Capítulo 1
BÚSQUEDA

Este capítulo describe cómo los algoritmos de búsqueda no informados se pueden mejorar con la ayuda de *información heurística*.

Los agentes basados en búsqueda mantienen un modelo simbólico de su entorno, en el que representan su situación actual, una serie de metas que se quieren alcanzar, así como las acciones que pueden realizar y sus efectos. Primeramente, buscan sobre este modelo simbólico una secuencia de acciones (también llamado *plan*) que, al menos en el modelo, modifica la situación actual de tal forma que se alcancen las metas. Sólo entonces intentan ejecutar las acciones del plan en el mundo real.

En este capítulo nos restringimos a agentes que actúan en los entornos más simples, que se ciernen a las siguientes características:

- *Discretos*: el cambio del entorno se puede concebir como una secuencia de estados (en particular, existe un estado actual y un conjunto de estados meta alternativos);
- *Accesibles*: toda la información relevante sobre un estado está disponible para el agente (en particular, un agente puede determinar inequívocamente el estado del entorno en el que se encuentra);
- *Estáticos*: el único ente que puede iniciar un cambio en el entorno es el propio agente, es decir, el entorno cambia exclusivamente por las acciones del agente;
- *Deterministas*: las acciones del agente en un estado actual determinan completamente el estado resultante, i.e., no hay incertidumbre sobre el resultado de las acciones (las acciones no pueden fallar, etc.).

Un ejemplo de estos entornos simples es el mundo de los bloques. En una mesa hay un conjunto de *n* bloques. Cada bloque puede estar o bien sobre la mesa o bien sobre otro bloque. Solo es relevante la posición vertical de los bloques, la posición horizontal no importa (por ejemplo, estando los bloques *A* y *B* en la mesa, es irrelevante si *A* está a la izquierda o a la derecha de *B*). El agente puede poner un bloque *A* sobre un bloque *B*, si tanto *A* como *B* están libres (no tienen otro bloque encima). Puede poner un bloque *A* sobre la mesa si *A* está libre (se supone que en la mesa siempre hay sitio para colocar un bloque más). El agente desea transformar la configuración inicial de los bloques (el estado actual) en una configuración diferente determinada (estado meta). Cada acción le supone el mismo coste (p. e., porque la descarga de la batería de un agente por emprender una acción es independiente de la acción concreta que realiza). Otros ejemplos de entornos de este tipo son las Torres de Hanoi, el 8-puzzle, o el problema de encontrar rutas en una red de carreteras.

El modelo usado para representar dichos entornos es el **espacio de estados**. Se puede concebir simplemente como un grafo dirigido y etiquetado, donde los *nodos* representan los estados del entorno, los *arcos* modelan las acciones que puede emprender el agente en un estado junto con sus efectos, y las *etiquetas* de los arcos indican el coste de la acción correspondiente. Normalmente, el tamaño del grafo (el número de nodos) crece de forma exponencial en las características del entorno (por ejemplo, el número *n* de bloques en el mundo de los bloques), dando lugar a estructuras de datos para representar el grafo que pueden tener un tamaño potencialmente inviable. Suponemos que el diseñador dota al agente con el siguiente *conocimiento a priori*, que representa el grafo del espacio de estados de forma *implícita*:

- Un estado inicial $\boldsymbol{s_0} \in S$, siendo *S* el conjunto de estados del entorno;
- Una función ***expandir***: $s \rightarrow \{s_{i1}, \ldots s_{im}\}$ que asigna a cada estado *s* un conjunto finito de estados sucesores (y, por tanto, representa implícitamente la estructura del grafo antes mencionado);
- Una función booleana ***meta***: $s \rightarrow \{true, false\}$ que, para cada estado s, determina si se trata de un estado meta o no;
- Una función de coste ***c***: $(s,s') \rightarrow v$, $v \in \aleph$ que, para cada acción que lleva de un estado *s* a *s'*, determina el coste *v* como número natural positivo.

El agente utiliza un algoritmo de búsqueda *genérico*, capaz de encontrar un *plan* (es decir, una secuencia de acciones que lleva del estado inicial a un estado meta) para *cualquier* entorno que se pueda modelizar a través de un

espacio de estados. Con tal fin, partiendo del estado inicial, va generando un **árbol de búsqueda**, siguiendo la siguiente plantilla algorítmica, que utiliza el conocimiento a priori arriba indicado (subrayado en el pseudocódigo):

```
abierta ← s0
Repetir
        Si vacía?(abierta) entonces
                devolver(negativo)   % no se encuentran nodos meta
        nodo ← primero(abierta)
        Si meta?(nodo) entonces
                devolver(nodo)   % se ha encontrado un nodo meta
        sucesores ←expandir(nodo)
        Para cada n∈sucesores hacer
                n.padre← nodo
                ordInsertar(n,abierta,<orden>)
Fin {repetir}
```

Nótese que el árbol de búsqueda se representa a través de registros llamados *nodo* de un tipo que, aparte de la información propia del estado que representan, contiene un puntero al nodo padre. La lista *abierta* contiene los nodos hoja del árbol que quedan por tratar. *Primero* devuelve el primer elemento de *abierta*, borrándolo de dicha lista. *OrdInsertar* añade un nuevo nodo *n* a la lista *abierta*, de modo que ésta queda ordenada de acuerdo con la función <orden>. Dependiendo de qué función de orden se utiliza, la plantilla se instancia en diferentes algoritmos de búsqueda. Estamos interesados en algoritmos que sean:

- **completos**, es decir, siempre encuentran un camino a un nodo meta si existe uno, y
- **óptimos**, es decir, si se encuentra un camino a un nodo meta, éste es el camino de menor coste.

Asimismo, son relevantes las complejidades de los algoritmos en tiempo y en espacio.

La plantilla algorítmica permite comparar diferentes algoritmos de búsqueda de forma sencilla e intuitiva. Su desventaja es el hecho de que el mismo estado puede entrar varias veces en el árbol de búsqueda, lo cual puede conllevar a un comportamiento ineficiente del algoritmo, puesto que los subárboles que se van generando a partir de estados repetidos son idénticos. Este problema se puede atacar, añadiendo un "filtro" a la función *expandir*. Por ejemplo, si se filtran **ciclos simples**, el resultado de la función *expandir*

para un estado *s* no contendrá el nodo padre de *s* (es decir, no se consideraría un caso en el que inmediatamente después de realizar una acción se pretende efectuar la acción inversa). Para filtrar **todos los estados repetidos**, habría que evitar que el resultado de *expandir* contenga cualquier nodo que ya se encuentre en el árbol de búsqueda. Qué filtro se debe aplicar depende del dominio, y en los ejercicios de este libro suele estar claramente indicado. Nótese que en la literatura existen variaciones de la plantilla algorítmica aquí presentada que genera un *grafo de búsqueda*, y así obvia este problema.

Los **métodos de búsqueda no informados** utilizan únicamente los conocimientos a priori antes mencionados en el proceso de búsqueda. Posiblemente el método más sencillo sea la **búsqueda en amplitud** (*breadth first search*), que va generando el árbol de búsqueda por niveles (no se expande ninguna hoja de nivel *i* antes de haber expandido todas las hojas del nivel *i*–1). Este comportamiento se consigue con la plantilla algorítmica, implementando *ordInsertar* de tal forma que añade nuevos nodos siempre *al final* de la lista *abierta*. La búsqueda en amplitud es completa, pero sólo es óptima en caso de que todas las acciones tengan el mismo coste (como, por ejemplo, en el caso del mundo de los bloques). Asimismo, el número de nodos en el árbol de búsqueda generado por la búsqueda en amplitud crece de forma exponencial, en función del nivel de profundidad del mejor nodo meta. Desde esta perspectiva, su complejidad es exponencial tanto en tiempo como en espacio. La **búsqueda de coste uniforme** (*uniform cost search*) generaliza la búsqueda en amplitud, para que también sea óptima en dominios en los que las acciones llevan asociados costes diferentes (como, por ejemplo, en el problema de buscar rutas en una red de carreteras, donde el coste de la acción de ir de una ciudad a una ciudad vecina es proporcional a la distancia en kilómetros entre estas ciudades). Se define una **función *g*** que, para cada nodo *n* del árbol de búsqueda, compila el coste total de llegar desde el estado inicial a *n*, es decir, $g(n)$ es la suma de los costes de las acciones que llevan de s_0 a *n*. Para realizar el comportamiento de la búsqueda de coste uniforme, que entre todos los nodos hoja del árbol de búsqueda siempre elige aquél de menor valor de *g* para expandir, *ordInsertar* se implementa de modo que la lista *abierta* siempre esté ordenada de forma ascendente con respecto a *g*. Nótese que, si el coste de todas las acciones del espacio de estados es constante, la búsqueda de coste uniforme degenera en la búsqueda en amplitud. Por tanto, en el peor caso, hereda todas sus características negativas (complejidad exponencial en tiempo y espacio).

Los **métodos de búsqueda heurísticos** utilizan información adicional para mejorar la complejidad de los métodos no informados, manteniendo completitud y optimalidad. Con tal fin, disponen de una **función heurística** $\boldsymbol{h^*}$ que, para cada nodo n del espacio de estados, estima el coste del camino más corto desde n a un estado meta (representamos con $h(n)$ el coste real de este camino y con $h^*(n)$ el coste estimado). Podríamos guiar el algoritmo de búsqueda por h^*, eligiendo siempre para expandir el nodo hoja n de menor valor $h^*(n)$. Pero entonces simplemente preferiríamos en el proceso de búsqueda a los nodos que supuestamente estén más cerca de un nodo meta, sin importarnos el coste en el que incurrimos para llegar a estos nodos, lo cual puede comprometer completitud y optimalidad del método. En consecuencia, definimos una función f^* que, para cada nodo n, suma el coste de llegar desde el estado inicial s_0 hasta n, y el coste estimado para llegar desde n a un nodo meta, es decir:

$$\forall n \in S: f^*(n) = g(n) + h^*(n).$$

La **búsqueda** $\boldsymbol{A^*}$ ($A^*search$) siempre expande el nodo hoja del árbol de búsqueda de menor valor de f^*. Se implementa instanciando la plantilla algorítmica presentada de modo que *ordInsertar* mantenga la lista *abierta* siempre ordenada de forma ascendente con respecto a f^*.

Nos interesan dos clases de funciones heurísticas. Una función heurística h^* es

- **optimista** si para todo nodo $n \in S$ se cumple que $h^*(n) \leq h(n)$; y
- **consistente** si para todo nodo $n_i \in S$ y todo sucesor directo $n_j \in S$ de n_i se cumple que $h^*(n_i) - h^*(n_j) \leq c(n_i, n_j)$, siendo $c(n_i, n_j)$ el coste real de avanzar de n_i a n_j

Intuitivamente, una función heurística optimista siempre subestima el coste *total* de llegar desde cualquier estado n al nodo meta más cercano, mientras que una función heurística consistente también subestima el coste de todos los pasos (acciones) para llegar de n al nodo meta más cercano. Por tanto, una función heurística consistente también es optimista, pero no necesariamente viceversa.

Se puede demostrar que, si h^* es consistente, entonces los valores de f^* crecen débilmente a lo largo de todos los caminos del árbol de búsqueda. Esto significa que, antes de expandir un nodo hoja n, la búsqueda A^* habrá expandido todos los nodos del espacio de estados cuyo valor de f^* sea menor de $f^*(n)$. Nótese que, en este caso, al pasar del nodo n a su sucesor n', los

valores de f^* crecen mucho menos si *n*' está más cerca de la meta (crece *g* pero disminuye h^*) que si n' está más alejado de la meta (crecen tanto *g* como h^*). En el ejemplo de la búsqueda del camino más corto en un mapa de carreteras, esto significa que el conjunto de ciudades expandidas en cada momento forma un "haz" sesgado hacia el nodo meta (lo cual es similar a como buscarían las personas: para ir de Madrid a Sevilla se podría dudar en coger una ruta vía Córdoba o vía Badajoz, pero probablemente no se consideraría una ruta a través de Valladolid). En cambio, la búsqueda de coste uniforme, al no disponer de información heurística sobre la "dirección" de la meta, expandiría nodos en círculos concéntricos alrededor de la ciudad de partida.

Uno de los teoremas principales de este tema determina que es suficiente que h^* sea optimista para que la búsqueda A^* sea óptima. Otro pregona que la búsqueda A^* es siempre completa (si, como en nuestro caso, los costes de los operadores son números naturales positivos).

Una forma de generar funciones heurísticas es mediante aprendizaje automático. Un mecanismo simple se define como sigue: se parte de la función heurística $h^*(n)=0$ para todo estado $n \in S$ la cual, según la definición, es optimista en *cualquier* entorno. Luego, cada vez que se haya expandido un nodo n_i en el marco del algoritmo A^*, se aprende un nuevo valor de h^* para n_i siguiendo la siguiente **regla de aprendizaje**:

$$h^*(n_i) \leftarrow \min_{n_j \in expandir(n_i)} \left[h^*(n_j) + c(n_i, n_j) \right]$$

Dicha regla calcula estimaciones de los costes para llegar desde n_i al nodo meta más cercano a través de cada uno de los sucesores n_j de n_i, y luego se queda con el coste mínimo como nuevo valor $h^*(n_i)$. Nótese que, por construcción de la regla de aprendizaje, si los valores de h^* a la derecha de la asignación son optimistas, entonces el valor aprendido para $h^*(n_i)$ también lo es.

Las funciones heurísticas optimistas también se pueden diseñar. Un método simple de diseño parte de la noción de *problemas relajados*. Un problema relajado es idéntico al problema original en cuanto al conjunto de estados, estado inicial, estados meta, y coste de acciones, pero aparte de las acciones permitidas en el problema inicial, en el problema relajado se dispone de algunas acciones adicionales. Por ejemplo, un problema relajado para el 8-puzzle sería aquel en el que las fichas se pueden mover a cualquier casilla adyacente, y no solamente a una casilla adyacente que está vacía, como es el

caso en el problema original. El coste real h para llegar de un estado n al estado meta más cercano en el problema relajado, por construcción, es menor o igual al coste real h en el problema original. Por tanto, si los valores de h se pueden computar fácilmente para los estados del problema relajado, éstos pueden servir como función heurística h^* optimista en el problema original. En el problema relajado del 8-puzzle arriba mencionado, el coste $h(n)$ de un estado n sería la suma de las distancias de Manhattan entre la posición de cada ficha en el estado n y en el estado meta. Dicha función sería una función heurística h^* optimista si se aplicara al problema original (el 8-puzzle "no relajado").

Intuitivamente, una función heurística optimista h^* es "mejor" que otra si su *error heurístico* $\varepsilon = h(n) - h^*(n)$ es menor. La siguiente definición generaliza esta noción: Sean h_1^* y h_2^* dos funciones heurísticas optimistas; h_1^* es *más informada* que h_2^*, si para todo nodo n se cumple que $h_1^*(n) \geq h_2^*(n)$. Se puede demostrar que, en este caso, un algoritmo A^* que usa la "mala" función heurística h_2^* expande *al menos* tantos nodos como un algoritmo A^* que usa la "buena" función heurística h_1^*. Por tanto, si se dispone de varias funciones heurísticas optimistas, todas ellas de fácil evaluación, en cada nodo conviene quedarse con el mayor valor de todas ellas.

A no ser que se puedan asegurar límites muy estrictos al error heurístico, no es posible evitar que el árbol de búsqueda generado por la búsqueda A^* crezca de forma exponencial en el *peor caso*. Sin embargo, se puede demostrar de forma experimental que en el *caso medio* se consigue una mejora muy significativa de la complejidad, tanto en tiempo como en espacio, con respecto a métodos de búsqueda no informados, como por ejemplo la búsqueda de coste uniforme. Nótese que, siempre y cuando h^* sea optimista, se consigue todo esto sin prescindir de las garantías de completitud y optimalidad.

Una forma de extender la búsqueda A^* para que sea aplicable a entornos aun más complejos es mediante *heurísticas fuertes*, es decir, descartando las garantías de completitud y optimalidad. La **búsqueda por islas** (*island-driven search*) presupone que el agente conoce una serie de "estados isla" $i_1,\ldots,i_n$ por los que el proceso de búsqueda tendrá que pasar necesariamente. Entonces, en vez de realizar una única búsqueda A^* "de larga distancia" desde s_0 a la meta s_m, se realizan n+1 búsquedas A^* "de corta distancia": de s_0 a i_1; de i_1 a i_2; …; y de i_n a s_m. Nótese que la suma del número de nodos en los n+1 árboles de búsqueda "de corta distancia" es muy inferior a los nodos

contenidos en el árbol generado por la búsqueda A^* "de larga distancia". Por ejemplo, en el mundo de los bloques, un estado isla sería aquel en el que todos los bloques están sobre la mesa. Muy a menudo, con esta heurística fuerte se encuentra el mejor plan, pero tampoco es difícil construir un ejemplo en el que el mejor plan no pasa por el estado isla mencionado.

Otro enfoque para atacar entornos más complejos y difíciles es mediante una clase de métodos llamada "búsqueda en línea" (*online search*). En vez de buscar un plan de actuación completo sobre el modelo del entorno que mantiene el agente, y seguidamente ejecutar dicho plan en el entorno real, estos algoritmos "engranan" ambas fases. La búsqueda con horizonte (*limited-horizon search*), por ejemplo, se basa en la siguiente heurística fuerte: en el modelo, se genera un árbol de búsqueda sólo hasta un cierto limite de profundidad k (el "horizonte"), y se asume que el camino que lleva al nodo de menor valor de f^* a profundidad k es, en este instante, el camino más prometedor. Se ejecuta la *primera acción* de dicho camino en el mundo real, y se vuelve a realizar una nueva búsqueda hasta un horizonte k. Se repite este procedimiento hasta que el agente se encuentre en un estado meta del entorno real.

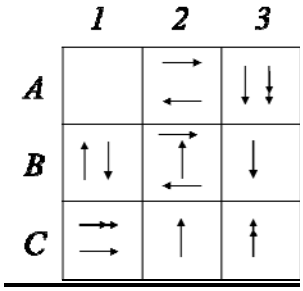
La implementación de dicho método se realiza en base a *búsquedas en profundidad*, limitadas al nivel k, puesto que así se puede aprovechar la menor complejidad en espacio de dicho algoritmo (sólo hay que mantener en memoria los nodos de un "camino actual" y sus hermanos). Además, si h^* es consistente y, por tanto, f^* crece de forma débilmente monótona a lo largo de todos los caminos del árbol de búsqueda, se puede aplicar una técnica denominada *poda* α para que la búsqueda en profundidad pueda abandonar ciertos caminos antes de llegar al nivel de profundidad k, sin que ello influya en el resultado del algoritmo.

Ejercicio 1.1

El entorno de un agente es un tablero 3×3, en cuyas 9 casillas (A1, A2, A3, B1, B2, B3, C1, C2, C3) puede haber una o varias flechas con cuatro orientaciones posibles: norte (N), sur (S), este (E) y oeste (O). El agente avanza en la dirección que marcan las flechas: las flechas simples hacen avanzar una casilla y las dobles dos. Cuando hay varias flechas en la misma casilla, el agente puede elegir entre sus respectivas direcciones. Cuando no hay flechas, ha de quedarse en la casilla. El coste de cada acción es 1 (independiente del número de casillas que se avanza)

Considere el tablero que se muestra a continuación. Considere que el agente se encuentra en la casilla central (B2). Saliendo de ella, desea encontrar el camino más corto para volver a la casilla central.

a) Resuma el conjunto de estados posibles, y defina la función *expandir* (es decir, su imagen para cada uno de los estados)
b) Desarrolle el árbol de búsqueda generado por la búsqueda en amplitud, asumiendo que se filtran *todos* los estados repetidos (excepto el estado meta, por supuesto). Además suponga que, cuando hay varias alternativas, el agente tiene las siguientes preferencias para sus movimientos: E–S–N–O. Indique el orden en el que se expanden los nodos, así como el estado de la lista *abierta* en cada paso del algoritmo.



Ejercicio 1.2

Considere el siguiente problema:

> *Un hombre se encuentra en la orilla izquierda de un río junto con un lobo, una oveja y una col. Quiere cruzar el río llevando consigo el lobo, la oveja y la col. En la barca sólo hay dos plazas, una de las cuales debe ir ocupada por el hombre. Cada uno de los restantes pasajeros (lobo, oveja, col) ocupa una plaza, de tal modo que sólo uno puede acompañar al hombre en cada viaje. Además, no puede dejar solos en una orilla al lobo con la oveja, ni a la oveja con la col (ni los tres), porque la primera se comería a la segunda en cada paso*

Suponga que se modela el problema como un espacio de estados, donde cada estado se describe como un par de conjuntos, indicando quien(es) se encuentran en cada orilla del río (*c* = col; *o* = oveja; *l* = lobo; *h* = hombre). Por tanto, el estado inicial del problema sería ({*c*,*o*,*l*,*h*},{}), y el estado meta ({},{*c*,*o*,*l*,*h*}).

a) Simule la estrategia de *búsqueda en profundidad* para este problema, asumiendo que se filtran *todos* los estados repetidos. Expanda los sucesores de los nodos siguiendo las preferencias del hombre, las cuales se ordenan (de mayor a menor) como sigue: viajar con la oveja, viajar con la col, viajar sólo, viajar con el lobo. Dibuje el árbol de búsqueda correspondiente e indique el orden en el que se exploran los nodos.

b) Suponga ahora que no se filtra ningún estado repetido. ¿La búsqueda en profundidad sigue siendo completa? Discuta *brevemente* la adecuación de otros métodos de búsqueda no informados para este caso con respecta a completitud, complejidad en tiempo y complejidad en espacio.

Ejercicio 1.3

En el problema de las Torres de Hanoi hay tres agujas (*A*, *B* y *C*) y *n* discos de distintos tamaños que reposan sobre la aguja *A* (*D1*, *D2*, ..., *Dn*). Se trata de pasar la torre de discos de la aguja *A* bien a la aguja *B o* bien a la aguja *C*. En cada operación es posible mover un disco de una aguja a otra, siempre y cuando dicho disco no se coloque sobre un disco de menor tamaño. A continuación se muestra el estado inicial y un estado meta para el caso de *n*=3 discos.

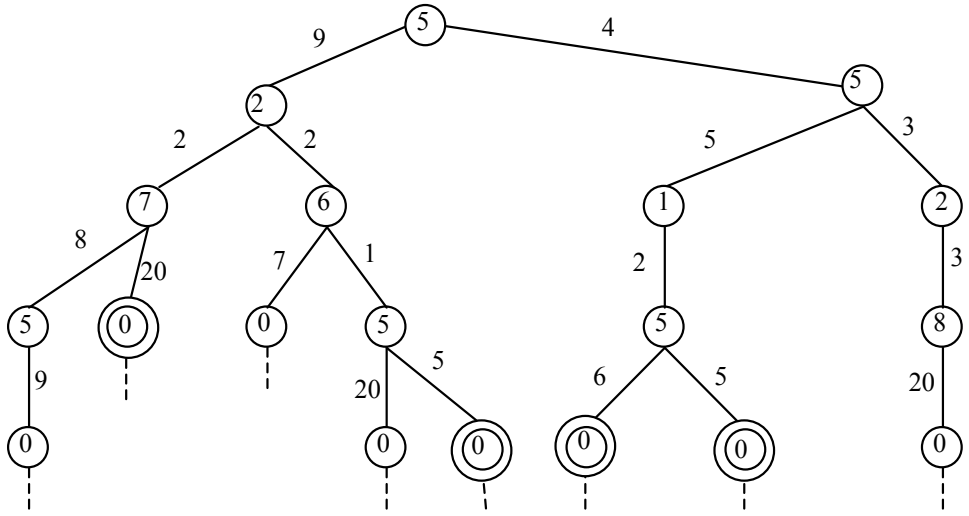
a) Dos estados se consideran *equivalentes*, si están los mismos discos en la aguja *A* mientras que los discos de las agujas *B* y *C* están intercambiadas. Desarrolle el árbol de búsqueda generado por la búsqueda en amplitud para el problema arriba indicado (*n*=3), asumiendo que se filtran todos los estados *repetidos* y todos los estados *equivalentes*. Indique el orden en el que se expanden los nodos, así como el estado de la lista *abierta* en cada paso del algoritmo.

b) Filtrando los estados repetidos y equivalentes, ¿la búsqueda en profundidad encontraría siempre una solución al problema general de las Torres de Hanoi (es decir: para cualquier *n* fijo) ? Explique *brevemente* el por qué de su posición.

c) Diseñe una función heurística h^* optimista para el problema general de las Torres de Hanoi (cuanto más informado mejor) ¿Suponiendo que no se filtran los estados repetidos y equivalentes, el algoritmo A^* con esta función heurística h^* encontraría siempre la solución óptima al problema de las Torres de Hanoi? Justifique *brevemente* su opinión.

Ejercicio 1.4

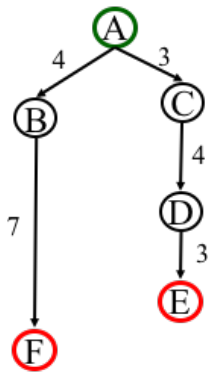
Considera el siguiente subárbol de un problema de búsqueda. Los números asignados a cada arco representan los costes de las operaciones/acciones correspondientes. Los números en los nodos representan una estimación del coste del camino más corto de este nodo a un nodo meta. Los nodos meta están marcados con doble círculo.



Construye el árbol que expandiría el algoritmo A^* aplicado a este problema, indica el orden en el que se expandirían los nodos, los valores de la función f^* y el nodo meta que el algoritmo encontraría.

Ejercicio 1.5

El grafo que se muestra a continuación describe un problema de búsqueda. El nodo *A* representa el estado inicial. Los nodos *F* y *E* son los estados meta. Los arcos están etiquetados con el coste real de los operadores.



a) Suponga la función h^* que se indica en la tabla siguiente. Aplique el algoritmo A^*, indicando el orden en el que se expanden los nodos y los valores de g, h^*, y f^* de cada nodo del árbol de búsqueda. ¿Se encuentra el mejor nodo meta? ¿La función h^* es optimista y/o consistente?

	A	B	C	D	E	F
h^*	8	6	6	5	0	0

b) Defina una nueva función heurística h_2^* que resulte ser optimista y *no consistente*, modificando alguno de los valores de la tabla anterior, y demuéstrelo formalmente.

c) Vuelva a aplicar el algoritmo A^* del apartado a) pero usando la función heurística h_2^* del apartado b). ¿Se encuentra ahora el mejor nodo meta?

Ejercicio 1.6

Considere el problema en el mundo de los bloques cuyo estado inicial y estado meta se muestran en la siguiente figura:

Estado Inicial Estado Meta

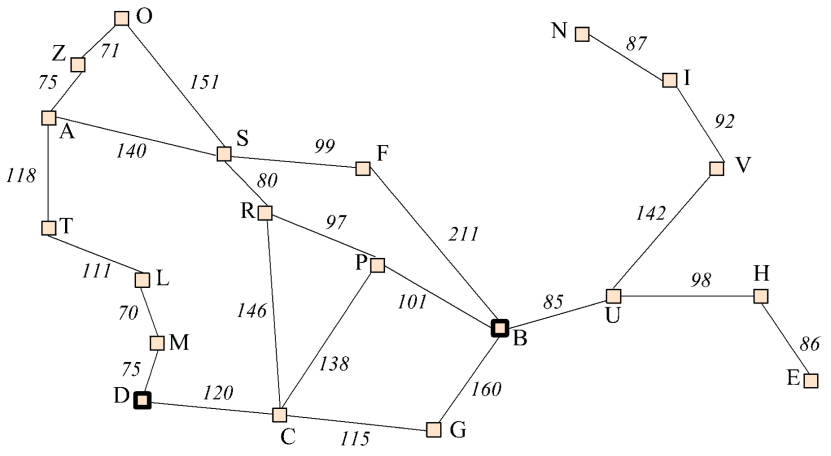
Desarrolle el árbol de búsqueda que expande el algoritmo A^* filtrando los ciclos simples, y utilizando la siguiente heurística:

$h^*(n)$ = número de bloques *descolocados*

Con tal fin, considere que un bloque está descolocado si por debajo no tiene el elemento correcto (bien el bloque deseado o bien la mesa). Indique el orden de expansión de los estados y muestre en cada paso los valores de g, h^*, y f^*. Suponga que el coste real de cada operador es 1.

Ejercicio 1.7

El grafo que se muestra a continuación representa un esquema de un mapa de carreteras. Los nodos están etiquetados con el nombre de ciudades, y los arcos con la distancia por carretera entre dichas ciudades. Inicialmente nuestro agente se encuentra en la ciudad *D*, y procura trasladarse por carretera a la ciudad *B* por el camino más corto. Con tal fin, puede hacer uso de su conocimiento de la región, y en particular de la distancia aérea entre ciudades. La función h^* que se muestra al lado indica la distancia aérea entre cualquier ciudad y la ciudad de destino *B*.



	h^*
A	366
B	0
C	160
D	242
E	161
F	178
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	98
R	193
S	253
T	329
U	80
V	199
Z	374

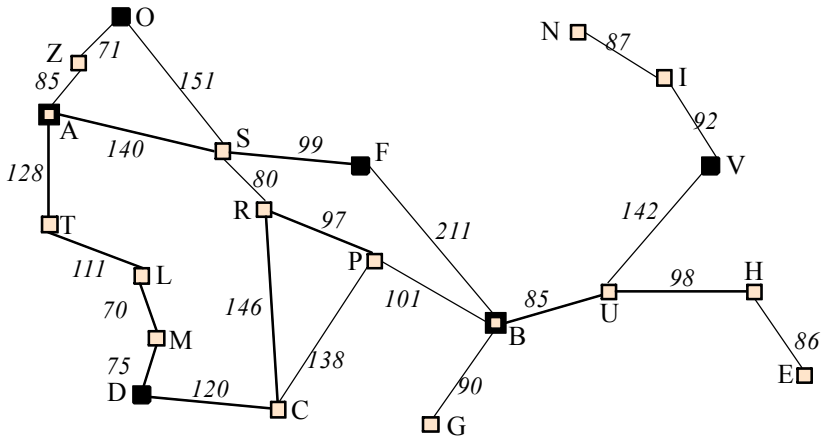
Suponga que el programa de agente se basa en el algoritmo A^* para resolver este problema.

a) Desarrolle el árbol de búsqueda que genera la búsqueda A^*, asumiendo que se filtran ciclos simples (p.e.: *D-M-D*). Indique el orden en el que se expanden los nodos, así como el valor de f^* de cada nodo del árbol de búsqueda.

b) Indique el estado de la lista *abierta* en cada paso del algoritmo.

Ejercicio 1.8

El grafo que se muestra a continuación representa un esquema de un mapa de carreteras. Los nodos están etiquetados con el nombre de ciudades, y los arcos con la distancia por carretera entre dichas ciudades. La función h^* que se muestra al lado indica la distancia aérea entre cualquier ciudad y la ciudad de *B*. Suponga que nuestro agente dispone de un coche eléctrico con una novedosa batería, que le permite recorrer 320km sin recargar. Sólo puede recargar en ciudades con estación de recarga pública, los cuales se indican en el mapa por los cuadrados rellenos (ciudades *D*, *F*, *O*, y *V*). Inicialmente el agente se encuentra en la ciudad *A* con la batería completamente cargada, y desea trasladarse a *B* por la ruta más corta posible (e.d. sin quedarse tirado por el camino con la batería vacía). Conoce la red de carreteras, la posición de las estaciones de recarga, y las características de su vehículo, por lo que en cada momento sabe a qué ciudades vecinas puede ir dado el estado de carga de su batería. El agente decide aplicar la búsqueda A^* para resolver el problema.



	h^*
A	366
B	0
C	160
D	242
E	161
F	178
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	98
R	193
S	253
T	329
U	80
V	199
Z	374

a) Represente el problema como espacio de estados, definiendo el conjunto de estados S, el estado inicial s_0, los estados meta (a través de la función *meta?*), el coste $c(s_i,s_j)$ para ir de una ciudad s_i a la ciudad vecina s_j, así como la función *expandir* (para dichas definiciones, puede suponer que exista un predicado binario *adyacente* que indica que dos ciudades son vecinas, un predicado unario *carga* que indica si hay una estación de recarga en una ciudad, y una función binaria *d* que proporciona la distancia entre dos ciudades vecinas de acuerdo con el mapa).
b) Desarrolle el árbol de búsqueda que genera el algoritmo A^* (no se filtran ciclos de ningún tipo). Indique el orden en el que se expanden los nodos, los valores de g, h^*, y f^* de cada nodo del árbol de búsqueda, así como el estado de la lista *abierta* en cada paso del algoritmo.
c) ¿Para la representación del apartado (a), la función heurística h^* (distancia área entre ciudades) sería optimista? Justifique brevemente su opinión.

Ejercicio 1.9

En juego de los "cuadrados latinos" se parte de un tablero 3 × 3 vacío. En cada posición vamos colocando números del 1 al 9, ninguno de los cuales se puede repetir. El objetivo es tener el tablero completo, es decir, un número en cada posición del mismo, y es necesario que el valor de la suma de las filas, columnas y diagonales sea siempre el mismo valor: 15. Un ejemplo en el cual tenemos el tablero completo, se han utilizado todos los números, pero no se consigue el objetivo indicado está en la siguiente figura. En este ejemplo sólo una diagonal y la última fila cumplen que la suma de sus números es 15.

1	9	2
7	5	6
8	4	3

a) Defina una representación eficiente para el juego de los cuadrados latinos 3×3, especificando el conjunto de estados, el estado inicial, y las operaciones permitidos en cada estado.

b) Considere el estado inicial que se muestra a continuación donde tenemos seis posiciones ocupadas, tres libres, y sólo una diagonal cumple que la suma de sus números es 15. Cada paso tiene coste uno.

2		4
	5	3
6	1	

Asimismo, considere la siguiente función heurística definida para cualquier estado *n* del tablero:

$h^*(n)$ = cantidad de filas + cantidad de columnas + cantidad de diagonales en el estado *n* que no cumplen que la suma de sus números es 15.

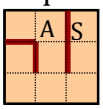
Por ejemplo, el valor de h^* de la configuración de la primera figura es 6, mientras que para la configuración de la segunda figura es 7.

Desarrolle el árbol de búsqueda que genera el algoritmo A^* para este problema. Indique el orden en el que se expanden los nodos, los valores de g, h^* y f^* para cada nodo del árbol de búsqueda, y la evolución de la lista *abierta*.

c) ¿La función h^* es optimista y/o consistente? ¿El algoritmo A^* encuentra siempre la solución de menor coste? Razone brevemente sus respuestas.

Ejercicio 1.10

Considere el problema el laberinto que se presenta en la siguiente figura.

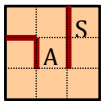


El agente A tiene el objetivo de encontrar la salida S. Las únicas acciones de las que el agente dispone son los movimientos (derecha, arriba, abajo, e izquierda) desde el cuadrado en el que se encuentra en un momento dado a un cuadrado adyacente. Sin embargo, cada una de estas acciones sólo es posible si en la dirección correspondiente no existe una barrera ni se saldría del tablero. Cada acción tiene un coste de una unidad. De antemano, el agente conoce el mapa del laberinto y la posición de la salida, pero no las posiciones de las barreras. Además el agente es capaz de identificar en cada momento su propia posición en el mapa.

a) Define una función heurística h^* para este problema. ¿Es su función h^* optimista y/o consistente? ¡Justifique brevemente su propuesta!

b) Suponiendo la posición inicial del agente que se indica en la figura arriba, aplique la *búsqueda* A^* con su función heurística h^* a este problema. Desarrolle el árbol de búsqueda suponiendo que no se evitan estados repetidos. Indique el orden en el que se expanden los nodos, así como los valores de g, h^* y f^* para cada nodo del árbol de búsqueda.

c) Suponga el siguiente estado inicial que se muestra a continuación. El agente no dispone de información heurística inicial respecto a las distancias de las distintas posiciones en el tablero hacía la salida.



Aplique la *búsqueda* A^* con el aprendizaje de la función heurística h^* a este problema. Desarrolle el árbol de búsqueda suponiendo que no se evitan estados repetidos. Indique el orden en el que se expanden los nodos, así como los valores de g, h^* y f^* para cada nodo del árbol de búsqueda. Además, anote los valores de la función heurística h^* de cada nodo en una tabla de tal modo que se aprecian los cambios de estos valores a lo largo de la ejecución del algoritmo. Si al expandir un nodo hay que elegir aleatoriamente entre varios, expanda preferiblemente primero el nodo que está más cerca de un nodo meta.