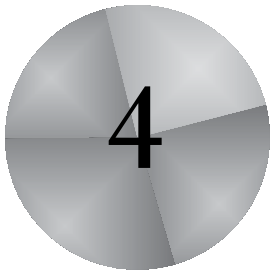

4 Búsqueda informada y exploración

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.

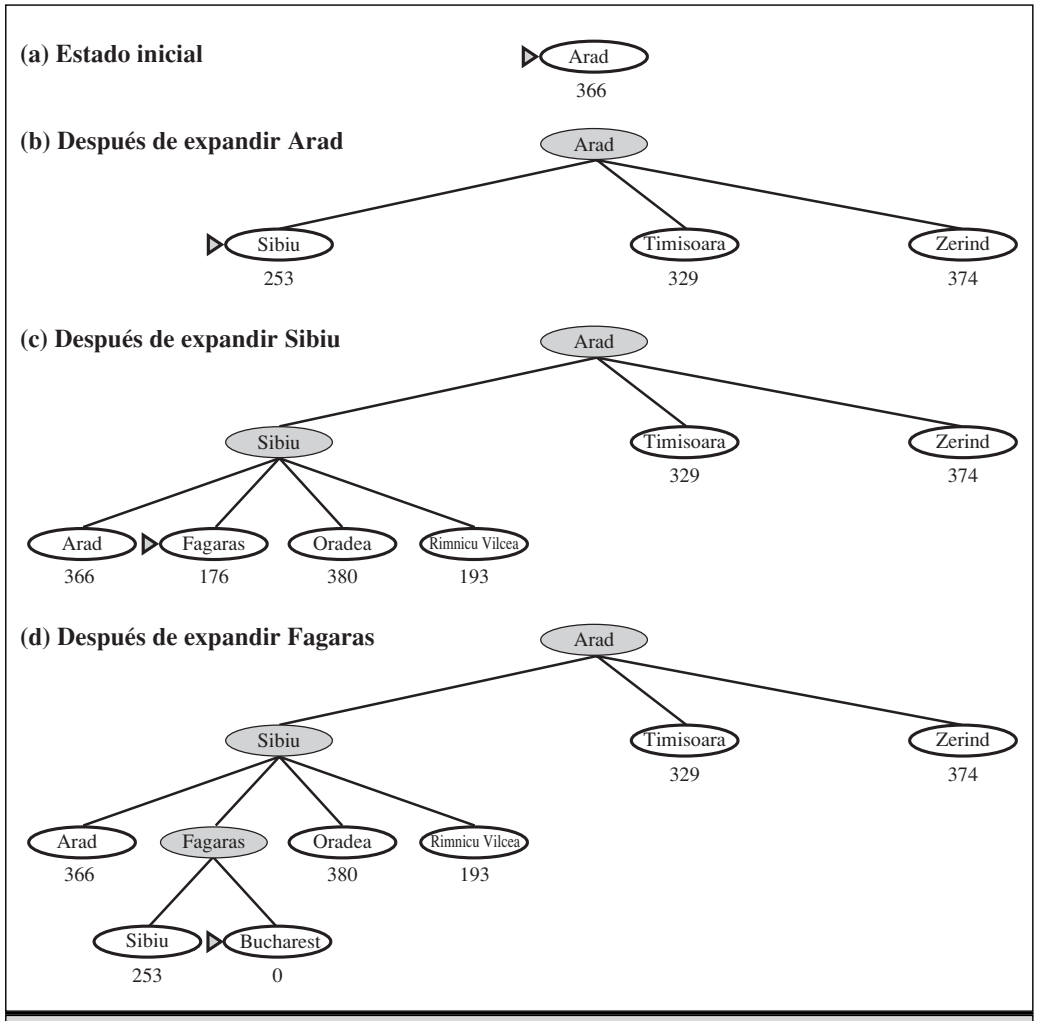

Figura 4.2 Etapas en una búsqueda primero el mejor avara para Bucarest utilizando la heurística distancia en línea recta h_{DLR}. Etiquetamos lo nodos con sus h-valores.

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 $h(n)$,

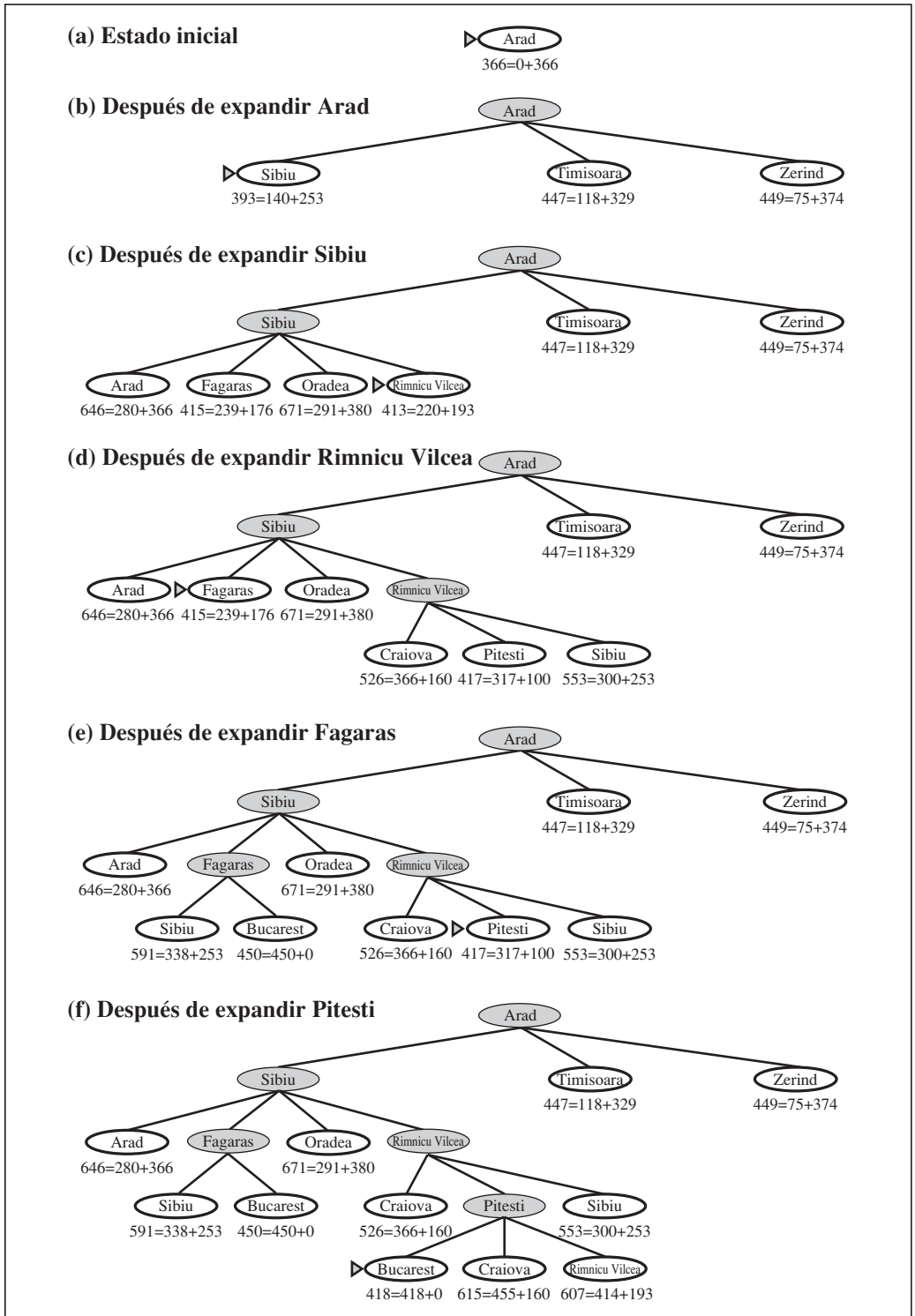

Figura 4.3 Etapas en una búsqueda A* para Bucarest. Etiquetamos lo nodos con $f = g + h$. Los valores h son las distancias en línea recta a Bucarest tomadas de la Figura 4.1.

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

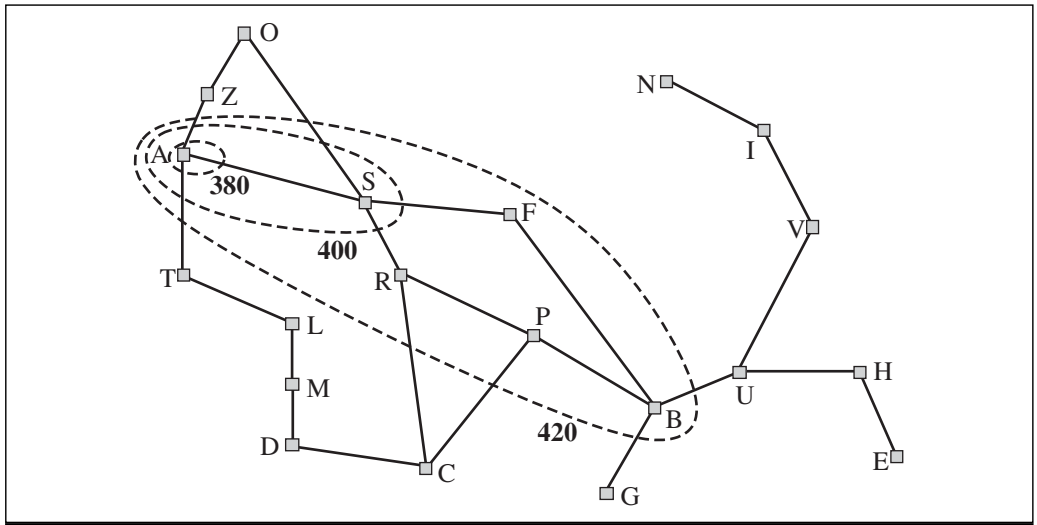

Figura 4.4 Mapa de Rumanía mostrando curvas de nivel en $f = 380$, $f = 400$ y $f = 420$, con Arad como estado inicial. Los nodos dentro de cada curva de nivel tienen f-costos menores o iguales al valor de la curva de nivel.

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

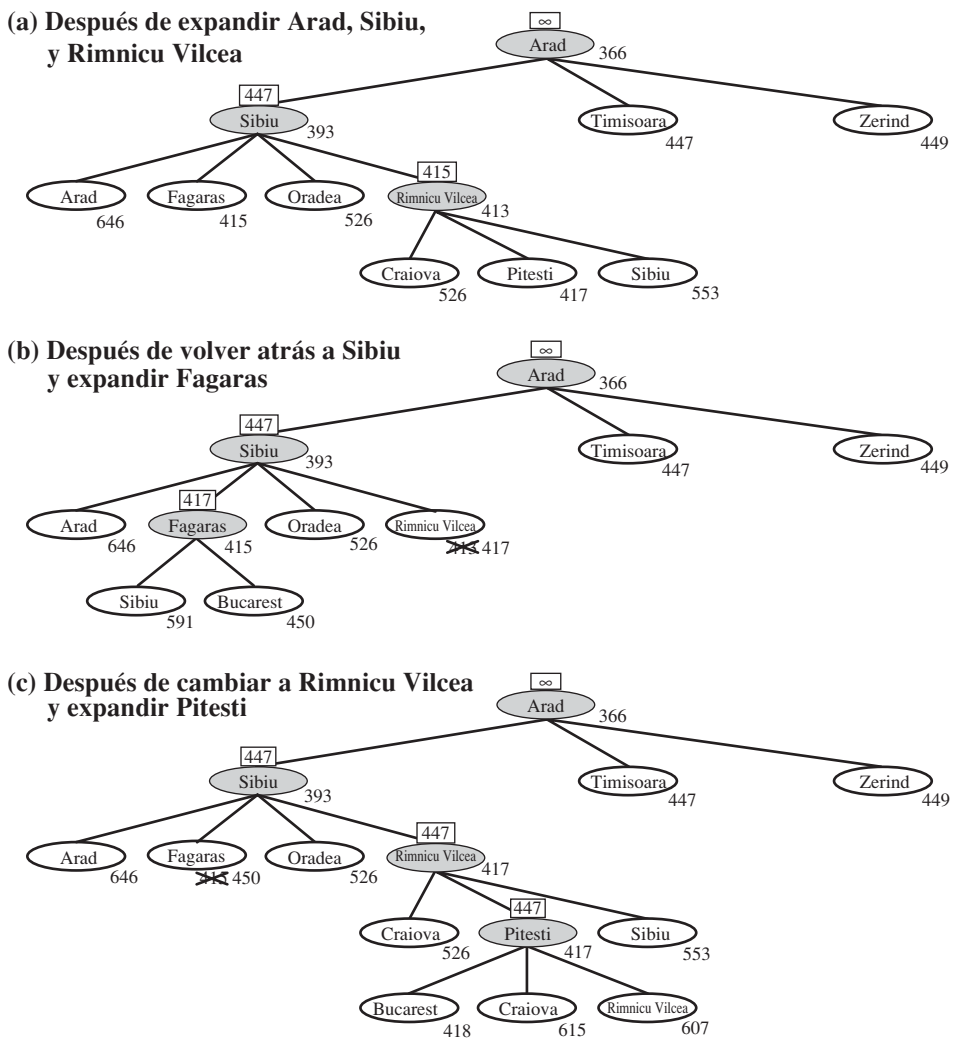

Figura 4.6 Etapas en una búsqueda BRPM para la ruta más corta a Bucarest. Se muestra el valor del *f*-límite para cada llamada recursiva sobre cada nodo actual. (a) Se sigue el camino vía Rimnicu Vilcea hasta que la mejor hoja actual (Pitesti) tenga un valor que es peor que el mejor camino alternativo (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).

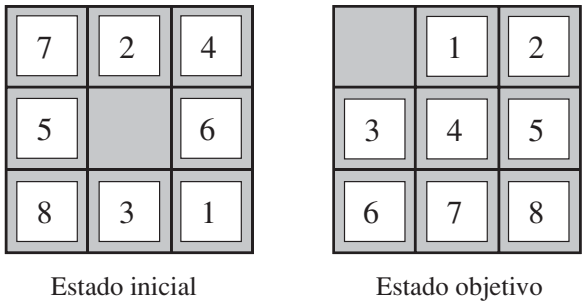

Figura 4.7 Un caso típico del 8-puzle. La solución tiene 26 pasos.

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-

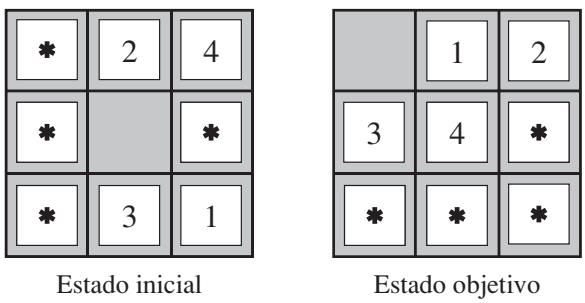

Figura 4.9 Un subproblema del puzle-8 dado en la Figura 4.7. La tarea es conseguir las fichas 1, 2, 3 y 4 en sus posiciones correctas, sin preocuparse de lo que le pasan a las otras fichas.

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.