el algoritmo A* con la heurística AM (Ejercicio 4.8).

b) Idee una aproximación del algoritmo genético al problema del viajante de comercio. Compare los resultados a las otras aproximaciones. Puede consultar Larrañaga *et al*. (1999) para algunas sugerencias sobre las representaciones.

4.16 Genere un número grande de ejemplos del 8-puzle y de las 8-reinas y resuélvalos (donde sea posible) por la ascensión de colinas (variantes de subida más escarpada y de la primera opción), ascensión de colinas con reinicio aleatorio, y temple simulado. Mida el coste de búsqueda y el porcentaje de problemas resueltos y represente éstos gráficamente contra el costo óptimo de solución. Comente sus resultados.

4.17 En este ejercicio, examinaremos la ascensión de colinas en el contexto de navegación de un robot, usando el entorno de la Figura 3.22 como un ejemplo.

a) Repita el Ejercicio 3.16 utilizando la ascensión de colinas. ¿Cae alguna vez su agente en un mínimo local? ¿Es *posible* con obstáculos convexos?

b) Construya un entorno no convexo poligonal en el cual el agente cae en mínimos locales.

c) Modifique el algoritmo de ascensión de colinas de modo que, en vez de hacer una búsqueda a profundidad 1 para decidir dónde ir, haga una búsqueda a profundidad-k. Debería encontrar el mejor camino de k-pasos y hacer un paso sobre el camino, y luego repetir el proceso.

d) ¿Hay algún k para el cual esté garantizado que el nuevo algoritmo se escape de mínimos locales?

e) Explique cómo LRTA* permite al agente escaparse de mínimos locales en este caso.

4.18 Compare el funcionamiento de A* y BRPM sobre un conjunto de problemas generados aleatoriamente en dominios del 8-puzle (con distancia de Manhattan) y del PVC (con AM, *véase* el Ejercicio 4.8). Discuta sus resultados. ¿Qué le pasa al funcionamiento de la BRPM cuando se le añade un pequeño número aleatorio a los valores heurísticos en el dominio del 8-puzle?