7 BÚSQUEDA EN GRAFOS

Una gran cantidad de problemas pueden ser clasificados como búsquedas, i.e., su solución consiste en precisar dónde se encuentra cierta información dentro de un espacio de posibilidades determinado. La clase de estos problemas es suficientemente importante como para ser estudiada como un tema en sí, en especial, si se puede dar una estructura de grafo dirigido al espacio de posibles soluciones.

Los algoritmos correspondientes a soluciones de problemas de búsqueda en grafos son -naturalmente- esquemáticos o paradigmáticos. Por su generalidad, es difícil estimar complejidades espaciales, temporales o de otra clase, de estos algoritmos. Más aun, las soluciones que se presentan demandan, en primera instancia, demasiados recursos. En la práctica son algoritmos que deben utilizarse buscando especializar las soluciones a los casos específicos, de manera que las características propias del problema que se trate puedan ser aprovechadas para mejorar la pobre eficiencia de los métodos generales.

Por otra parte, un algoritmo de búsqueda que sepa aprovechar características específicas del espacio en el que se encuentran las posibles soluciones puede llegar a ser muy eficiente, o bien, lo más eficiente de que se disponga. La conveniencia o inconveniencia de los métodos aquí presentados debe evaluarse en cada caso particular; en muchas ocasiones, su utilización se debe a que no se conocen otros métodos -menos generales- para resolver el problema en cuestión.

7.1 EL PROBLEMA DE BÚSQUEDA

En general, el *problema de búsqueda* consiste en buscar un elemento que cumpla una determinada condición dentro de un conjunto dado.

Se utilizará la siguiente notación:

`SOLPOS` : *espacio* o *conjunto de soluciones posibles*
`sat: SOLPOS` → **`bool`** : *criterio de satisfacción*
`SOL = {x∈ SOLPOS | sat(x) }` : *soluciones*

En otras palabras, el problema consiste en encontrar uno o varios elementos del conjunto de soluciones posibles `SOL`. Toda solución cumple el criterio de satisfacción `sat(.)`, que no es otra cosa que una expresión formal de la condición que define, precisamente, las cosas que se buscan.

7.1.1 Problema de búsqueda en grafos

La solución de problemas de búsqueda se puede estudiar en forma algorítmica si, para empezar, se conoce una estructura para el conjunto de posibles soluciones, que permita establecer cómo se efectúa la búsqueda en sí. Por razones prácticas, puede ser conveniente ampliar el conjunto de soluciones a un conjunto que lo contenga, que se llamará *espacio de búsqueda*, donde sea posible establecer un cierto orden en la búsqueda en sí.

Quizás lo más general es pensar que hay una relación de precedencia entre los elementos del espacio de búsqueda, que corresponde, *grosso modo*, al orden en que las posibles soluciones son consideradas. El precisar esta relación de precedencia equivale a dotar al espacio de búsqueda de una estructura de grafo dirigido, de modo que se puede hablar de un *problema de busqueda en grafos*.

Más concretamente, para conformar un problema de búsqueda en grafos, a partir de un problema de búsqueda dado, se debe:

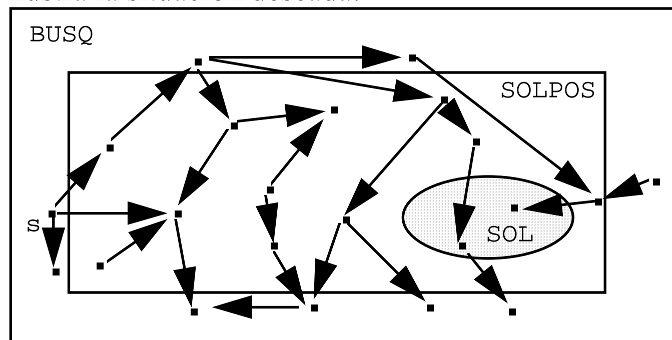
- Sumergir `SOLPOS` en un conjunto `BUSQ` (*espacio de búsqueda*), i.e. `BUSQ` $\supseteq$ `SOLPOS`. El criterio `sat` debe extenderse a `BUSQ`, demodo que `sat(u)` no vale en `BUSQ\SOLPOS`.

- Definir una relación binaria sobre `BUSQ`, digamos $.\rightarrow.$.
  De esta manera, `G(BUSQ,→)` es un grafo dirigido.
  La relación $.\rightarrow.$ debe ser tal, que el grafo `G` tenga ramificación finita.

- Para $u \in$ `BUSQ`: $SUC(u) = \{v \in BUSQ \mid u \rightarrow v\}$ (es finito!)
  $SUC^*(u) = \{v \in BUSQ \mid u \rightarrow^* v\}$
  $SUC^+(u) = \{v \in BUSQ \mid u \rightarrow^+ v\}$

- En `BUSQ` se define un valor `s` que sirva como *punto de partida* de la búsqueda. No necesariamente es un elemento de `SOLPOS`. Pero sí debería pasar que todo punto en `SOLPOS` fuera $\rightarrow$-alcanzable desde `s`.

  La siguiente figura ilustra la situación deseada:



- Se pueden definir funciones:

  $primh: BUSQ \rightarrow BUSQ^{\perp}$ $\quad$ $sigh: BUSQ \times BUSQ \rightarrow BUSQ^{\perp}$

donde:
`primh(v) =` "primer hijo de `v`"
`sigh(v,u) =` "siguiente hermano de `u`, considerando a `v` como padre"
El valor $\perp$ corresponde a la no existencia de un primer hijo o un siguiente hermano.
Con estas funciones se puede construir `SUC(u)`, para $u \in$ `BUSQ`.

Ahora el problema queda planteado al conocer `G, sat` y `s`. Formalmente:
"Dado `P = (G,s,sat)`, encontrar $x \in$ `SOLPOS`: `sat(x)` y $s \rightarrow^* x$ "

Si la condición de que `SOLPOS` sea alcanzable desde el punto de partida es verdadera, el problema se puede plantear:
"Dado `P = (G,s,sat)`, encontrar $x \in$ `SOLPOS`: `sat(x)`"

Ejemplo

Problema: colocar 4 damas en un tablero de ajedrez de 4x4, de forma que no se ataquen entre sí.

$SOLPOS = \{<x_1,x_2,x_3,x_4> | \text{ Para } i=1,\ldots,4: 1 \le x_i \le 4\}$

$sat(x_1,x_2,x_3,x_4) \equiv \forall i,j: 1 \le i < j \le 4: |x_i - x_j| \ne |i-j|$

El espacio de búsqueda y el punto de comienzo `s` se pueden definir como:

$BUSQ = \{<x_1,\ldots,x_k> | \ 0 \le k \le 4, \text{ Para } i=1,\ldots,4: 1 \le x_i \le 4\}$

`s = <>`

Los arcos del grafo se definen:

$<x_1,\ldots,x_k> \rightarrow <x_1,\ldots,x_k,x_{k+1}> \text{ si } 0 \le k < 4, \ 1 \le x_{k+1} \le 4.$

Otra posibilidad:

$SOLPOS1 = \{<x_1,x_2,x_3,x_4> | \text{ Para } i,j=1,\ldots,4: 1 \le x_i,x_j \le 4, \ x_i \ne x_j\}$

etc.

7.1.2 Estrategias de búsqueda en grafos

Los algoritmos cuentan con los siguientes elementos:

- Un *frente de exploración*
- Una manera de calcular `SUC`
- Una *estrategia de control* , para decidir qué vértices se deben explorar.
  Típicamente:
  - *Irrevocable* : algoritmos voraces. Puede no encontrar lo que busca.
  - *Tentativo* : no descartar nodos para exploración posterior.
    * *Reintento* (inglés: *backtracking*)
    * *Grafo explícito* : el frente de exploración va explicitando el grafo de búsqueda.

7.2 ALGORITMOS DE AGENDA

Buscar con un régimen determinado por las funciones `primh` y `sigh`, cuidando de no repetir vértices ya visitados. Para esto se *marcan* los vértices que se exploran.

Es un método de fuerza bruta. Complejidad: `O(n+e)`, donde `n =|BUSQ|, e = |→|`. Así, si `BUSQ` es infinito, esta clase de procesos puede no terminar. Nótese que si `BUSQ` es finito, el número de arcos también debe serlo: en este caso, el proceso debe terminar.

Los algoritmos iterativos pueden entenderse como algoritmos de agenda. El manejo de la agenda, como estructura de datos, influye en el control del programa y determina el régimen de búsqueda. Por ejemplo, si se escoge siempre el último nodo visitado para continuar la búsqueda por sus hijosd, se obtiene una búsqueda por profundidad (preorden). Puede usarse cualquier otro recorrido, v.gr. por anchura, etc.

En general, la forma del algoritmo es:

```
[  Ctx: SOL ⊆ SOLPOS ⊆ SUC*(s);
   x,MARCADOS,AGENDA:= s,{s},SUC(s);
   {inv:     SOL ⊆ {x}∪SUC*(AGENDA)  ∧  ¬sat(MARCADOS\{x})
          ∧  AGENDA ⊆ SUC(MARCADOS)  ∧  MARCADOS ⊆ SUC*(s) }
   do¬sat(x) ∧  AGENDA≠∅
          →   x:= algún(u | u∈AGENDA);
              AGENDA:= AGENDA \ {x};
              if   x∈MARCADOS    →   skip
              []   x∉MARCADOS    →   MARCADOS:= MARCADOS ∪ {x};
                                     AGENDA:=AGENDA ∪ SUC(x)
              fi
   od
   {Pos: sat(x) ∨  SOL=∅ }   ]
```

Los distintos regímenes de control se concretan cuando se explica la forma en que se calculan las operaciones que manejan la agenda, i.e., cómo se escoge un elemento (`algún`) y cómo se unen nuevos elementos a la agenda.

Por ejemplo, si la agenda se maneja como una pila, seleccionar un elemento corresponde a elegir el último que entró en la agenda. Si además la unión de `AGENDA` con `SUC(x)` se hace agregando los elementos (con `push`) en el orden contrario a `sigh(x,.)`, se obtiene un régimen de búsqueda por profundidad.

¿Cómo se haría un recorrido por anchura?

7.2.1 Efecto dominó

Supóngase que se cuenta con un predicado adicional

```
dom: BUSQ → bool
```

tal que:

```
[d1]  sat(v)   ⇒   dom(v)
[d2] ¬dom(v)   ⇒   ∀u∈SUC*(v): ¬dom(u).
```

Como también `¬dom(v) ⇒ ¬sat(v)`, se tiene, además:

```
¬dom(v)   ⇒   ∀u∈SUC*(v): ¬sat(u).
```

Entonces el algoritmo básico de agenda puede cambiarse a:

```
[  Ctx: SOL ⊆ SOLPOS ⊆ SUC*(s) ∧ [d1] ∧ [d2];
   x,MARCADOS,AGENDA:= s,{s},SUC(s);
   {inv:     SOL ⊆ {x}∪SUC*(AGENDA)  ∧  ¬sat(MARCADOS\{x})
          ∧  AGENDA ⊆ SUC(MARCADOS)  ∧  MARCADOS ⊆ SUC*(s) }
   do¬sat(x) ∧  AGENDA≠∅
          →   x:= algún(u | u∈AGENDA);
              AGENDA:= AGENDA \ {x};
              if   x∈MARCADOS    →   skip
              ▯    x∉MARCADOS    →   MARCADOS:= MARCADOS ∪ {x};
                                     if dom(x)  → AGENDA:=AGENDA ∪ SUC(x)
                                     ▯  ¬dom(x) → skip
                                     fi
              fi
   od
   {Pos: sat(x) ∨  SOL=∅ }
]
```

De esta manera la agenda no crece innecesariamente, con nodos para los que se puede afirmar que no tienen sucesores que sean soluciones.

Pequeña posible ineficiencia:

No se daña la corrección, pero es posible que algún vértice implícitamente desechado por efecto dominó sea posteriormente visitado, por ser sucesor de otro nodo para el cual el criterio dominó sea verdadero.

Ejemplo: Coloración de mapas

Se trata de colorear un mapa de `n` países.

```
COLOR = {1,2,3,4}                    (4 colores bastan!)
PAIS = {1,2,...,n}
vecinos: PAIS x PAIS →bool           (conocida!)
SOLPOS = {<c1,...,cn>| ci∈COLOR, 1≤i≤n}
sat(c1,...,cn) ≡ ∀i,j: 1≤i<j≤n ⇒ [ci=cj ⇒ ¬vecinos(i,j)]

BUSQ =  {<c1,...,ck>| ci∈COLOR, 1≤i≤k≤n, k≥0}
<c1,...,ck> → <c1,...,ck,c>    ,para 0≤k<n, c∈COLOR
   SUC*(c1,...,ck) = <c1,...,ck,...>
sat(c1,...,ck)  ≡   F                                        , si k<n,
                    ∀i,j: 1≤i<j≤n ⇒ [ci=cj ⇒ ¬vecinos(i,j)]   , si k=n.
dom(c1,...,ck)  ≡   ∀i,j: 1≤i<j≤k ⇒ [ci=cj ⇒ ¬vecinos(i,j)]
```

Nótese que se cumplen las condiciones de dominó:

```
[d1] sat(c1,...,cn)  ⇒   dom(c1,...,cn)
[d2] ¬dom(c1,...,ck) ⇒   Para u∈SUC*(c1,...,ck): ¬dom(u).
```

Una solución, manejando la agenda como una pila:

```
{Pre: T }
x,MARCADOS,AGENDA:= <>,[<>],[<1>,<2>,<3>,<4>];
do ¬sat(x) ∧ ¬isempty(AGENDA)
     →   x:= head(AGENDA);
         AGENDA:= tail(AGENDA);
         if  x∈MARCADOS → skip
         ▯   x∉MARCADOS → MARCADOS:= MARCADOS ∪ {x};
                          if  dom(x)   →   AGENDA:= AGENDA ∪ SUC(x)
                          ▯   ¬dom(x)  →   skip
                          fi
         fi
od
{Pos: sat(x) ∨ SOL = ∅ }
```

Mejoras: (1) Siempre hay solución; entonces no hay que verificar si la agenda se vacía.
(2) No hay ciclos: no es necesario recordar los marcados.

Otra solución:

```
{Pre: T }
x,AGENDA:= <>,[<1>,<2>,<3>,<4>];
do ¬sat(x)
     →   x:= head(AGENDA);
         AGENDA:= tail(AGENDA);
         if  dom(x)   →  AGENDA:= AGENDA ∪ SUC(x)
         ▯   ¬dom(x)  →  skip
         fi
od
{Pos: sat(x)}
```

Ejercicio
Explique cómo se calculan:

- dom(x)
- AGENDA ∪ SUC(x).

7.3 BÚSQUEDA HEURÍSTICA

Se ha expuesto un método de búsqueda en grafos que puede funcionar si se cuenta con las siguientes premisas:

- El grafo G es de ramificación finita (o bien, la relación .→. es localmente finita), i.e., SUC(x) es finito, para cualquier $x \in$ BUSQ.
- Naturalmente, SOL≠∅.

El algoritmo en cuestión usa una agenda para guardar nodos tales que ellos o sus descendientes pueden ser soluciones. Para evitar no-terminación debida a ciclos en la visita de nodos, el algoritmo marca los nodos ya visitados. Adicionalmente se puede pensar en la disponibilidad de un criterio dominó dom, que permita desechar subárboles que tengan como raíz un nodo para el que se sepa que ni él ni sus descendientes son soluciones. Llámese Beste algoritmo básico.

El algoritmo B es parcialmente correcto con respecto a la especificación señalada. Sin embargo, no se garantiza terminación, de modo que no se puede hablar de corrección total.

Puesto que se trata del recorrido de un grafo de ramificación finita, se puede afirmar que la única razón por la que el algoritmo B puede no terminar es la existencia de un camino infinito desde el nodo de comienzo s[1]. Esto significa que la agenda nunca se vacía. Incluso, puede contener soluciones que no son elegidas nunca en la instrucción

---

[1] Un famoso resultado, conocido en teoría de grafos como el Lema de König, afirma que un grafo dirigido de ramificación finita tiene un camino infinito a partir de un nodo x, si y sólo si x tiene infinitos descendientes. La no

```
x:= algún( u | u∈AGENDA ).
```

En realidad, la forma en que se implante esta última instrucción da lugar a diversas variantes del algoritmo B, que pueden terminar o no. Por ejemplo, si se escogen siempre los nodos que están a menor distancia de s el algoritmo busca "por anchura" (inglés: *breadth-first*). Este método avaro o voraz (inglés: *greedy*) es utilizado con éxito por el algoritmo de Dijkstra, que puede considerarse una versión importante del algoritmo B.

Para plantear el algoritmo de Dijkstra como un caso especial del algoritmo B, se puede explicar el problema de la ruta mínima entre dos nodos -una de las aplicaciones del algoritmo de Dijkstra- como un problema de búsqueda en grafos. Así, se puede pensar que se quiere encontrar un nodo z, conectado por un camino con el nodo de comienzo s. Adicionalmente, se desea que el costo sea mínimo, considerando los posibles caminos de s a z.

El algoritmo B no habla de costos. Para el caso de Dijkstra, puede añadirse al invariante la afirmación adicional de que para los nodos marcados los costos encontrados son mínimos. Como al final z es marcado, el costo que se encuentra es mínimo. Por otra parte, la agenda se vacía, ya que el grafo es finito: es decir, se garantiza la terminación.

El método avaro puede ser costoso y demorado. Es un método ciego frente al posible buen futuro que podría conllevar la elección de un nodo que, sin ser el más cercano al nodo de comienzo, sí se aproximara más a una solución que cualquiera de los demás nodos de la agenda. De hecho, la situación ideal sería disponer de una función

```
F: BUSQ → VALOR
```

que midiera el costo de la búsqueda de una solución a partir de un elemento de BUSQ. Entonces, se podría implantar:

```
algún(u | u∈AGENDA) = x     :⇔      F(x) = <min u: u∈AGENDA : F(u)>
```

En la práctica, no se conoce la función F. Sin embargo, se pueden estudiar condiciones para las que el algoritmo B se comporta bien. Hay dos niveles de satisfacción cuando se estudian implantaciones del algoritmo B que consideran costos: se puede desear encontrar una solución, aunque no necesariamente la menos costosa, o bien, se puede querer encontrar siempre una solución óptima. Se estudiará la posibilidad de determinar condiciones suficientes para afirmar que una variante del algoritmo B termine y encuentre soluciones de costo mínimo.

Para x∈BUSQ, sean:

```
f*(x) := <min c: a∈SOL, c = (s→*x→*a): costo(c)>
```

"costo mínimo de un camino de s a una solución, pasando por x"

```
g*(x) := <min c: c = (s→*x): costo(c)>
```

"costo mínimo de un camino de s a x"

---

terminación del algoritmo no puede darse por repetir nodos ya visitados, puesto que éstos se marcan para evitar este problema. Es decir, s debe tener infinitos descendientes y , por tanto, debe existir un camino infinito a partir de s.

```
h*(x) := <min c: a∈SOL, c = (x→*a): costo(c)>
```

"costo mínimo de un camino de x a una solución"

Es claro que, para x∈BUSQ:

```
f*(x) = g*(x) + h*(x).
```

Cuando existen soluciones, nótese que f*(s) es el costo de cualquier solución de costo mínimo.

Las funciones g* y h* son desconocidas en el caso general. En cambio, se puede pensar que se estiman con un par de funciones f y g, es decir:

g(x) : "costo estimado de un camino de costo mínimo de s a x"

h(x) : "costo estimado de un camino de costo mínimo de x a una solución"

Así, se define una *función de evaluación*:

```
f(x) :=  g(x) + h(x)
```

"costo estimado de un camino de costo mínimo de s a una solución, pasando por x".

Con estas definiciones, se implanta una variante del algoritmo básico B, denominada en la literatura "algoritmo A", para la que:

```
algún(u | u∈AGENDA) = x     :⇔       f(x) = <min u: u∈AGENDA : f(u)>
```

Para g(x)es natural definir el costo en que se haya incurrido para encontrar que x es un descendiente de s. Como es posible que g(x) no sea óptimo, se tiene, en el caso general, que, para x∈BUSQ:

```
g(x) ≥  g*(x).
```

La función h(x) es una *heurística*. La denominación proviene del hecho de que su definición adecuada depende, en cada caso, del conocimiento específico que se tenga del espacio de búsqueda, a veces apoyado en razones más bien intuitivas.

Por ejemplo, cuando h = 0, se tiene que f = g. En este caso, el algoritmo A busca por anchura y coincide con el algoritmo de Dijkstra. Encuentra una solución de costo óptimo, incluso si el grafo es infinito. A pesar de ser seguro, como ya se anotó, este método puede ser demasiado demorado, por expandir muchos nodos antes de encontrar una solución.

Más generalmente, se verá que si h es una cota inferior para h*, i.e., para x∈BUSQ:

```
h(x) ≤ h*(x)
```

se tendrá también el resultado de que el algoritmo A encuentra una solución óptima. En este caso, la literatura denomina el algoritmo correspondiente A* y dice de la heurística h que es *admisible*. También se dice de un algoritmo de búsqueda que es *admisible* si termina y siempre lo hace con una solución de costo mínimo, cuando exista una solución.

Supóngase que se cuenta con un algoritmo A* para un problema de búsqueda que tiene una solución alcanzable desde s. Se probará que A* es admisible. El siguiente lema es fundamental para probar el resultado deseado:

Lema 1

El ciclo de A* mantiene invariante, además de lo ya anotado:

$$[\neg sat(x) \wedge AGENDA \neq \emptyset] \Rightarrow \exists x \in AGENDA: [f(x) \leq f^*(s) \wedge f^*(x)=f^*(s)]$$

Es decir, mientras no se termine, siempre habrá un nodo sobre un camino óptimo, en la agenda cuya estimación sea inferior al costo de una solución de costo mínimo.

Demostración:

Considérese un camino $<x_0,x_1,...,x_r>$, tal que $x_0=s$ y x_r es una solución de costo mínimo desde s. Si el algoritmo no ha terminado, se puede probar que también $AGENDA \cap \{x_0,x_1,...,x_r\} \neq \emptyset$:

Al empezar, $s \in AGENDA$, y se tiene que:

$$\begin{aligned} f(s) &= g(s) + h(s) \\ &= h(s) && \text{, porque } g(s) = 0 \\ &\leq h^*(s) && \text{, porque h es admisible} \\ &\leq f^*(s). \end{aligned}$$

Si $AGENDA \cap \{x_0,x_1,...,x_r\}=\emptyset$, como un nodo sale de la agenda sólo si es marcado y entran a la agenda sus sucesores, entonces cada uno de los $x_i \in MARCADOS$. En particular, la solución x_r ha sido marcada. Como el invariante afirma que entre los marcados, excepto el nodo de turno x, no hay soluciones, x_r debe ser igual a x y A* ha terminado. Este no es el caso, porque se está suponiendo $\neg sat(x)$.

Sea x el primero de los x_i que está en AGENDA. Todos los ancestros de x_i están marcados y, sobre el camino, hasta este punto, debe valer que $g = g^*$.

Entonces:

$$\begin{aligned} f(x) &= g(x) + h(x) \\ &= g^*(x) + h(x) \\ &\leq g^*(x) + h^*(x) \\ &= f^*(x) \\ &= f^*(s). \end{aligned}$$

[]

Ahora se puede probar la terminación de A*:

Lema 2

Sea c = <inf e: e arco de G: costo(e)>.
Supóngase $c>0$ y que hay una solución alcanzable desde s.
Entonces, A* termina.

Demostración:

Supóngase que A* no termina. A* no incurre en ciclos infinitos de ejecución por el hecho de visitar más de una vez un mismo nodo, ya que los nodos se marcan para evitar esto. Por tanto, por tratarse de un grafo de ramificación finita, debe haber un camino infinito desde s.

En cualquier caso, sea d*(x) la longitud del camino más corto de s a x. Entonces:

```
f(x) = g(x) + h(x)
     ≥ g(x)              , se supone: 0 ≤ h(x)
     ≥ g*(x)
     ≥ d*(x)c
```

Por tanto, la función f no está acotada sobre el camino infinito que se ha supuesto que exista. Los nodos de este camino se van escogiendo sucesivamente, puesto que la función f tiene valor minimal sobre ellos dentro de la agenda. Sin embargo, el Lema 1 afirma que, dentro de la agenda, debe haber siempre un nodo con estimación menor o igual a f*(s). De este modo, el camino infinito no puede ser expandido. Por tanto A*debe terminar.

[]

Teorema 3
Sea c = <inf e: e arco de G: costo(e)>.
Supóngase $c>0$ y que hay una solución alcanzable desde s. Entonces, A* es admisible.

Demostración:

A* puede terminar porque encuentra una solución o porque la agenda se vacía. Este último caso no puede darse antes de encontrar una solución, porque el Lema 1 garantiza que si hay una solución, la agenda debe contener un nodo sobre un camino óptimo. Es decir, A* debe terminar encontrando una solución.

Además, la solución que encuentre A* debe ser de costo mínimo. De lo contrario, si A* terminara en una solución y que no fuera de costo mínimo, se tendría:

```
f(y) = g(y)
     > f*(s)
     ≥ f(x)          , para cierto x∈AGENDA (última iteración, Lema 1)
     ≥ f(y)          (!!)
```

[]

Corolario

Cualquier nodo $n \in AGENDA$, para el que $f(n)<f^*(s)$ debe haber sido expandido en algún paso de la búsqueda.

Demostración

Supóngase que se termina en un nodo u, que es solución de costo mínimo y que hay un nodo $n \in AGENDA$ para el que $f(n)<f^*(s)$, que no fue expandido. Entonces:

$f(u) \leq f(n)$, porque u fue expandido en el último paso

$< f^*(s)$, por hipótesis

$= f^*(u)$, porque u es solución de costo mínimo.

Ahora, expresando estas desigualdades en términos de las funciones g, h, g^*, h^*:

$$g(u)+h(u) \leq g(n)+h(n) < g^*(u)+h^*(u).$$

De este modo:

$g(u) < g^*(u) + (h^*(u)-h(u))$

$\leq g^*(u)$, porque h es admisible

$\leq g(u)$, porque g^* minimiza costos

Y se tendría que $g(u)<g(u)$. Esta contradicción prueba el corolario.

□

7.3.1 Comparación de algoritmos A*

Supónganse dos algoritmos A^*, que usan funciones de evaluación

$$f_i = g_i + h_i \quad , \; i=1,2$$

donde cada h_i se supone admisible. Se dice que el algoritmo A_2 es *más informado* que el algoritmo A_1, si

$$h_2 > h_1.$$

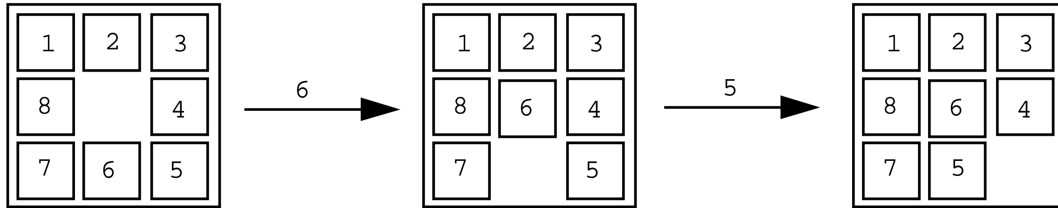
Ejemplo

Considérese un juego del tipo *taquin* , en el que se tiene un tablero de la forma

1	2	3
8		4
7	6	5

El cuadro central es el *blanco* de la posición mostrada, que se llamará la *configuración objetivo*.. A él se puede mover cualquiera de los cuadros (fichas) vecinos en sentido horizontal o vertical, de

modo que en la nueva posición, el blanco ocupará el sitio de la ficha movida. Por ejemplo, las siguientes son dos configuraciones resultantes de mover, a partir de la anterior, primero el `6` y después el `5`:



El acertijo puede plantearse de la siguiente manera: se comienza en una configuración *inicial*, que proviene de haber efectuado algunas movidas desde la configuración objetivo; se desea encontrar una sucesión de movidas que transformen la configuración inicial en la objetivo.

En términos de un problema de búsqueda:

```
SOLPOS = {tab | "tab es un tablero posible" }
sat(tab) ≡  "tab = configuración_objetivo"
BUSQ = SOLPOS
tab1 → tab2   ≡    "hay una movida que lleva de tab1 a tab2"
s : cualquier elemento de SOLPOS.
c(tab1,tab2)  = 0  , si tab1 = tab2
              = 1  , si tab1 → tab2
              = ∞  , en otro caso.
```

Heurísticas para la búsqueda

- h_0(tab) = 0
  Efectuará una búsqueda por anchura en el grafo. Encontrará la solución más barata, pero demorará mucho para hallarla.

- h_1(tab) = "número de fichas que aun no están en el puesto correcto"
  Es una heurística admisible, porque para que lleguen al puesto correcto será necesario efectuar al menos una movida, aun en el caso de la solución más corta. Es decir: $h_1(tab) \leq h^*(tab)$.

- h_2(tab) = "suma de distancias de fichas a su puesto correcto"
  Las distancias se entienden en el sentido de la "métrica de la ciudad", i.e., la distancia entre dos puestos es la suma de las diferencias, en valor absoluto, de las respectivas abscisas y ordenadas. De nuevo, es una heurística admisible, porque en la solución h_2 es nula y, para que una ficha llegue al puesto correcto será necesario efectuar tantas movidas como su distancia, aun en el caso de la solución más corta. Es decir: $h_2(tab) \leq h^*(tab)$.

En orden de información, se tiene que:

$$h_0 \leq h_1 \leq h_2.$$

Se mostrará que, cuando hay solución, un algoritmo más informado expande menos nodos que uno menos informado. Así, la heurística h_2 es más rápida para encontrar la solución que las otras dos.

Sin embargo, es la más dispendiosa de calcular. En el caso general, ante dos heurísticas admisibles, debe encontrarse un balance entre el costo de calcular la heurística y el ahorro que se consigue al usar la más informada.

Lema 4
Sean A_1 y A_2 dos versiones del algoritmo A*, con heurísticas respectivas h_1 y h_2. Supóngase que A_2 es más informado que A_1. Si hay una solución desde el nodo inicial s, todo nodo expandido en la búsqueda por A_2 también debe haber sido expandido por A_1.

Demostración:

Sean N_i, $i=1,2$, los conjuntos de nodos expandidos por cada versión del algoritmo A*. El resultado se probará por inducción sobre la profundidad de cada nodo en N_2 en el grafo de búsqueda desde s (i.e., la distancia en arcos desde s).

El único nodo de profundidad 0 es, precisamente, s, el cual está tanto en N_2 como en N_1. Supóngase que el resultado vale para nodos de profundidad j, con $0 \leq j \leq k$.

Considérese un nodo n, de profundidad k+1. Todos los N_2-ancestros de n tienen profundidad menor o igual a k, de modo que estos nodos, por la hipótesis de inducción, están en N_1. Por tanto, n es uno de los nodos en el grafo de búsqueda de A_1 (aunque no necesariamente sea expandido). Así, debe haber un camino de la raíz a n en cada uno de los grafos de búsqueda; más aun, el costo del camino encontrado por A_1, al poder pasar por más nodos (por hipótesis de inducción, los nodos de nivel inferior o igual de N_1 son, por lo menos, los de N_2 de nivel inferior o igual a k), debe ser inferior o igual al encontrado por A_2, i.e.,

$$g_1(n) \leq g_2(n).$$

Supóngase que A_2 expande el nodo n pero que A_1 no lo expande. Por supuesto, al terminar n debe estar dentro de los nodos marcados por A_1. Como éste termina en un nodos de costo minimal f*(s) sin expandir el nodo n, debe cumplirse que

$$f_1(n) \geq f^*(s) \qquad \text{, por el Corolario del Teorema 3}$$
$$\Rightarrow g_1(n) + h_1(n) \geq f^*(s)$$
$$\Rightarrow h_1(n) \geq f^*(s) - g_1(n)$$
$$\Rightarrow h_1(n) \geq f^*(s) - g_2(n).$$

Ahora, como A_2 expande n, debe valer que

$$f_2(n) \leq f^*(s) \qquad \text{, por el Lema 1}$$
$$\Rightarrow g_2(n) + h_2(n) \leq f^*(s)$$
$$\Rightarrow h_2(n) \leq f^*(s) - g_2(n)$$
$$\Rightarrow h_2(n) \leq h_1(n)$$

Esta última afirmación contradice el hecho de que h_2 sea más informado que h_1.

□

7.3.2 La restricción de monotonía

Una función heurística h satisface la *restricción de monotonía* si $h(u)=0$ en cada solución de costo mínimo y para cada par de nodos n,n', tales que $n \to n'$, se cumple que:

$$h(n) - h(n') \leq costo(n,n').$$

La restricción de monotonía es deseable por el siguiente resultado:

Lema 5

Si la restricción de monotonía vale, A* encuentra un camino de costo mínimo para cada nodo que expande.

Demostración

Sea n un nodo expandido por A*. Debe probarse que $g(n)=g^*(n)$.

Si $n=s$, se tiene trivialmente que $g(n)=g^*(n)=0$. Si $n \neq s$, considérese un camino $\langle n_0,\dots,n_k \rangle$ de $n_0=s$ hasta $n_k=n$, de costo mínimo. Sea n_u el último nodo de este camino que está en MARCADOS\AGENDA cuando A* selecciona a n para ser expandido (obsérvese que $u<k$). Como hipóteisi de inducción, se puede suponer que $g(n_u)=g^*(n_u)$ y, por lo tanto, también $g(n_{u+1})=g^*(n_{u+1})$, ya que el costop en que se incurre para llegar a n_{u+1} es el del nodo anterior más el costo del arco que los une, que forma parte de un camino óptimo. Obsérvese que $n_{u+1} \in AGENDA$ en el momento en que se selecciona a $n \in AGENDA$.

Ahora, para $i=0,\dots,k-1$:

$$\begin{aligned} g^*(n_i)+h(n_i) &\leq g^*(n_i)+costo(n_i,n_{i+1})+h(n_{i+1}) \quad \text{, por monotonía} \\ &= g^*(n_{i+1})+ h(n_{i+1}). \end{aligned}$$

Esta última conclusión se puede afirmar, porque n_i,n_{i+1} están sobre un camino óptimo. Se puede continuar hasta el nodo $n_k = n$. En particular, se puede afirmar que:

$$g^*(n_{u+1})+h(n_{u+1}) \leq g^*(n_k)+h(n_k)$$

$$\Rightarrow g(n_{u+1})+h(n_{u+1}) \leq g^*(n)+h(n).$$

$$\Rightarrow f(n_{u+1}) \leq g^*(n)+h(n)$$

$$\Rightarrow f(n_{u+1}) \leq g^*(n)+h(n) \leq g(n)+h(n) = f(n)$$

Como al escoger n no se escoge n_{u+1}, debe ser que estos dos valores son iguales. Por tanto, $g^*(n)=g(n)$.

▯

Lema 6

Si se cumple la restricción de monotonía, los valores de la secuencia de nodos expandidos por el algoritmo A* son no decrecientes.

Demostración

Supóngase que n′es expandido inmediatamente después de n. Si n′ estaba en la agenda cuando se expandió n, es claro que $f(n) \leq f(n')$. En otro caso, n′ tampoco está en MARCADOS, porque ya hubiera sido expandido.

Entonces n′ fue añadido a la agenda por la expansión de n. Es decir, $n \rightarrow n'$. Cuando n′ es expandido:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g^*(n') + h(n') && \text{, por Lema 5} \\ &= g^*(n) + costo(n,n') + h(n') \\ &= g(n) + costo(n,n') + h(n') && \text{, por Lema 5} \\ &\geq g(n) + costo(n,n') + h(n) - costo(n,n') && \text{, por monotonía} \\ &= f(n). \end{aligned}$$

□

7.3.3 Otros usos de heurísticas

Las heurísticas pueden utilizarse para maximizar beneficios en forma simétrica a la de minimizar costos. En este caso, $f^*(x)$ debe interpretarse como el valor máximo de beneficio que puede conseguirse partiendo desde x. Si se puede dar una cota superior del máximo beneficio al que se puede aspirar, digamos B, entonces el máximo se alcanzará de la misma manera en que se puede conseguir el mínimo de la función $f(x) = B - f^*(x)$, y la misma teoría ya expuesta puede ser utilizada.

Por ejemplo, el problema del morral, ya resuelto con programación dinámica, puede solucionarse, en forma aproximada, con esta técnica:

Se tienen

- n objetos, $o_1, \ldots, o_n$.
- Para $i = 1, \ldots, n$, p_i es el peso de o_i.
- Se tiene un morral, que soporta un peso máximo P.
- Para $i = 1, \ldots, n$, si se carga en el morral el objeto o_i, se obtiene un beneficio o utilidad u_i.
- Se quiere maximizar la utilidad de cargar en el morral algunos de los objetos.

Sin pérdida de generalidad, se puede suponer que los objetos se han ordenado de manera que, para $i=1,2,\ldots,n-1$:

$$\frac{u_i}{p_i} \geq \frac{u_{i+1}}{p_{i+1}}$$

Es decir, los primeros objetos reportan más utilidad por unidad de peso.

Para plantear el problema como búsqueda:

$$SOLPOS = \{<x_1,\dots,x_n> \mid 1\le i\le n: x_i\in\{0,1\},\ <+i: 1\le i\le n: p_i x_i> \ \le P\}$$

$$BUSQ = \{<x_1,\dots,x_k> \mid 0\le k\le n,\ 1\le i\le k: x_i\in\{0,1\},\ <+i: 1\le i\le k: p_i x_i>\le P\}$$

$$sat(x_1,\dots,x_n) \equiv <+i: 1\le i\le n: p_i x_i> \ \le P$$

(no necesariamente el máximo!)

$$s = <>$$

$$<x_1,\dots,x_k> \to <x_1,\dots,x_k,x_{k+1}> \quad \text{(si las tuplas están en BUSQ)}$$

Para dirigir la búsqueda:

$$g(x_1,\dots,x_k) := <+i: 1\le i\le k: p_i x_i>$$

$$h(x_1,\dots,x_k) := \begin{cases} (P - <+i: 1\le i\le k: p_i x_i>)\ \dfrac{u_{k+1}}{p_{k+1}} & \text{, si } P - <+i: 1\le i\le k: p_i x_i> \ge p_{k+1} \\ 0 & \text{, si } P - <+i: 1\le i\le k: p_i x_i> < p_{k+1} \end{cases}$$

Es decir, h mide la máxima utilidad posible, suponiendo que la capacidad restante del morral se usa para el objeto o_{k+1}.

7.3.3 Ejercicios

1 Genere, sin repeticiones, todos los subconjuntos de k elementos del conjunto $\{1,2,\dots,n\}$.

2 Genere, sin repeticiones, todas las descomposiciones de un número entero positivo en sumandos no nulos.

3 Genere, sin repeticiones, todas las particiones del conjunto $\{1,2,\dots,n\}$.

4 Encuentre una configuración con un mínimo de damas en un tablero de ajedrez, que dominen todos los cuadros del tablero.

5 Dado un entero positivo n, considere un "tablero de ajedrez" de lado n. Decida si existe una forma de recorrer todas las casillas del tablero con movimientos de caballo, sin pasar más de una vez por cada cuadro.

6 (*Asignación de labores*) Se deben asignar, de manera biyectiva, n personas a n labores, $n>0$. El costo de asignar la i-sima persona al j-simo trabajo es c_{ij}. Una *asignación* es una biyección entre personas y labores, y su costo es la suma de los costos c_{ij} correspondientes. Encuentre una asignación de costo mínimo.

7 (*Programación de tareas*) Un computador secuencial debe efectura un conjunto de n tareas, $T_1, \dots, T_n$. La k-sima tarea requiere t_k unidades de tiempo y cuesta $C_k(t)$, donde t es el tiempo en que termina su ejecución (la primera tarea se comienza a ejecutar en el tiempo 0). Cada función C_k se supone monótona creciente. Encuentre un orden de ejecución que minimice el costo de efectuar todas las tareas.