Técnicas basadas en búsquedas heurísticas

Contenido

Algoritmo de búsqueda primero el mejor

El algoritmo de búsqueda primero el mejor, o BF (Best First), es una especialización del algoritmo general de búsqueda en grafos en el que se utiliza una función de evaluación de los nodos, f, tal que para cada nodo n, $f(n)$ da un valor numérico que indica una medida de lo prometedor que es el nodo para ser expandido. Esta función se utiliza para ordenar la lista *ABIERTA*, de modo que aquellos nodos más prometedores estarán al principio. La medida de lo prometedor que resulta un nodo para su expansión se puede estimar de varias formas. Por ejemplo, se puede tratar de medir la dificultad de resolver el subproblema que plantea el nodo, o bien estimar la calidad de las soluciones que se pueden obtener a partir de ese nodo. En cualquier caso, al definir la función f para un nodo n hay que tener en cuenta la descripción del propio estado n, toda la información acumulada durante la búsqueda, y lo que suele ser más importante, cualquier tipo de información, pista o conocimiento que se tenga sobre el dominio del problema. Por esa razón, f se suele denominar *función heurística de evaluación.*

Algoritmo A*

Sobre el algoritmo

El algoritmo A* es a su vez una especialización del algoritmo BF en el que la función de evaluación f se define de una forma particular. Antes de ver esta definición de f se introducen algunos conceptos y notación. Sea n un nodo cualquiera del espacio de búsqueda

Definición 9.1. *$g^*(n)$ es el coste del camino más corto, o de menor coste, desde el inicial a n.*

Definición 9.2. *$h^*(n)$ es el coste del camino más corto desde n al objetivo más cercano a n.*

Definición 9.3. *$f^*(n) = g^*(n) + h^*(n)$. Es decir, $f^*(n)$ es el coste del camino más corto desde el inicial a los objetivos condicionado a pasar por el nodo n.*

Definición 9.4. *$C^* = f^*(inicial) = h^*(inicial)$. Es decir, C^* es el coste de la solución óptima.*

Es evidente que si se pudiese determinar, con un método eficiente, los valores de las funciones g^* y h^* para todos los nodos, se dispondría de un algoritmo de búsqueda muy eficiente que iría de forma directa a la solución óptima. La idea sería simplemente expandir siempre el nodo de menor f^*, y en el caso de empate elegir un sucesor del último nodo expandido. De esta forma solamente se expandirían nodos de un camino óptimo. Para todos estos nodos se cumple que $f^*(n) = C^*$, mientras que para aquellos nodos que no están en un camino óptimo ocurre que $f^*(n) > C^*$. Luego f^* es un discriminante perfecto. Obviamente, para problemas complejos, los valores de estas funciones no se pueden calcular en un tiempo razonable, con lo cual el método de búsqueda anterior no es factible. Lo que sí se puede hacer es trabajar con aproximaciones de las funciones g^* y h^*. Esto es precisamente lo que se hace en el algoritmo A*, en el que se utilizan las siguientes aproximaciones.

Definición 9.5. *$g(n)$ es el coste del mejor camino desde el inicial a n obtenido hasta el momento durante la búsqueda.*

Definición 9.6. *$h(n)$ es una estimación positiva del valor de $h^*(n)$, tal que $h(n) = 0$, si n es un objetivo.*

De estas dos definiciones resulta obvio que $g(n) \geq g^*(n)$ y $h(n) \geq 0$. Por último se define la función f del algoritmo BF, para obtener el algoritmo A*, como

Definición 9.7. *$f(n) = g(n) + h(n)$.*

De este modo f es una estimación de f^* y se utiliza para ordenar *ABIERTA* de modo que siempre estará al principio de esta lista el nodo con menor valor de f.

La función h se suele denominar *función heurística* o simplemente *heurístico*. La definición de esta función es un problema no trivial, y para ello habrá que tener en cuenta todo tipo de información obtenida por métodos más o menos formales, como cálculos estadísticos, o cualquier tipo de intuiciones o conocimientos que suelen tener los expertos en la resolución del problema. Como se verá más adelante, las propiedades de esta función condicionan totalmente el comportamiento del algoritmo A*.

Descripción del algoritmo

El algoritmo A* es el mismo que el AGBG descrito en el Algoritmo 8.6 del capítulo 8 con algunos cambios debidos a la forma de definir la función de evaluación f. En la versión que se muestra en el Algoritmo 9.1, se hace una ligera modificación del contenido de la $TABLA\_A$ para registrar, para cada nodo n encontrado, el valor de $g(n)$ que es lo mismo que $Coste(inicial, n)$ del algoritmo general, y el valor de $h(n)$. En esta descripción se supone que el valor de $h(n)$, una vez encontrado el nodo n, no cambia a lo largo del tiempo. Como la evaluación de h tiene en general un coste no despreciable, el valor de $h(n)$ se calcula una sola vez para cada nodo n y se almacena este valor en la $TABLA\_A$ para su uso posterior.

Algoritmo 9.1 Algoritmo A*.

```
ABIERTA = (inicial);
mientras NoVacia(ABIERTA) hacer
  n = ExtraePrimero(ABIERTA);
  si EsObjetivo(n) entonces
    devolver Camino(inicial, n) y para;
  fin si
  S = Sucesores(n);
  Añade S a la entrada de n en la TABLA_A;
  para cada q de S hacer
    si (q ∈ TABLA_A) entonces
      Rectificar(q, n, Coste(n, q));
      Ordenar(ABIERTA); {si es preciso}
    si no
      pone q en la TABLA_A con
          Anterior(q) = n,
          g(q) = g(n) + Coste(n, q),
          h(q) = Heuristico(q);
      ABIERTA = Mezclar(q, ABIERTA);
    fin si
  fin para
fin mientras
devolver "no solución";
```

Algoritmo 9.2 Funciones de rectificación de nodos ya expandidos.

```
Función Rectificar (n, p, costepn);
si (g(p) + costepn < g(n)) entonces
  Modifica la entrada del nodo n en la TABLA_A con g(n) = g(p) + costepn; Anterior(n) = p;
  RectificarLista(n);
fin si

Función RectificarLista (n);
LISTA = Sucesores(n); /* Registrados en la TABLA_A */
para cada q de LISTA hacer
  Rectificar(q, n, Coste(n, q));
fin para
```

Expansión de nodo y generación se hijos en el algoritmo A*

Ejercicio 1. (Valoración: 2.5 puntos)
Explique detalladamente cada uno de los tres casos que pueden tener lugar en el algoritmo A* cuando se expande un nodo y se generan sus hijos. Haga énfasis en lo que ocurre en ABIERTA y en TABLA_A en los tres casos mencionados.

SOLUCIÓN por Severino Fernández Galán y José Luis Aznarte Mellado:

Sólo pueden tener lugar tres casos al **expandir** el primer nodo *n* de ABIERTA en el algoritmo A* y **generar** cada uno de sus hijos/sucesores *q*. (Obsérvese que *n* es sacado de ABIERTA para su expansión.)

Caso 1)
Si *q* no estaba en TABLA_A, quiere decir que hemos generado un nodo nuevo en la búsqueda. Por tanto, hay que meter dicho nodo nuevo *q* en TABLA_A, marcar *n* como su mejor padre y calcular $g(q)$ sumando $g(n)$ y el coste del arco que une *n* con *q*. Además, a *q* se le asigna $h(q)$ y es introducido en ABIERTA según su valor de $f(q) = g(q) + h(q)$. Éste es el caso más sencillo de los tres.

Caso 2)
Si *q* estaba en TABLA_A y su lista de hijos/sucesores en dicha tabla es vacía, quiere decir que *q* había sido previamente generado pero no expandido. En este caso hay que estudiar si hay que rectificar $g(q)$, ya que se ha encontrado un nuevo camino desde *q* al nodo inicial, que podría mejorar el mejor camino actual entre *q* y el nodo inicial. En caso de que la rectificación de $g(q)$ se produzca finalmente, quiere decir que el mejor padre de *q* ha cambiado en TABLA_A o, dicho de otro modo de forma más gráfica, el enlace que une a *q* con su mejor padre actual ha sido reorientado hacia *n*.

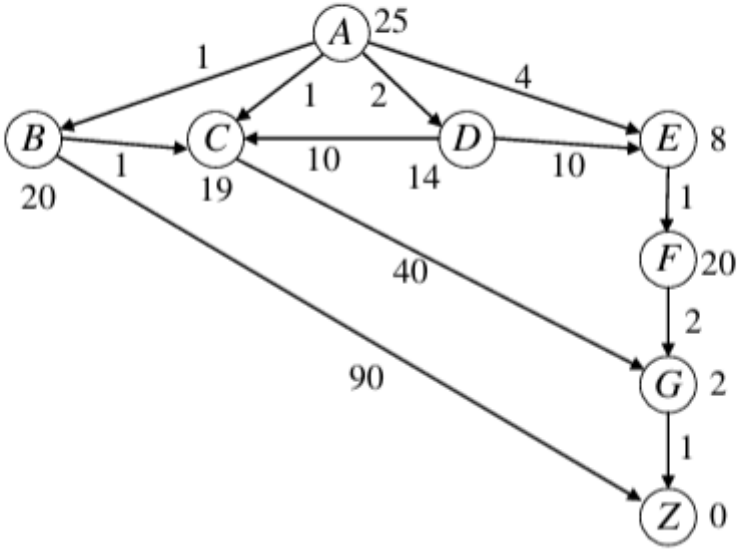
Caso 3)
Si *q* estaba en TABLA_A y su lista de hijos/sucesores en dicha tabla es no vacía, quiere decir que *q* ya había sido expandido anteriormente. En este caso, al igual que en el caso 2), hay que estudiar si hay que rectificar $g(q)$; en caso afirmativo, habría que repetir los pasos descritos en el caso 2) y, además, pasar a estudiar si hay que rectificar el valor de *g* de los hijos de *q* (y así recursivamente hasta que no queden más descendientes de *q* en TABLA_A que haya que estudiar para una posible rectificación de su valor de *g*). Éste es el caso más complejo de los tres.

Nótese que en cualquiera de los tres casos anteriores siempre hay que añadir *q* como hijo de *n* en TABLA_A.

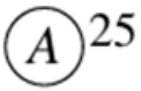
Por último, obsérvese que, mientras que los valores de *g* pueden cambiar a lo largo del proceso de búsqueda, los valores de *h* los da el experto y no cambian. A* utiliza una función heurística *f* para ordenar ABIERTA que tiene la forma: $f = g + h$.

Aplicación del algoritmo A* a un grafo (ejemplo 1)

Ejercicio 1. (Valoración: 3 puntos)
Considere el grafo de la figura, donde *A* es el nodo inicial y *Z* el único nodo meta. Cada arco lleva asociado su coste y en cada nodo aparece la estimación de la menor distancia desde ese nodo a la meta. Aplicar paso a paso el algoritmo A* al grafo dado.

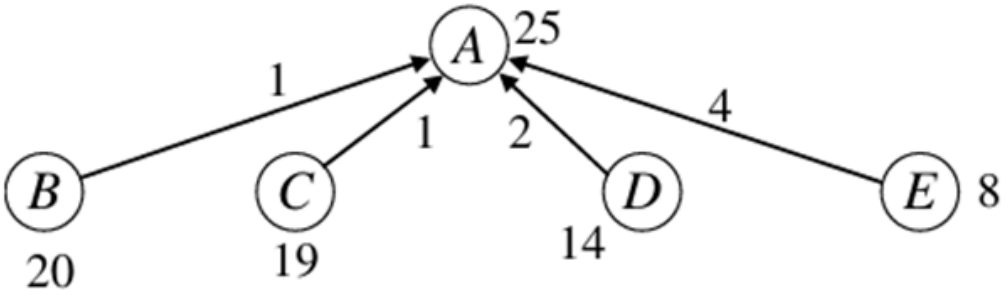


- **PASO 0.** El nodo inicial *A* es introducido en ABIERTA y en TABLA_A, con lo que tenemos la siguiente situación:



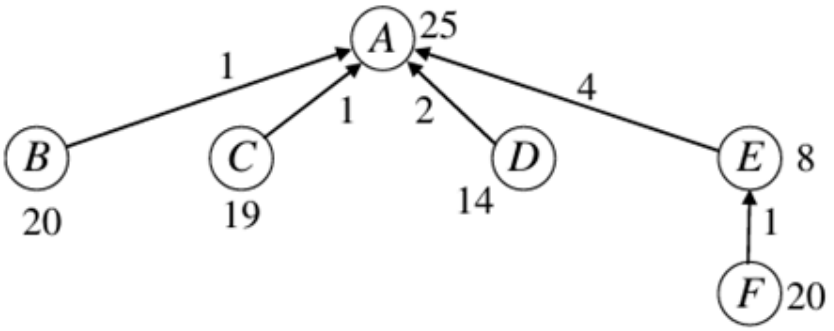
ABIERTA = {*A*(0+25)}

- **PASO 1.** Expandimos el nodo *A* de ABIERTA. Tras la expansión, la situación es la siguiente:



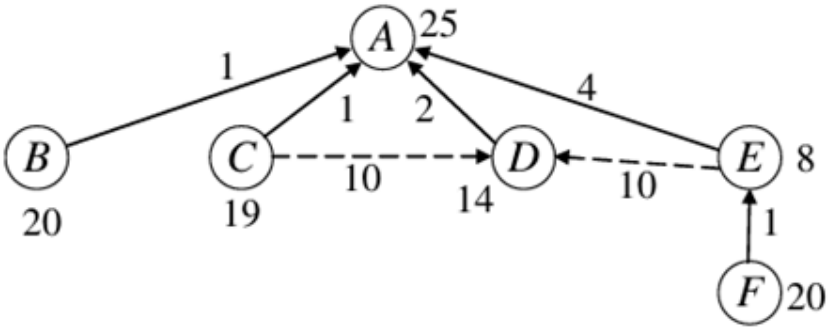
ABIERTA = {*E*(4+8), *D*(2+14), *C*(1+19) , *B*(1+20)}

- PASO 2. Expandimos *E* por ser el nodo de ABIERTA con menor valor de la función de evaluación heurística, *f*=*g*+*h* (al ser *g*=4 y *h*=8).



ABIERTA = {*D*(2+14), *C*(1+19), *B*(1+20), *F*(5+20)}

- PASO 3. Expandimos *D*.

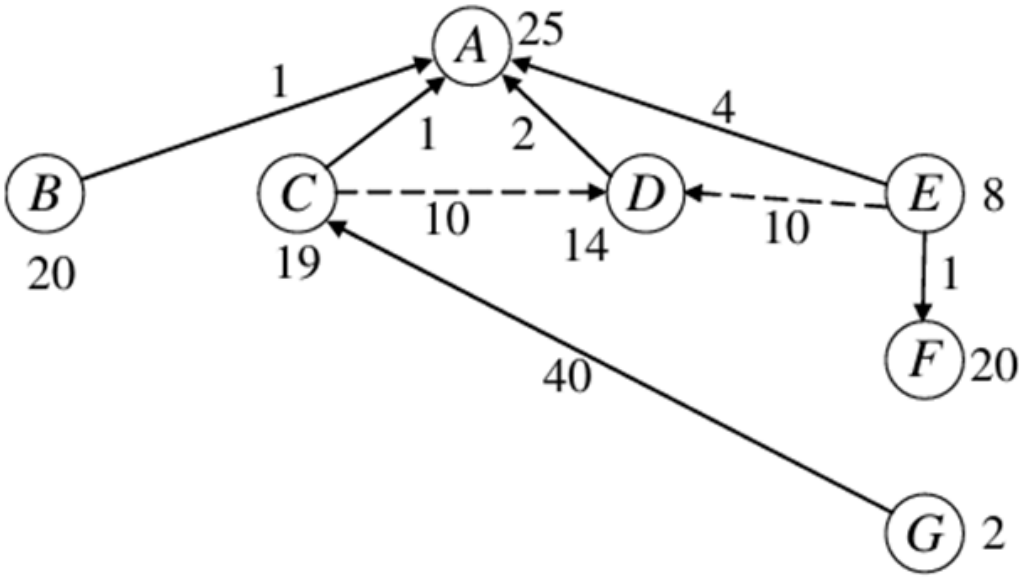


ABIERTA = {*C*(1+19), *B*(1+20) , *F*(5+20)}

Observe que el mejor camino desde *C* al nodo inicial lo marca su padre *A* (coste 1) y no su padre *D* (coste 10+2=12). Por ello, el arco ascendente de *C* a *A* se marca con trazo continuo y el arco ascendente de *C* a *D* se marca con trazo discontinuo. Un razonamiento similar se puede aplicar al nodo *E*, cuyo mejor camino al nodo inicial lo marca su padre *A* (coste 4) y no su padre *D* (coste 10+2=12).

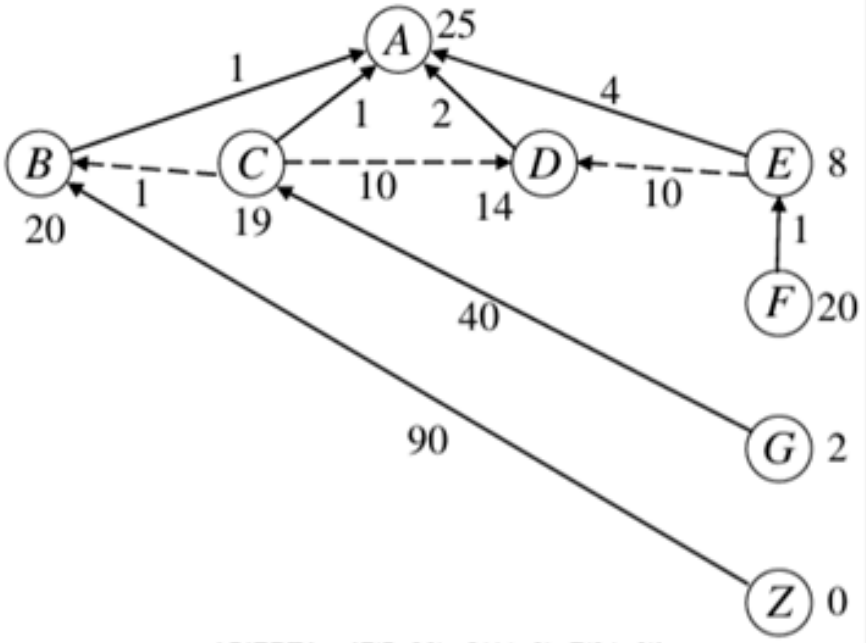
Es importante darse cuenta que el conjunto de arcos con trazo continuo formará siempre un árbol en el grafo parcial de búsqueda desarrollado hasta el momento.

- PASO 4. Expandimos *C*.



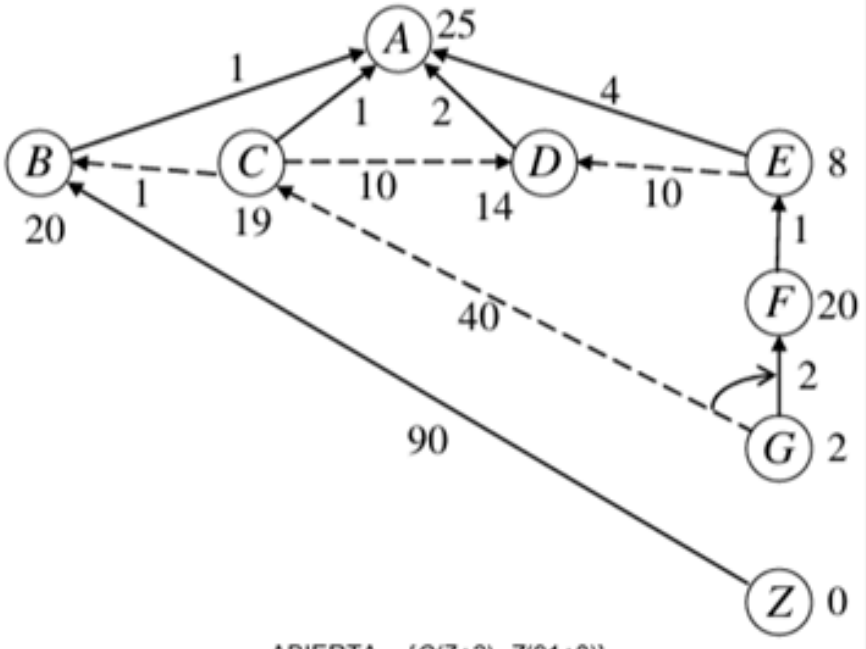
ABIERTA = {*B*(1+20), *F*(5+20) , *G*(41+2)}

- PASO 5. Expandimos *B*.



ABIERTA = {*F*(5+20), *G*(41+2), *Z*(91+0)}

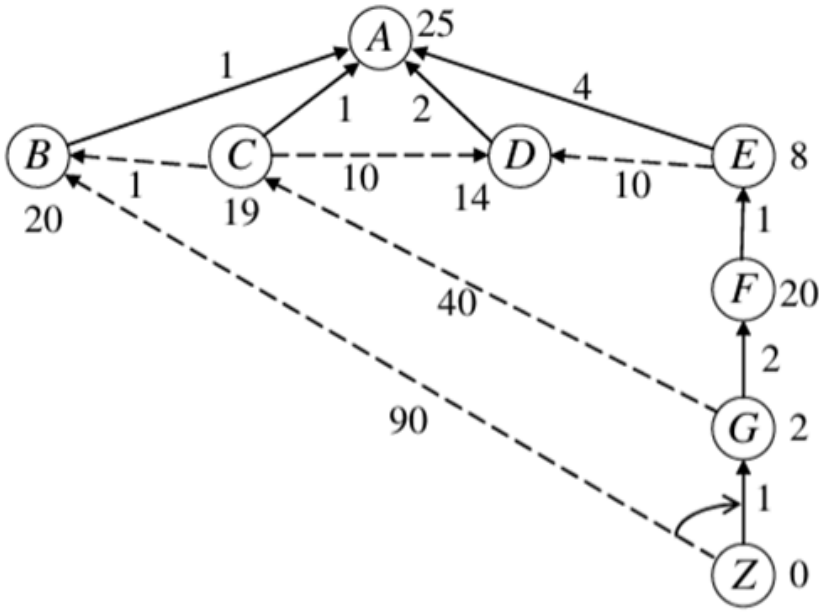
- PASO 6. Expandimos *F*.



ABIERTA = {*G*(7+2), *Z*(91+0)}

Observe que ha habido una reorientación del mejor padre de *G*, que antes era *C* y ahora pasa a ser *F*. El nuevo coste de *G* al nodo inicial es 2+1+4=7, que se puede calcular a partir de los costes de los mejores arcos ascendentes hallados hasta el momento: $G \rightarrow F$, $F \rightarrow E$ y $E \rightarrow A$.

- PASO 7. Expandimos *G*



ABIERTA = {Z(8+0)}

- PASO 8. Expandimos *Z* y alcanzamos una meta, con lo que el algoritmo termina. El camino solución es: $Z \rightarrow G \rightarrow F \rightarrow E \rightarrow A$, cuyo coste es 8.

Aplicación del algoritmo A* a un grafo (ejemplo 2)

Ejercicio 1, Semana 2. (Valoración: 2.75 puntos)
Considere el espacio de búsqueda de la figura, donde aparece el coste asociado a cada operador y el valor para cada nodo de la función heurística *h* de estimación de la menor distancia a meta. El nodo *A* es el nodo inicial, mientras que los nodos *Y* y *Z* son los nodos meta. Se pide desarrollar razonadamente los siguientes apartados:

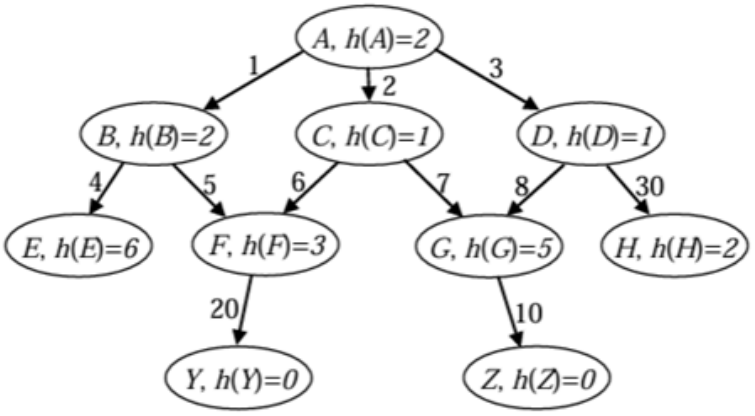
(1) Calcule g^* para cada nodo.

(2) Calcule h^* para cada nodo.

(3) ¿Es *h* admisible?

(4) ¿Es *h* monótona?

(5) Teniendo en cuenta los resultados obtenidos en los apartados anteriores, si se aplicara el algoritmo A*: ¿qué nodos se expandirían con seguridad?, ¿qué nodos no se expandirían con seguridad? y, finalmente, ¿qué nodos se podrían expandir o no dependiendo de la forma en que se resolvieran los empates? (En este último apartado no se permite aplicar ninguna iteración del algoritmo A* para justificar la respuesta.)



(1) La función $g^*(n)$ representa el menor coste desde el nodo inicial al nodo *n*. En la siguiente tabla aparece el cálculo de g^* para cada nodo.

Nodo	g^*	Camino
A	0	*A*
B	1	*A–B*
C	2	*A–C*
D	3	*A–D*
E	5	*A–B–E*
F	6	*A–B–F*
G	9	*A–C–G*
H	33	*A–D–H*
Y	26	*A–B–F–Y*
Z	19	*A–C–G–Z*

(2) La función $h^*(n)$ representa el menor coste desde el nodo n a una meta. En la siguiente tabla aparece el cálculo de h^* para cada nodo.

Nodo	h^*	Camino
A	19	*A–C–G–Z*
B	25	*B–F–Y*
C	17	*C–G–Z*
D	18	*D–G–Z*
E	∞	*No hay*
F	20	*F–Y*
G	10	*G–Z*
H	∞	*No hay*
Y	0	*Y*
Z	0	*Z*

(3) Efectivamente, h es admisible, ya que $h(n) \leq h^*(n)$ para todo nodo n. Esto se puede comprobar de forma inmediata a partir de la siguiente tabla.

Nodo	h	h^*	¿ $h \leq h^*$?
A	2	19	*sí*
B	2	25	*sí*
C	1	17	*sí*
D	1	18	*sí*
E	6	∞	*sí*
F	3	20	*sí*
G	5	10	*sí*
H	2	∞	*sí*
Y	0	0	*sí*
Z	0	0	*sí*

(4) Efectivamente, h es monótona, ya que $h(n) \leq k(n, n') + h(n')$ para todo par de nodos (n, n'), donde $k(n, n')$ es el menor coste para ir de n a n'. Esto se puede comprobar de forma inmediata a partir de la siguiente tabla.

$n,h(n)$ \ $n',h(n')$	*A,2*	*B,2*	*C,1*	*D,1*	*E,6*	*F,3*	*G,5*	*H,2*	*Y,0*	*Z,0*
A,2	0	1	2	3	5	6	9	33	26	19
B,2	∞	0	∞	∞	4	5	∞	∞	25	∞
C,1	∞	∞	0	∞	∞	6	7	∞	26	17
D,1	∞	∞	∞	0	∞	∞	8	30	∞	18
E,6	∞	∞	∞	∞	0	∞	∞	∞	∞	∞
F,3	∞	∞	∞	∞	∞	0	∞	∞	20	∞
G,5	∞	∞	∞	∞	∞	∞	0	∞	∞	10
H,2	∞	∞	∞	∞	∞	∞	∞	0	∞	∞
Y,0	∞	∞	∞	∞	∞	∞	∞	∞	0	∞
Z,0	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

Valores de $k(n, n')$

Por ejemplo, para $n = C$ y $n' = G$, $h(C) \leq k(C, G) + h(G)$, ya que $h(C) = 1$ (véase columna más a la izquierda de la tabla), $k(C, G) = 7$ (véase la celda marcada en la tabla) y $h(G) = 5$ (véase fila superior de la tabla).

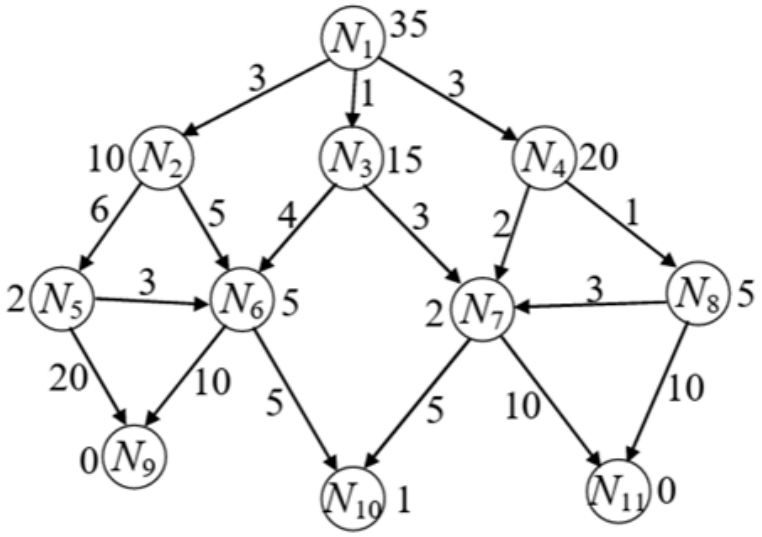
(5) Al ser h monótona, según el teorema 9.11 (página 352) del texto base, la condición necesaria de expansión de un nodo es que $g^*(n) + h(n) \leq C^*$ y la condición suficiente es que $g^*(n) + h(n) < C^*$, donde $C^* = h^*(A)$ (coste óptimo desde el estado inicial a una meta). En la siguiente tabla aparecen los nodos ordenados según su valor de $g^* + h$.

Nodo	$g^* + h$
A	2
B	3
C	3
D	4
F	9
E	11
G	14
Z	19
Y	26
H	35

Como $C^* = 19$, los nodos {*A*, *B*, *C*, *D*, *F*, *E*, *G*} se expandirían con seguridad, los nodos {*Y*, *H*} no se expandirían con seguridad y, finalmente, el nodo *Z* se expandiría en función de la forma de deshacer los empates. (De hecho, por el teorema 9.10 del texto base, como h es monótona, al elegir *Z* para su expansión entontes $g(Z) = g^*(Z)$. Esto quiere decir que, como *Z* es un nodo meta cuyo valor de $g^* + h = g + h$ es estrictamente menor que los valores de $g^* + h = g + h$ del resto de metas, también sería expandido finalmente.)

Aplicación del algoritmo A* a un grafo (ejemplo 3)

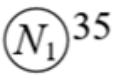
Ejercicio 1. (Valoración: 2.5 puntos)
Considere el grafo de la figura, donde el nodo inicial es N_1 y los nodos meta son N_9 y N_{11}. Cada arco lleva asociado su coste y en cada nodo aparece la estimación de la menor distancia desde ese nodo a una meta. Aplicar paso a paso el algoritmo A* al grafo dado.



SOLUCIÓN por Severino Fernández Galán y José Luis Aznarte Mellado:

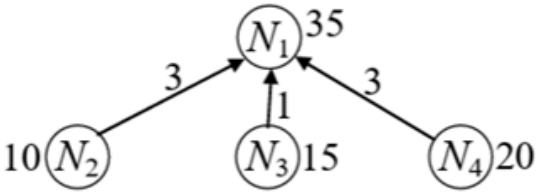
De cara a ilustrar la respuesta convenientemente, incluimos la información sobre TABLA_A gráficamente. Para ello es necesario trazar, para cada nodo generado en una expansión, un arco ascendente a su padre expandido; además, hay que anotar para cada nodo su mejor padre encontrado hasta el momento. De esta manera, siguiendo cada arco al mejor padre, se puede saber cuál es el mejor camino encontrado hasta el momento desde cada nodo al nodo inicial. Además, los arcos ascendentes que llegan a un nodo ya expandido lo enlazan a sus nodos hijos.

- PASO 0. El nodo inicial N_1 es introducido en ABIERTA y en TABLA_A, con lo que tenemos la siguiente situación:



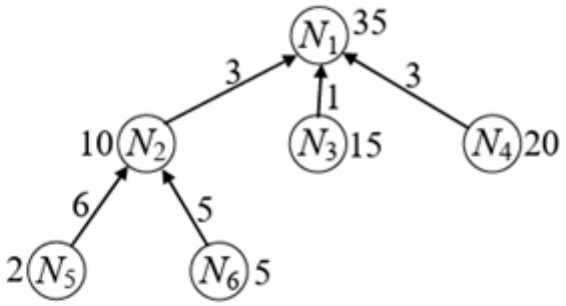
ABIERTA = {$\underline{N_1(0+35)}$}

- PASO 1. Expandimos el nodo N_1 de ABIERTA. Tras la expansión, la situación es la siguiente:



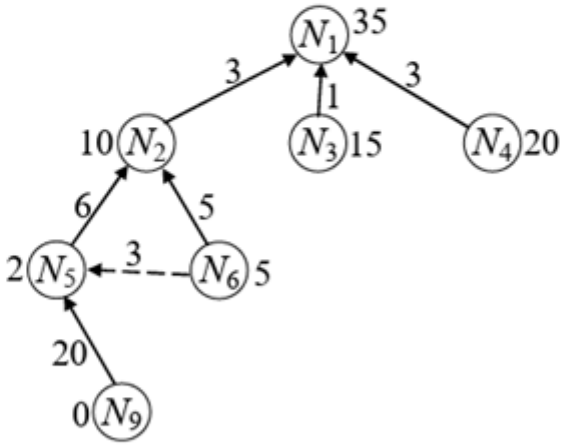
ABIERTA = {$\underline{N_2(3+10)}$, $N_3(1+15)$, $N_4(3+20)$}

- PASO 2. Expandimos N_2 por ser el nodo de ABIERTA con menor valor de la función de evaluación heurística, $f=g+h$ (al ser $g=3$ y $h=10$).



ABIERTA = {N_5(9+2), N_6(8+5), N_3(1+15), N_4(3+20)}

- PASO 3. Expandimos N_5.

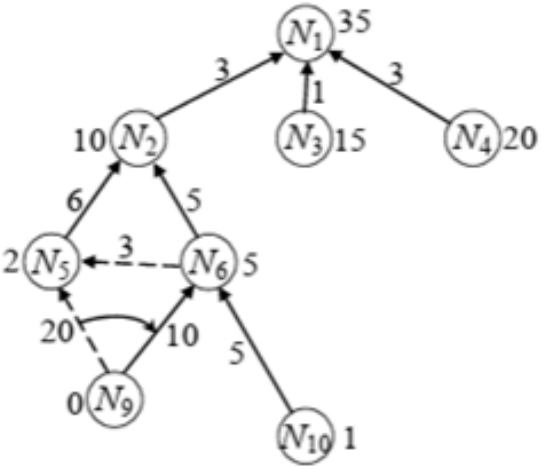


ABIERTA = {N_6(8+5), N_3(1+15), N_4(3+20), N_9(29+0)}

Observe que el mejor camino desde N_6 al nodo inicial lo marca su padre N_2 (coste 5+3=8) y no su padre N_5 (coste 3+6+3=15). Por ello, el arco ascendente de N_6 a N_2 se marca con trazo continuo y el arco ascendente de N_6 a N_5 se marca con trazo discontinuo.

Es importante darse cuenta que el conjunto de arcos con trazo continuo formará siempre un árbol en el grafo parcial de búsqueda desarrollado hasta el momento.

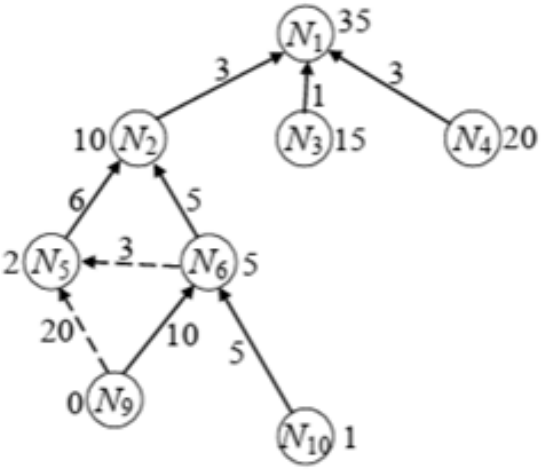
- **PASO 4**. Expandimos N_6.



ABIERTA = {$\underline{N_{10}(13+1)}$, $N_3(1+15)$, $N_9(18+0)$, $N_4(3+20)$}

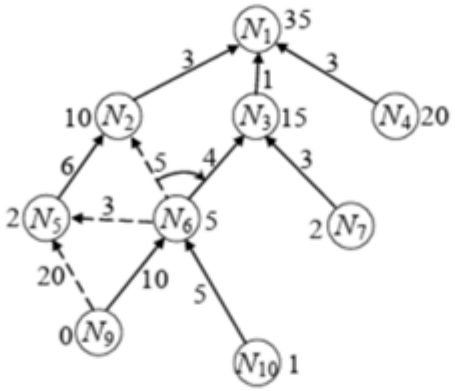
Observe que ha habido una reorientación del mejor padre de N_9, que antes era N_5 y ahora pasa a ser N_6. El nuevo coste de N_9 al nodo inicial es 10+5+3=18, que se puede calcular a partir de los costes de los mejores arcos ascendentes hallados hasta el momento: $N_9 \rightarrow N_6$, $N_6 \rightarrow N_2$ y $N_2 \rightarrow N_1$.

- **PASO 5**. Expandimos N_{10}.



ABIERTA = {$N_5(1+15)$, $N_9(18+0)$, $N_4(3+20)$}

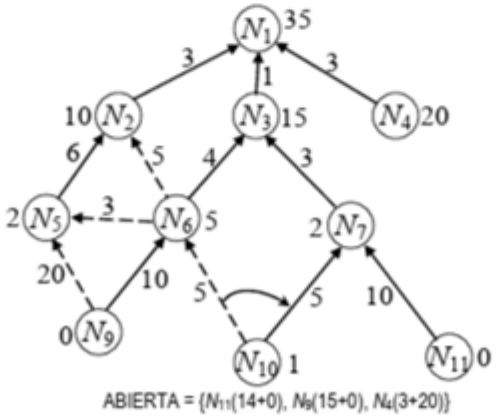
- **PASO 6**. Expandimos N_5.



ABIERTA = {N_7(4+2), N_9(15+0), N_4(3+20)}

Observe que ha habido una reorientación del mejor padre de N_6, que antes era N_2 y ahora pasa a ser N_3. El nuevo coste de N_6 al nodo inicial es 4+1=5, que se puede calcular a partir de los costes de los mejores arcos ascendentes hallados hasta el momento: $N_6 \rightarrow N_3$ y $N_3 \rightarrow N_1$.

- **PASO 7**. Expandimos N_7.



ABIERTA = {N_{11}(14+0), N_9(15+0), N_4(3+20)}

Observe que ha habido una reorientación del mejor padre de N_{10}, que antes era N_6 y ahora pasa a ser N_7. El nuevo coste de N_{10} al nodo inicial es 5+3+1=9, que se puede calcular a partir de los costes de los mejores arcos ascendentes hallados hasta el momento: $N_{10} \rightarrow N_7$, $N_7 \rightarrow N_3$ y $N_3 \rightarrow N_1$.

- **PASO 8**. Expandimos N_{11} y alcanzamos una meta, con lo que el algoritmo termina. El camino solución es: $N_{11} \rightarrow N_7 \rightarrow N_3 \rightarrow N_1$, cuyo coste es 14.

Búsqueda con memoria limitada

El principal problema que tiene el algoritmo A* es el requerimiento de memoria que crece de forma exponencial con la profundidad, aunque dispongamos de buenos heurísticos. Para tratar de salvar este problema, se han propuesto variantes que incorporan mecanismos para limitar la cantidad de memoria. El primero de estos métodos se denomina IDA* (Iterative Deepening A*) y es una extensión del método de búsqueda iterativa en profundidad. El segundo se denomina SMA* (Simplified Memory-Bounded A*) y es similar a A*, pero restringe el tamaño de *ABIERTA* a un valor máximo prefijado. Para simplificar la exposición, se consideran versiones de estos algoritmos para el recorrido de árboles.

Algoritmo IDA*

Este algoritmo fue propuesto por R. Korf [Korf, 1985]. En principio, se establece una longitud[2] límite para la búsqueda igual al valor de $f(inicial)$ que es, obviamente, una cota inferior de la solución óptima, supuesto que el heurístico es admisible. En cada iteración, IDA* busca con una estrategia en profundidad descartando los nodos n cuya estimación $f(n)$ supere la longitud límite. Si en una iteración no se encuentra

[2]En este caso la longitud de un camino se refiere a la suma de los costes de sus arcos y no al número de éstos.

solución, se realiza una nueva iteración en la que la búsqueda comienza otra vez desde el principio, pero esta vez con una nueva longitud límite, el menor valor de f de los nodos descartados en la iteración anterior. Puesto que el algoritmo dispone de una función heurística h, se puede utilizar esta información para ordenar los sucesores del nodo expandido antes de insertarlos en *ABIERTA*, de modo que se consideren antes los sucesores más prometedores de cada nodo expandido. De este modo, si en una iteración se encuentra la solución, el número de nodos expandidos en esta iteración puede ser menor. Si por el contrario en una iteración no se encuentra solución, el número de nodos expandidos es el mismo que si los sucesores se ordenasen de otro modo. A continuación se muestran el algoritmo IDA* (9.3) y el algoritmo de búsqueda heurística con longitud limitada que es utilizado por el algoritmo IDA* en cada iteración (9.4).

Algoritmo 9.3 Algoritmo IDA* (Iterative Deepening A*).

```
LongitudLímite = f(inicial);
Solución = Primero en Profundidad con Longitud Limitada;
mientras (Solución = "no encontrado") hacer
    ProfundidadLímite = NuevaLongitudLímite;
    Solución = Primero en Profundidad con Longitud Limitada;
fin mientras
devolver Solución;
```

Algoritmo 9.4 Algoritmo de búsqueda heurística en profundidad en árboles con longitud limitada.

```
ABIERTA = (inicial);
mientras (NoVacía(ABIERTA)) hacer
    n = ExtraePrimero(ABIERTA);
    si (EsObjetivo(n)) entonces
        devolver Camino(inicial, n)
    fin si
    S = ∅ ;
    si (f(n) < LongitudLímite entonces
        S = Sucesores(n);
    si no
        Actualiza NuevaLongitudLímite con f(n);
    fin si
    si (Vacía(S)) entonces
        LimpiarTABLA_A(n)
    fin si
    para cada q de S hacer
        pone q en la TABLA_A con
            Anterior(q) = n,
            g(q) = Coste(inicial, n) + Coste(n, q),
            h(q) = Heurístico(q);
    fin para
    Ordena S según los valores de f de menor a mayor;
    ABIERTA = Concatenar(S, ABIERTA);
fin mientras
devolver "no encontrado" y NuevaLongitudLímite;
```

Pregunta de examen sobre el algoritmo IDA*

Ejercicio 2. (Valoración: 2.5 puntos)
Describa detalladamente las características del algoritmo IDA*.

SOLUCIÓN por Severino Fernández Galán

El algoritmo A* requiere memoria que crece de forma exponencial con el tamaño del problema, incluso disponiendo de buenos heurísticos. Una de las variantes del algoritmo A* que aborda este problema es el algoritmo IDA*, propuesto a mediados de los 80 y cuyas siglas en inglés significan *Iterative Deepening A**. En líneas generales, IDA* amplía la búsqueda iterativa en profundidad asociando a cada nodo *n* su valor de la función heurística propia del algoritmo A*, $f(n) = g(n) + h(n)$.

IDA* realiza búsquedas primero en profundidad iterativamente desde el nodo raíz, aumentando en cada iteración la profundidad límite de dichas búsquedas. La primera búsqueda en profundidad se realiza a través de todos aquellos nodos *n* tal que $f(n) < f(\textit{nodo raíz})$ (Si suponemos que *f* es admisible entonces *f*(nodo raíz) es menor que el coste de la solución óptima.) Cualquier nodo que no cumpla dicha condición es tratado como un callejón sin salida al intentar ser expandido, es decir, es descartado en el proceso de búsqueda en profundidad. El menor valor de *f* de los nodos descartados en la búsqueda en profundidad de la iteración actual será el límite de la búsqueda en profundidad de la siguiente iteración.

Por lo general, los hijos de cada nodo expandido se introducen ordenados según su valor de *f* en ABIERTA, que actúa como una **pila**. De este modo, se consideran antes los hijos más prometedores de cada nodo expandido, lo cual generalmente reduce el número de nodos expandidos en una iteración de búsqueda en profundidad en caso de que en la misma se llegue a la meta.

Si *h* es admisible, IDA* también lo será. Por otra parte, al realizar IDA* una iteración de búsquedas en profundidad, su consumo de memoria es proporcional al producto de la profundidad de la solución y del factor de ramificación, lo cual supone un importante ahorro de memoria. Sin embargo, el tiempo de búsqueda en cada iteración de búsqueda en profundidad es exponencial con la profundidad límite.

Algoritmo SMA*

El principal problema del algoritmo IDA* es la cantidad de nodos que tiene que volver a expandir, debido a que lo único que recuerda de una iteración a la siguiente es el valor del menor valor de f de los nodos descartados. El algoritmo SMA* propuesto por S. Russell [Russell, 1992] resuelve este problema a la vez es capaz de operar con una cantidad limitada de memoria. Para mantener una cierta uniformidad con las descripciones de los algoritmos anteriores, la descripción que se sigue aquí, y que se muestra en el Algoritmo 9.5, difiere en algunos aspectos de la original que puede verse también en el texto de Russell y Norvig [Russell y Norvig, 2003].

En el Algoritmo 9.5, el valor de la variable *Limite* se refiere al número máximo de nodos que se pueden almacenar en la $TABLA\_A$, y por lo tanto a la profundidad máxima de un camino registrado en esta tabla. La llamada a la función $SiguienteSucesor(n)$ proporciona en cada llamada un sucesor del nodo, en principio en un orden no determinado, por lo que es preciso algún mecanismo de control de los sucesores generados hasta el momento. Este mecanismo se puede apoyar en la $TABLA\_A$. En líneas generales, el funcionamiento del algoritmo SMA* es el siguiente. Cuando necesita expandir un nodo y no tiene espacio en la $TABLA\_A$, elimina un nodo de esta tabla y de *ABIERTA*. Estos son los llamados nodos olvidados y son aquellos menos prometedores, es decir los que tienen un mayor valor de f en *ABIERTA*. El algoritmo recuerda en cada nodo el mejor f de los hijos de ese nodo que han sido olvidados. Para no volver a explorar subárboles que han sido eliminados de memoria, SMA* mantiene en los nodos ancestros información acerca de la calidad del mejor camino en cada subárbol descartado. De este modo, solamente reexplora un subárbol descartado cuando el resto de posibilidades es peor de acuerdo con las estimaciones. Para proceder de este modo, cuando se descartan todos los sucesores de un nodo n, el algoritmo recuerda la estimación del mejor camino a través de n.

El algoritmo SMA* es bastante sofisticado, por lo que una forma de entenderlo es a través de un buen ejemplo como el que se puede encontrar en el texto de Russell y Norvig [Russell y Norvig, 2003, pp. 108-109].

Algoritmo 9.5 Algoritmo SMA* (Simplified Memory-Bounded A*).

Inserta el *inicial* en $ABIERTA$ y en la $TABLA\_A$;
mientras ($NoVacía(ABIERTA)$) **hacer**
  $n = Primero(ABIERTA)$;
  si ($EsObjetivo(n)$) **entonces**
    devolver $Camino(inicial, n)$
  fin si
  $s = SiguienteSucesor(n)$;
  si ($NoEsObjetivo(s)$ y $Profundidad(s) = Límite$) **entonces**
    $f(s) = \infty$;
  si no
    $f(s) = \text{máx}\,(f(n), g(s) + h(s))$;
  fin si
  si Todos los sucesores de n han sido generados **entonces**
    Actualiza $f(n)$ y el f de los ancestros de n
  fin si
  si (Todos los sucesores de n están en la $TABLA\_A$) **entonces**
    Elimina n de $ABIERTA$;
  fin si
  si (la $TABLA\_A$ está llena) **entonces**
    Elimina de $ABIERTA$ y de la $TABLA\_A$ el nodo con mayor f en $ABIERTA$ y menor profundidad, actualizando la estimación del mejor nodo hijo olvidado en el padre del nodo eliminado;
    Inserta el padre del nodo eliminado en $ABIERTA$ si no está;
  fin si
  Inserta s en $ABIERTA$ y en la $TABLA\_A$;
fin mientras
devolver "no solución";

Por último, tal y como se indica en el texto de Russell y Norvig, las propiedades principales del algoritmo SMA* son las siguientes:

- Es capaz de evolucionar utilizando la memoria que tenga disponible.
- Evita estados repetidos en la medida en que la cantidad de memoria disponible se lo permita.
- Es completo si la memoria disponible es suficiente para almacenar el camino a la solución menos profunda.
- Es admisible si tiene suficiente memoria para almacenar el camino hasta la solución óptima menos profunda. En otro caso, devuelve la mejor solución que se puede alcanzar con la memoria disponible.
- Si la memoria es suficiente para el árbol de búsqueda completo, la eficiencia de la búsqueda es óptima.

Sin embargo, no está resuelta la cuestión de si SMA* tiene una eficiencia mejor o igual que cualquier otro algoritmo, dada la misma información heurística y la misma cantidad de memoria.

Pregunta de examen sobre los algoritmos SMA* e IDA*

Ejercicio 1. (Valoración: 2.5 puntos)
Explique qué similitudes y qué diferencias existen entre los algoritmos SMA* e IDA*.

SOLUCIÓN por Severino Fernández Galán:

Como similitudes se encuentran las siguientes:

- Tanto un algoritmo como el otro son variantes del algoritmo A*.
- Estos dos algoritmos intentan solventar el problema que tiene el algoritmo A* de requerimiento exponencial de memoria con la profundidad.
- En relación al punto anterior, incorporan mecanismos para limitar la cantidad de memoria necesaria durante la ejecución.
- Fueron propuestos con sólo siete años de diferencia, IDA* en 1985 y SMA* en 1992.

Como diferencias se pueden citar las siguientes:

- Mientras que IDA* es una extensión del método de búsqueda iterativa en profundidad, SMA* se basa en limitar el tamaño de TABLA_A a un valor máximo prefijado.
- IDA* utiliza ABIERTA como una pila y no necesita establecer ningún límite a la capacidad de TABLA_A. SMA* trata ABIERTA como una lista ordenada según los valores de *f* y limita la capacidad de TABLA_A a un cierto número de nodos prefijado.
- Como novedad respecto a A*, IDA* establece en cada iteración un coste límite para la búsqueda. En la primera iteración el coste límite es el valor de "*f*" del estado inicial y, en sucesivas iteraciones, el coste límite es el menor valor de *f* de los nodos descartados en la iteración anterior. Por su parte, la novedad que introduce SMA* con respecto a A* consiste en establecer un límite en el número máximo de nodos que se pueden almacenar en TABLA_A. Si se necesita expandir un nodo y no hay espacio en TABLA_A, se elimina de ABIERTA y de TABLA_A el nodo con mayor valor de *f* en ABIERTA. Estos nodos eliminados se denominan "olvidados" y SMA* recuerda en cada nodo el mejor *f* de los hijos de ese nodo que han sido olvidados.
- Mientras que IDA* es completo, SMA* es completo si la memoria disponible es suficiente para almacenar el camino a la solución menos profunda.
- Mientras que IDA* es admisible si *h* es admisible, SMA* es admisible si además tiene suficiente memoria para almacenar el camino hasta la solución óptima menos profunda. En caso contrario, devuelve la mejor solución que se puede alcanzar con la memoria disponible.

Algoritmos voraces

Los algoritmos voraces (greedy) constituyen una alternativa a la búsqueda exhaustiva. La idea básica es que un algoritmo de esta clase toma decisiones de forma irrevocable, de modo que nunca considera otra alternativa. Una consecuencia de este modo de proceder es que los algoritmos voraces no son admisibles, y en general tampoco son completos. Pero por otra parte resultan extremadamente eficientes, por lo que son muy adecuados por ejemplo para los sistemas de tiempo real. Además si la decisión irrevocable está guiada por un buen heurístico, la probabilidad de obtener una buena solución aumenta. Estos algoritmos aparecen con profusión en la literatura, por ejemplo los algoritmos de planificación del procesador, u otro tipo de recursos, de los sistemas operativos, o los algoritmos de enrutamiento de mensajes en redes de comunicaciones, suelen ser algoritmos de este tipo.

El diseño de un algoritmo voraz para la búsqueda en espacios de estados es simple a partir de los algoritmos que se han visto hasta aquí; no obstante dada su simplicidad lo lógico es realizar una implementación más sencilla y eficiente. Por ejemplo, a partir de un algoritmo A*, si en cada expansión se prescinde de todos los sucesores salvo del más prometedor de acuerdo con el heurístico h, se tiene un algoritmo voraz. En este caso se trata de un algoritmo determinista. Si, además, los empates se resuelven de forma aleatoria, se tiene un algoritmo no determinista que, si se ejecuta varias veces, proporciona distintas soluciones con una calidad media que dependerá de la precisión de las estimaciones del heurístico. La variedad de las soluciones se puede aumentar si en lugar de elegir aleatoriamente entre los mejores sucesores con una distribución uniforme, la elección se hace con una distribución proporcional a la bondad estimada de cada sucesor.

Los algoritmos voraces son útiles en la práctica para resolver muchos problemas, y se pueden utilizar además en combinación con otras estrategias de búsqueda, como por ejemplo los esquemas de ramificación y poda que se describen en la sección siguiente, o los algoritmos evolutivos que se verán en el capítulo 11.

Algoritmos de ramificación y poda

- **Ejercicio 1, Semana 2. (Valoración: 2.5 puntos)**

Describa detalladamente las características de los algoritmos de ramificación y poda.

SOLUCIÓN por Severino Fernández Galán

Los *algoritmos de ramificación y poda* interpretan cada estado o nodo como un subconjunto de soluciones de todo el conjunto posible de soluciones del problema original planteado. Este subconjunto de soluciones normalmente no se puede representar por extensión. De esta manera, el nodo raíz representa todas las soluciones posibles al problema original, mientras que un nodo hoja sería aquel que no puede ser expandido por contener una única solución. El proceso de ramificación consiste en descomponer un determinado subconjunto de soluciones en la unión disjunta de varios subconjuntos suyos, con lo que el espacio de búsqueda adquiere forma de árbol. De este modo, la ramificación de un nodo sería equivalente a la expansión de un nodo para generar sus nodos hijos.

Mientras que un nodo hoja o terminal tendrá asociado un coste concreto y conocido, cada nodo no terminal del árbol tiene asociado un valor heurístico que representa una cota inferior del coste de la mejor solución (considerada como la de menor coste) contenida en el nodo. Generalmente, cada vez que la cota inferior de un nodo rebasa el menor de los costes de los nodos hoja encontrados hasta el momento en el resto del árbol, dicho nodo es podado. Este esquema es el más habitual, aunque podría ser ampliado para considerar una cota superior para cada nodo no terminal.

Por motivos de ahorro en memoria, la estrategia de control para la exploración del árbol de búsqueda suele ser en profundidad, aunque otros métodos (heurísticos o no) son posibles. En el caso de la búsqueda en profundidad, ABIERTA actúa como una pila y es habitual introducir en la misma los hijos generados en cada expansión de un nodo ordenados según los valores de sus cotas inferiores. La eficiencia del algoritmo de ramificación y poda dependerá de lo ajustadas que sean las cotas inferiores obtenidas de forma heurística. Por otra parte, los algoritmos de ramificación y poda son admisibles si recorren todo el espacio de búsqueda, excepto las partes podadas, hasta que ABIERTA se agote; en caso contrario, si la condición de terminación es que el algoritmo pare después de expandir cierto número de nodos, se pierde la admisibilidad y únicamente se puede devolver la mejor solución encontrada hasta el momento.

Algoritmos de mejora iterativa o búsqueda local

Recorridos de grafos: Algoritmos de escalada

Algoritmos de escalada o máximo gradiente: pág. 371

EJERCICIO 6:

Considere el grafo de la figura 6.1, donde el nodo inicial es *D* y donde los nodos meta son desconocidos. Cada arco u operador lleva asociado su coste y en cada nodo aparece su valor de la función de evaluación heurística (que hay que minimizar). Aplique paso a paso el algoritmo de escalada o máximo gradiente al grafo dado. Para ello indique de forma razonada qué nodo se expande en cada paso y cuál es el nodo final devuelto por el algoritmo. Utilice como *criterio de selección* el de mejor vecino. Utilice como *criterio de terminación* el que no se hayan producido mejoras durante los tres últimos pasos del algoritmo.

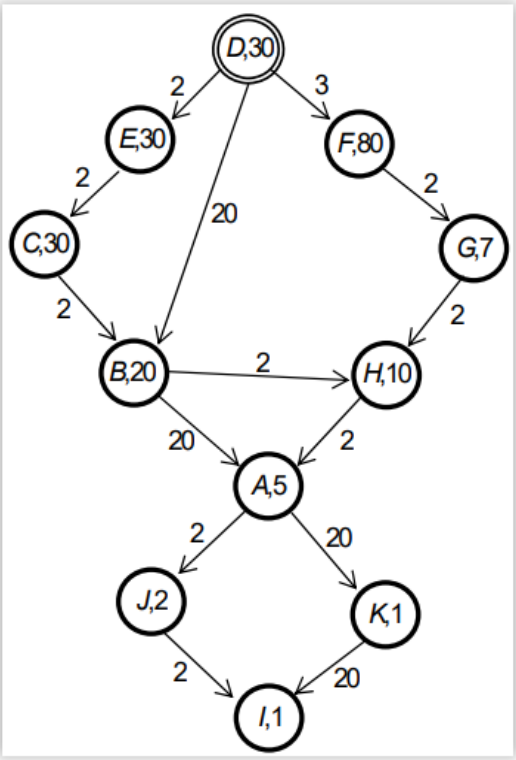


Figura 6.1: Grafo de búsqueda en el que el nodo inicial es *D*, los nodos meta son desconocidos, el coste de cada operador aparece al lado del arco que lo representa y al lado de cada nodo aparece el valor de su función de evaluación heurística (que hay que minimizar).

SOLUCIÓN DEL EJERCICIO 6 (por Severino Fernández Galán):

- PASO 1: Al principio expandimos el nodo inicial *D*, generando sus nodos hijos {*E*(30), *B*(20), *F*(80)}. Seleccionamos el nodo *B* por ser el mejor de los nodos hijos. A continuación aceptamos el nodo *B* como nuevo nodo actual en sustitución de *D*, debido a que el valor de la función de evaluación heurística de *B* es mejor o igual que el de *D* ($20 \leq 30$).

- PASO 2: Expandimos el nodo actual *B* y generamos sus hijos: {*A*(5), *H*(10)}. Seleccionamos el nodo *A* por ser el mejor de los nodos hijos. Seguidamente aceptamos el nodo *A* como nuevo nodo actual en sustitución de *B*, debido a que el valor de la función de evaluación heurística de *A* es mejor o igual que el de *B* ($5 \leq 20$).

- PASO 3: Expandimos el nodo actual *A* y generamos sus hijos: {*J*(2), *K*(1)}. Seleccionamos el nodo *K* por ser el mejor de los nodos hijos. A continuación aceptamos el nodo *K* como nuevo nodo actual en sustitución de *A*, debido a que el valor de la función de evaluación heurística de *K* es mejor o igual que el de *A* (se cumple que $1 \leq 5$).

- PASO 4: Expandimos el nodo actual *K*, generando sus nodos hijos {*I*(1)}. Seleccionamos el nodo *I* por ser el mejor de los nodos hijos. A continuación aceptamos el nodo *I* como nuevo nodo actual en sustitución de *K*, debido a que el valor de la función de evaluación heurística de *I* es mejor o igual que el de *K* ($1 \leq 1$).

La búsqueda terminaría en este punto. El algoritmo devolvería el nodo *I*(1) como mejor nodo encontrado.

Algoritmo del Temple simulado

Algoritmo del temple simulado: pág. 372

Ejercicio 1. (Valoración: 2.5 puntos)
Describa detalladamente las características del algoritmo de "temple simulado".

El temple simulado es un método de búsqueda local. La búsqueda local se suele emplear en problemas en los que el nodo meta por sí sólo contiene toda la información sobre la solución de un problema y no es necesario considerar la información proveniente del camino que ha llevado a la meta.

La búsqueda local consiste en realizar cambios en el estado actual que nos lleven a un estado vecino de cara a acabar en el estado meta tras una serie de iteraciones. Se suele partir de un estado inicial y se calculan los estados vecinos del actual mediante una **regla de vecindad**. Los vecinos del nodo actual son evaluados y se selecciona normalmente el mejor de ellos que cumpla cierto **criterio de aceptación**. Dicho vecino pasa a ser el nodo actual en la siguiente iteración de la búsqueda local. Este proceso se repite hasta que se cumpla cierto **criterio de finalización**; por ejemplo, que se haya realizado un determinado número de iteraciones o que no se hayan producido mejoras en la calidad del nodo actual durante las últimas iteraciones.

La búsqueda local realiza una búsqueda en un espacio de estados a los que se puede asociar una función objetivo. La función objetivo puede contener numerosos óptimos locales, lo cual puede dificultar enormemente la consecución del óptimo global. En cualquier caso, la búsqueda local es un método eficiente que puede encontrar soluciones aceptablemente buenas en un tiempo reducido. Además, se puede combinar con otros métodos de búsqueda más sofisticados, como pueden ser los algoritmos evolutivos. De forma ideal, la regla de vecindad debería cumplir la denominada "regla de conectividad", según la cual desde cualquier estado se debe poder alcanzar el objetivo óptimo mediante una secuencia de transformaciones.

El método de temple simulado tiene las siguientes peculiaridades:

- Realiza una selección aleatoria entre los vecinos del nodo actual.
- El criterio de aceptación permite admitir, con una cierta probabilidad, algunas transiciones entre estados en las que empeore el valor de la función objetivo. Dicha probabilidad depende de dos parámetros: la temperatura T y el incremento de energía ΔE = *coste*(*vecino-seleccionado*) – *coste*(*estado-actual*). Esto evita caer en óptimos locales durante la búsqueda. En cualquier caso, si el valor de la función objetivo (o "coste") del estado vecino elegido fuera mejor que el valor del estado actual ($\Delta E < 0$), la transición se realizaría siempre al igual que en el caso del algoritmo de escalada.
- Al principio de la búsqueda, la temperatura tiene un valor alto, de modo que la probabilidad de aceptar un estado peor que el actual sea grande. La temperatura va decreciendo a lo largo del proceso de búsqueda según un determinado plan de enfriamiento, que puede ser más rápido o más lento. Por tanto, a medida que la búsqueda progresa, cada vez es más probable que sólo se admitan soluciones que mejoren o igualen a la actual.
- Generalmente, la probabilidad de llegar a una buena solución mediante el temple simulado es mayor si la temperatura inicial es alta y el plan de enfriamiento es lento. Sin embargo, en dicho caso la búsqueda consume demasiado tiempo, ya que el criterio de terminación suele ser que la temperatura alcance un valor suficientemente pequeño.

Un inconveniente del temple simulado es que requiere un ajuste apropiado de sus parámetros en función del problema abordado, sobre todo del plan de enfriamiento, de cara a que el proceso de búsqueda resulte eficaz y se obtenga una buena solución. Este ajuste depende fuertemente de la forma de la función objetivo y de la distribución de sus óptimos locales, es decir, del problema particular que estemos intentando resolver.

Búsqueda tabú

Búsqueda tabú: pág. 373

Ejercicio 2. (Valoración: 2.5 puntos)
Describa detalladamente las características de la búsqueda tabú.

La búsqueda tabú es un método de búsqueda local. La búsqueda local se suele emplear en problemas en los que el nodo meta por sí sólo contiene toda la información sobre la solución de un problema y no es necesario considerar la información proveniente del camino que ha llevado a la meta.

La búsqueda local consiste en realizar cambios en el estado actual que nos lleven a un estado vecino de cara a acabar en el estado meta tras una serie de iteraciones. Se suele partir de un estado inicial y se calculan los estados vecinos del actual mediante una **regla de vecindad**. Los vecinos del nodo actual son evaluados y se selecciona normalmente el mejor de ellos que cumpla cierto **criterio de aceptación**. Dicho vecino pasa a ser el nodo actual en la siguiente iteración de la búsqueda local. Este proceso se repite hasta que se cumpla cierto **criterio de finalización**; por ejemplo, que se haya realizado un determinado número de iteraciones o que no se hayan producido mejoras en la calidad del nodo actual durante las últimas iteraciones.

La búsqueda local realiza una búsqueda en un espacio de estados a los que se puede asociar una función objetivo. La función objetivo puede contener numerosos óptimos locales, lo cual puede dificultar enormemente la consecución del óptimo global. En cualquier caso, la búsqueda local es un método eficiente que puede encontrar soluciones aceptablemente buenas en un tiempo reducido. Además, se puede combinar con otros métodos de búsqueda más sofisticados, como pueden ser los algoritmos evolutivos. De forma ideal, la regla de vecindad debería cumplir la denominada "regla de conectividad", según la cual desde cualquier estado se debe poder alcanzar el objetivo óptimo mediante una secuencia de transformaciones.

El método de búsqueda tabú tiene las siguientes peculiaridades:

- Dispone de un mecanismo de memoria (lista tabú) para evitar la generación de algunos vecinos. Por ejemplo, la lista tabú puede contener los últimos vecinos visitados o ciertos criterios de transformación de nodos.
- Se pueden aplicar excepciones a lo que indica la lista tabú mediante el denominado "criterio de aspiración". Un criterio de aspiración muy utilizado consiste en admitir nodos tabú que mejoren al mejor nodo encontrado hasta el momento.

Un inconveniente de la búsqueda tabú es que requiere un ajuste apropiado de sus parámetros en función del problema abordado (sobre todo del tamaño de la lista tabú, del criterio para la construcción de la lista tabú o de la definición del criterio de aspiración), de cara a que el proceso de búsqueda resulte eficaz y se obtenga una buena solución. Este ajuste depende fuertemente de la forma de la función objetivo y de la distribución de sus óptimos locales, es decir, del problema particular que estemos intentando resolver.

Ejercicios de examen

Septiembre 2016

Ejercicio 2. (Valoración: 2 puntos)
Explique brevemente si cada una de las cuatro afirmaciones siguientes es realmente verdadera o falsa. Utilice ejemplos sencillos que ilustren su explicación.

(1) En el algoritmo A*, si para todo nodo "n" se cumple que h(n) es mayor o igual que h*(n), entonces este algoritmo encuentra el camino solución óptimo.

(2) La complejidad espacial de la búsqueda primero en profundidad es, en el peor caso, exponencial con la profundidad de la solución (o con la profundidad límite).

(3) En la búsqueda bidireccional, las dos búsquedas realizadas deben ser en anchura para garantizar que se corten (o encuentren) en algún momento.

(4) El algoritmo IDA* realiza búsquedas primero en profundidad iterativamente desde el nodo raíz.

SOLUCIÓN por Severino Fernández Galán:

(1) Falso. La aseveración sería verdadera si se sustituye "mayor o igual" por "menor o igual". En la explicación conviene describir qué es h y qué es h*, tal como aparece en la sección 9.3 del texto base de la asignatura, y poner un ejemplo muy sencillo de grafo donde h sea menor o igual que h* para todos los nodos. Por ejemplo, una opción válida es aquélla en la que h=0 para todos los nodos.

(2) Falso. Es lineal con la profundidad de la solución (o con la profundidad límite). La explicación con ejemplos se encuentra en la sección 8.4.1.2 del texto base de la asignatura o, de forma alternativa, en uno de los vídeos creados por el equipo docente en el plan de trabajo del curso virtual (Bloque II).

(3) Falso. Sólo una de las dos búsquedas debe ser en anchura para garantizar que se corten. La otra puede ser, por ejemplo, en profundidad. El alumno puede hacer un dibujo sencillo para ilustrar esto.

(4) Verdadero. Basta con que el alumno haga una muy breve explicación de este algoritmo (tal como aparece en la sección 9.4.1 del texto base de la asignatura) o, alternativamente, realice un sencillo dibujo mostrando que el menor valor de $f=g+h$ de los nodos descartados en la búsqueda en profundidad de la iteración actual será el límite de la búsqueda en profundidad de la siguiente iteración.

Junio 22 semana 2

Ejercicio 2. (Valoración: 2.5 puntos)
Explique brevemente si cada una de las cuatro afirmaciones siguientes es realmente verdadera o falsa. Utilice ejemplos sencillos que ilustren su explicación.

(1) El algoritmo IDA* realiza búsquedas primero en profundidad iterativamente desde el nodo raíz.

(2) En el algoritmo A*, si para todo nodo "n" se cumple que h(n) es mayor o igual que h*(n), entonces este algoritmo encuentra el camino solución óptimo.

(3) La búsqueda primero en profundidad iterativa y la búsqueda primero en anchura iterativa poseen la misma complejidad espacial.

(4) La búsqueda primero en anchura iterativa puede encontrar una solución que no sea la más próxima al estado inicial.

***** Buscar solución en el libro *****

Junio 24 semana 2

Ejercicio 1. (Valoración: 2.5 puntos)
Explique brevemente cada una de las distintas formas posibles de utilización (o funcionamiento) de la lista ABIERTA tanto en métodos de búsqueda sin información como en métodos de búsqueda heurística.

***** Buscar solución en el libro *****

Septiembre 24

Ejercicio 1. (Valoración: 2.5 puntos)
Explique breve y razonadamente si son completos y si son admisibles cada uno de los diferentes métodos de búsqueda sin información y cada uno de los diferentes métodos de búsqueda heurística.

***** Buscar solución en el libro *****

Junio 25 semana 1

Ejercicio 1. (Valoración: 2.5 puntos)
Describa detalladamente las diferencias y semejanzas entre el algoritmo de escalada (o máximo gradiente) y el algoritmo A* (o "A estrella"). (NOTA IMPORTANTE: Esta pregunta no se refiere en absoluto a que el alumno describa uno de los algoritmos y a continuación describa el otro de forma separada. Esta pregunta pretende que se establezcan comparaciones detalladas entre cada uno de los aspectos importantes de los dos algoritmos.)

***** Buscar solución en el libro *****

Junio 25 semana 2

Ejercicio 1. (Valoración: 2.5 puntos)
Realice un estudio comparativo entre los siguientes métodos de búsqueda: IDA* y búsqueda en profundidad iterativa. Para ello analice semejanzas y diferencias en relación a los siguientes aspectos: tipo general de búsqueda que realizan, dependencia de la búsqueda del coste de los operadores, funcionamiento detallado de cada paso del algoritmo, admisibilidad y complejidad.

***** Buscar solución en el libro *****