

ALGORITMOS DE LOS MÉTODOS DE BÚSQUEDA EN LA INTELIGENCIA ARTIFICIAL

INTRODUCCIÓN

Las técnicas de búsqueda son una serie de esquemas de representación del conocimiento, que mediante diversos algoritmos nos permite resolver ciertos problemas desde el punto de vista de la inteligencia artificial

La I.A. proporciona de diversas técnicas de búsqueda que tienen un enunciado matemático, la cual hace posible su implementación.

Las técnicas de búsqueda son una serie de esquemas de representación del conocimiento, que mediante diversos algoritmos nos permite resolver ciertos problemas desde el punto de vista de la I.A.

Según el autor Marvin Minsky la *búsqueda no informada* se desarrolla en pequeños dominios, podemos intentar aplicar los métodos de *mindless search...*, pero no es práctico porque la búsqueda se vuelve enorme, y la *búsqueda informada* se utiliza para reducir la extensión de la búsqueda anterior y debe incorporarle tipos adicionales de conocimiento, incorporando la experiencia en resolución de problemas.

BÚSQUEDAS NO INFORMADAS¹

Los algoritmos de **búsqueda ciega o no informada** no dependen de información propia del problema a la hora de resolverlo, sino que proporcionan métodos generales para recorrer los árboles de búsqueda asociados a la representación del problema, por lo que se pueden aplicar en cualquier circunstancia. Se basan en la estructura del espacio de estados y determinan estrategias sistemáticas para su exploración, es decir, que siguen una estrategia fija a la hora de visitar los nodos que representan los estados del problema. Se trata también de algoritmos exhaustivos, de manera que, en el peor de los casos, pueden acabar recorriendo todos los nodos del problema para hallar la solución.

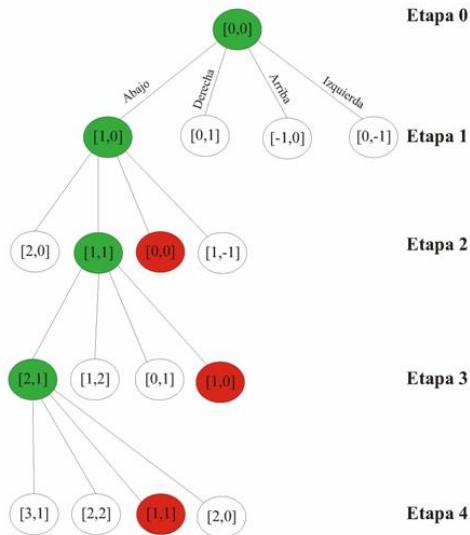
Existen, básicamente, dos estrategias de recorrido de un espacio de búsqueda, en **anchura** y en **profundidad**. Los algoritmos que veremos a continuación se basan en una de las dos (o incluso en las dos). El problema principal que tienen es que, al ser exhaustivos y sistemáticos su coste puede ser prohibitivo para la mayoría de los problemas reales, por lo tanto solo serán aplicables en problemas pequeños, pero presentan la ventaja de que no es necesario ningún conocimiento adicional sobre el problema, por lo que siempre son aplicables.

Vamos a repasar únicamente algoritmos de búsqueda ciega para árboles, que serán variantes del algoritmo de búsqueda general (para árboles) que se describe a continuación:

¹ <http://www.cs.us.es/~fsancho/?e=95>

BÚSQUEDA GENÉRICA (O SIN ESTRATEGIA)

Los espacios de búsqueda, en general, pueden ser representados como árboles o como



grafos. La diferencia principal entre los algoritmos que encontraremos para unos u otros radica, fundamentalmente, en que los algoritmos para árboles no necesitan mantener una lista de los nodos por los que ya ha pasado, ya que no pueden volver a visitarse nodos en el recorrido de búsqueda por el árbol. Sin embargo, si trabajamos con grafos podemos encontrar caminos que repitan nodos, y en este caso necesitaremos de una memoria de almacenamiento que nos indique si ya hemos visitado ese nodo en una etapa anterior o no.

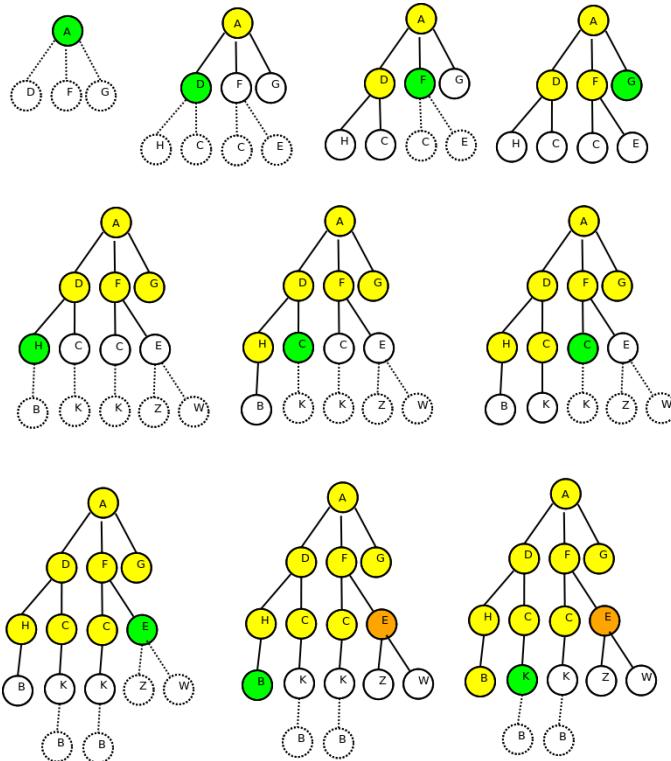
Durante una búsqueda, la **frontera** es la colección de nodos que esperan ser visitados (habitualmente almacenados en forma de pila, cola, lista...). Los nodos de la frontera se dicen **abiertos**. Cuando un nodo se elimina de la frontera, se etiqueta como **cerrado**.

En el algoritmo de búsqueda genérica no se proporciona ninguna estrategia para el recorrido de los nodos, simplemente es un patrón que indica que "existe una forma" (no determinada) de construir los caminos que empiezan en el nodo raíz y acaban en las hojas. Sobre este algoritmo general, cada uno de los que veremos a continuación añade únicamente una estrategia específica para decidir cómo se construyen dichos caminos. Dejaremos los algoritmos para grafos para cuando introduzcamos heurísticas, ya que los que veremos aquí sacan provecho del hecho de que trabajamos con árboles y se comportan muy mal (hasta el punto de poder ser inútiles) cuando los aplicamos sobre grafos.

BÚSQUEDA EN ANCHURA

La idea principal de la **Búsqueda en Anchura** (*Breadth-First-Search-BFS*) consiste en visitar todos los nodos que hay a profundidad i antes de pasar a visitar aquellos que hay a profundidad $i+1$. Es decir, tras visitar un nodo, pasamos a visitar a sus hermanos antes que a sus hijos. Si usamos estructuras de programación habituales, una posible implementación para BFS se haría almacenando el conjunto de nodos abiertos como una cola, a la que se accede por un procedimiento FIFO (el primero que entra es el primero que sale). Cuando hagamos uso de estructuras de programación basadas en agentes veremos que la

implementación de este algoritmo es ligeramente distinta, y mantienen el almacenamiento haciendo uso de la propia estructura que existe entre los agentes involucrados.



Este algoritmo es **completo**, es decir, si existe solución, este algoritmo la encuentra. Más aún, es **óptimo**, en el sentido de que si hay solución, encuentra una de las soluciones a distancia mínima de la raíz.

Respecto al tratamiento de nodos repetidos, se comporta bien. Si el nodo generado actual ya apareció en niveles superiores (más cerca de la raíz), el coste actual será peor ya que su camino desde la raíz es más largo, y si está al mismo nivel, su coste será el mismo. Esto quiere decir que si nos encontramos un nodo que ya ha sido repetido, su coste será peor o igual que algún nodo anterior visitado o no, de manera que lo podremos descartar, porque o lo hemos expandido ya o lo haremos próximamente con mejor coste.

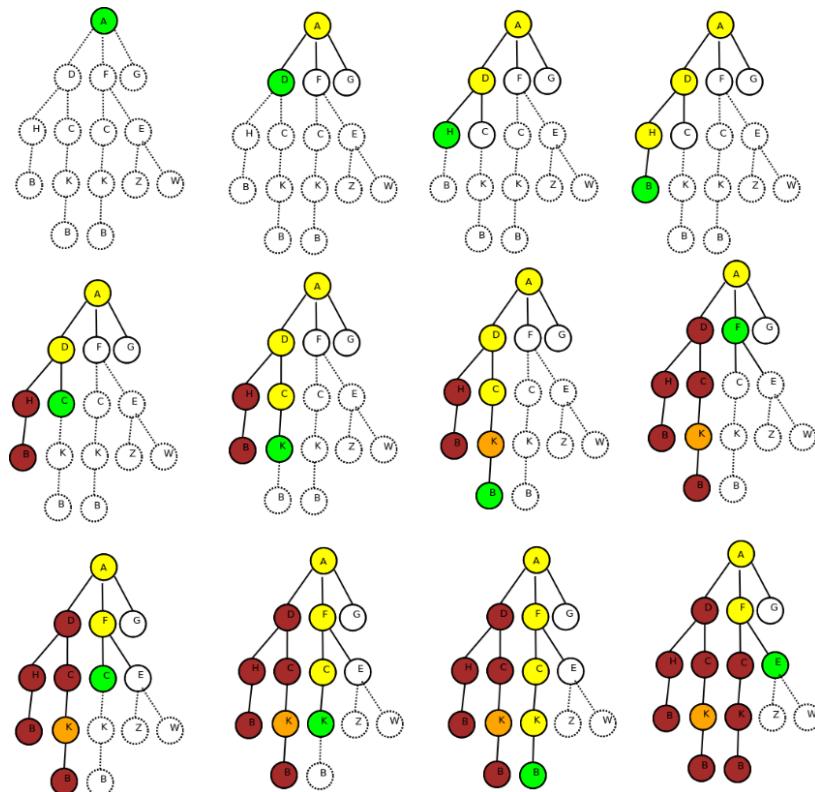
Éstas son las buenas noticias la mala noticia es que en términos de **tiempo**, lo que tarda es:

$$1 + b + b^2 + \dots + b^d = \sum_{i=0}^d b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$$

En términos de **espacio**, BFS almacena una lista con todos los nodos en todas las profundidades, y a distancia d esta lista es de longitud b^d . Lo que supone una cantidad exponencial de nodos. Por tanto, tanto en espacio como en tiempo su complejidad es exponencial en d .

BÚSQUEDA EN PROFUNDIDAD

Al igual que en el caso de la búsqueda en anchura, la búsqueda en profundidad (*Depth-First-Search - DFS*)² también puede ser vista como un proceso por niveles, pero con la diferencia de que, tras visitar un nodo, se visitan sus hijos antes que sus hermanos, por lo que el algoritmo tiende a bajar por las ramas del árbol hacia las hojas antes de visitar cada una de las ramas posibles. De nuevo, si hacemos uso de estructuras clásicas de programación, DFS se puede implementar por medio de una pila accediendo a sus elementos por un proceso de LIFO (último en entrar, primero en salir).



Como en el caso del BFS, la complejidad en tiempo del DFS es del orden de bd . Es exponencial ya que en el peor caso DFS tiene que visitar todos los nodos. Sin embargo, la complejidad en espacio es lineal en d , donde d es la longitud del camino más largo posible, ya que el máximo número de nodos que se almacenan es del orden bd .

DFS no es ni óptimo ni completo. No es óptimo porque si existe más de una solución, podría encontrar la primera que estuviese a un nivel de profundidad mayor, y para ver que no es completo es necesario irse a ejemplos en los que el espacio de búsqueda fuese infinito: *supongamos un robot que puede moverse a izquierda o derecha en cada paso y ha de encontrar un objeto; la búsqueda en profundidad le obligaría a moverse indefinidamente en una sola dirección, cuando el objeto podría estar en la dirección contraria (e incluso a un solo paso del origen)*. Para evitar este problema es común trabajar con una pequeña variante de este algoritmo que se llama de **Profundidad limitada**, que impone un límite máximo al nivel alcanzado.

² Richard E. Korf, 'Depth-first Iterative-Deepening: An Optimal Admissible Tree Search', journal Artificial Intelligence, 1985, volume 27, pages 97–109

Procedimiento: Busqueda en profundidad limitada (limite: entero)

```
Est_abiertos.insertar(Estado inicial)
Actual ← Est_abiertos.primero()
mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
    Est_abiertos.borrar_primer()
    Est_cerrados.insertar(Actual)
    si profundidad(Actual) ≤ límite entonces
        Hijos ← generar_sucesores (Actual)
        Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Est_abiertos.insertar(Hijos)
    fin
    Actual ← Est_abiertos.primero()
fin
```

Para el tratamiento de posibles nodos repetidos la idea consiste en lo siguiente: si el nodo generado actual está repetido en niveles superiores (más cerca de la raíz), su coste será peor ya que su camino desde la raíz es más largo, si está al mismo nivel, su coste será el mismo. En estos dos casos podemos olvidarnos de este nodo. En el caso de que el repetido corresponda a un nodo de profundidad superior, significa que hemos llegado al mismo estado por un camino más corto, de manera que deberemos mantenerlo y continuar su exploración, ya que nos permitirá llegar a mayor profundidad que antes.

La versión del mismo algoritmo usando recursión sería:

Función: Busqueda en profundidad limitada recursiva (actual:nodo, límite: entero)

```
si profundidad(Actual) ≤ límite entonces
    para todo nodo ∈ generar_sucesores (Actual) hacer
        si es_final?(nodo) entonces
            retorna (nodo)
        sino
            resultado ← Busqueda en profundidad limitada
                recursiva(nodo, límite)
            si es_final?(resultado) entonces
                retorna (resultado)
            fin
        fin
    fin
sino
    retorna ( $\emptyset$ )
fin
```

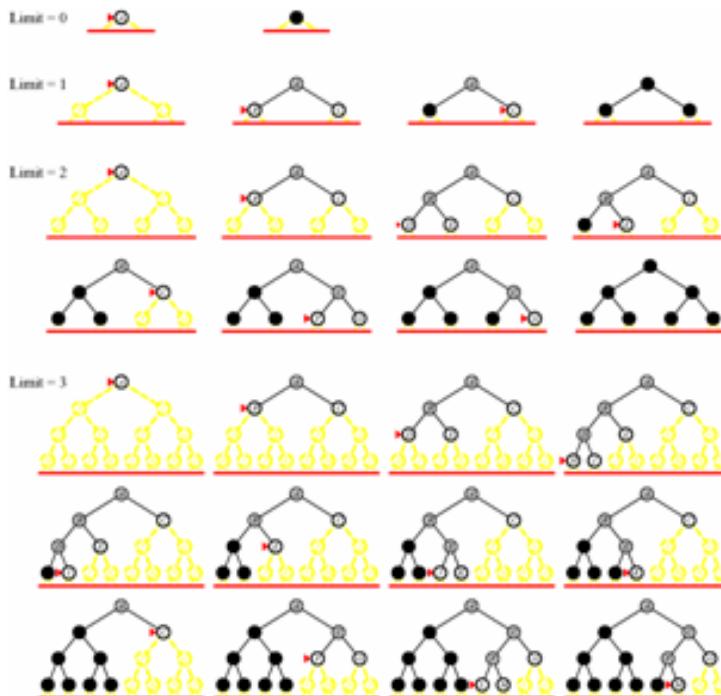
Las implementaciones recursivas de los algoritmos permiten por lo general un gran ahorro de espacio. En este caso, la búsqueda en profundidad se puede realizar recursivamente de manera natural. La recursividad permite que las alternativas queden almacenadas en la pila de ejecución como puntos de continuación del algoritmo sin necesidad de almacenar ninguna información. Así, el ahorro de espacio es proporcional al factor de ramificación que en la práctica en problemas difíciles no es despreciable. Lo único que perdemos es la capacidad de buscar repetidos en los nodos pendientes que tenemos en el camino recorrido,

pero habitualmente la limitación de memoria es una restricción bastante fuerte que impide resolver problemas con longitudes de camino muy largas.

En el caso de la búsqueda en profundidad, el tratamiento de nodos repetidos no es crucial ya que al tener un límite en profundidad los ciclos no llevan a caminos infinitos. No obstante, en este caso se pueden comprobar los nodos en el camino actual, ya que está completo en la estructura de nodos abiertos. Además, no tratando repetidos mantenemos el coste espacial lineal, lo cual es una gran ventaja. El evitar tener que tratar repetidos y tener un coste espacial lineal supone una característica diferenciadora de hace muy ventajosa a la búsqueda en profundidad, y permite obtener soluciones que se encuentren a gran profundidad.

PROFUNDIDAD ITERADA

El algoritmo de **Profundidad Iterada (ID)** mezcla los requerimientos espaciales del DFS (lineal en d) y las propiedades óptimas del BFS (completo y asintóticamente óptimo). La idea principal de este algoritmo es hacer una búsqueda DFS repetidamente sobre subárboles de profundidad 0, después de profundidad 1, después 2, etc.... hasta que se alcanza el objetivo.



El algoritmo ID es **óptimo y completo**, garantiza encontrar una solución, en caso de que exista y, si existen varias soluciones, devuelve una de las que tengan distancia mínima a la raíz.

Procedimiento: Busqueda en profundidad iterativa (límite: entero)

```

prof ← 1
Actual ← Estado inicial
mientras no es_final?(Actual) y prof
    Est_abiertos.inicializar()
    Est_abiertos.insertar(Estado inicial)
    Actual ← Est_abiertos.primero()

```

```

mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
    Est_abiertos.borrar_primer()
    Est_cerrados.insertar(Actual)
    si profundidad(Actual) ≤ prof entonces
        Hijos ← generar_sucesores (Actual)
        Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Est_abiertos.insertar(Hijos)
    fin
    Actual← Est_abiertos.primer()
fin
prof ← prof+1
fin

```

Aparentemente podría parecer que este algoritmo es más costoso que los anteriores al tener que repetir en cada iteración la búsqueda anterior, pero si pensamos en el número de nodos nuevos que recorremos a cada iteración, estos son siempre tantos como todos los que hemos recorrido en todas las iteraciones anteriores, por lo que las repeticiones suponen un factor constante respecto a los que recorreríamos haciendo solo la última iteración.

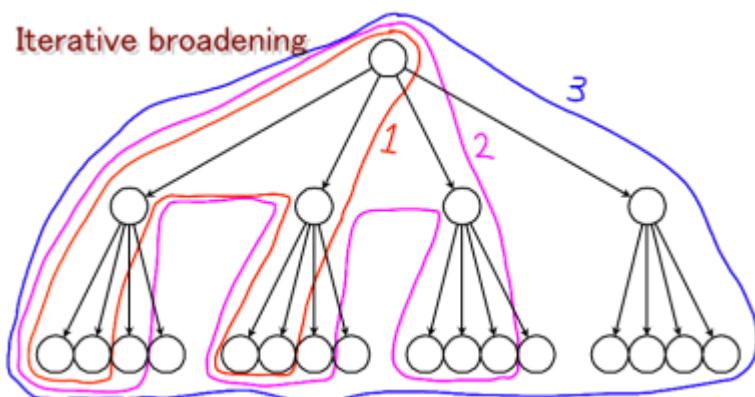
Respecto a la complejidad en tiempo, se comporta como DFS y BFS, del orden $O(b^d)$, esto es, en el peor caso es exponencial en d . ID visita los nodos a profundidad 0, $d+1$ veces, los que están a profundidad 1, d veces, ..., y los que están a profundidad d , 1 vez. Por tanto, el tiempo total requerido es:

$$1 + (1+b) + (1+b+b^2) + \dots + (1+b+b^2+\dots+b^d) = O(1+b+b^2+\dots+b^d) = O(b^d)$$

Igual que en el caso del algoritmo en profundidad, el tratar nodos repetidos acaba con todas las ventajas espaciales del algoritmo, por lo que es aconsejable no hacerlo. Como máximo se puede utilizar la estructura de nodos abiertos para detectar bucles en el camino actual.

ANCHURA ITERADA

El algoritmo en **Anchura Iterada (IB)** es análogo al ID, pero realiza una búsqueda DFS primero en los subárboles de anchura 1, después en los de anchura 2, ... hasta encontrar un nodo solución.

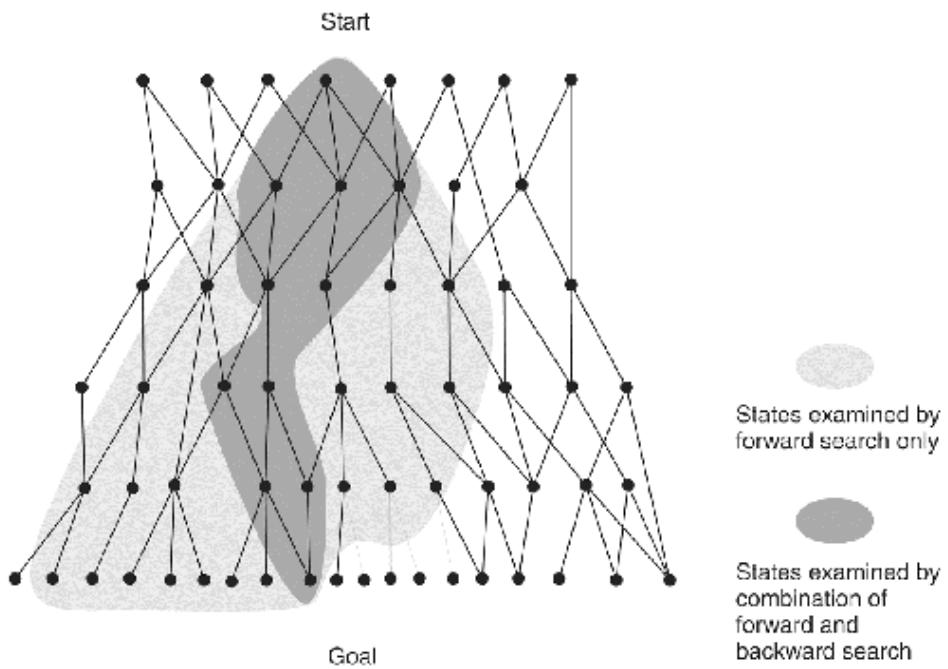


IB no es completo por las mismas causas que no lo era DFS. Como no ofrece muchas más novedades respecto a los ya vistos, no profundizaremos más en él, y sólo destacaremos que puede ser aconsejable su uso cuando el árbol tiene un valor de ramificación excesivamente alto.

BÚSQUEDA BIDIRECCIONAL

Un problema de **Búsqueda bidireccional** es una extensión del problema de búsqueda general planteado antes. Se define como una 5-tupla $(X, S, G, \delta, \gamma)$, donde (X, S, G, δ) es un problema de búsqueda básico y $\gamma: X \rightarrow 2^X$ es una función de **transición inversa**, es decir, que $\gamma(x)$ es el conjunto de predecesores de x .

En la búsqueda bidireccional (BB), tal y como sugiere su nombre, se realizan dos búsquedas simultáneas: una desde el estado inicial hasta el estado final, y otra desde el estado final hasta el estado inicial. La búsqueda global acaba cuando ambas búsquedas parciales se encuentran. Sin embargo, no todos los problemas de búsqueda básicos se pueden plantear de forma sencilla como problemas de búsqueda bidireccionales, muchas veces porque no es sencillo proporcionar la función de transición inversa.



Habitualmente, se suele implementar como un par de algoritmos BFS simultáneos, así que, como ellos, será óptimo y completo. Sin embargo, la BB mejora a BFS en términos de complejidad, ya que su complejidad en tiempo es exponencial en $d/2$ (cuando ambas búsquedas deben recorrer todos los nodos que le corresponden). Su complejidad en espacio es también del orden de $b^{b/2}$, ya que debe mantener una lista de todos los nodos a profundidad i , y el número de nodos a profundidad $d/2$ es $b^{b/2}$.

Como variante de este método, existe la **búsqueda dirigida por islas**, que es una generalización del BB en el que se establecen un conjunto de objetivos intermedios (**islas**), lo que suele reducir un problema de búsqueda grande en un conjunto de problemas de búsqueda menores. Si se van espaciando estas islas entre el nodo origen y el objetivo, se puede conseguir que su complejidad mejore, haciendo $mb^{d/m} \ll b^d$. Las cotas decrecen, pero suele ser difícil garantizar la optimidad, ya que habría que asegurar que todas las islas (u objetivos intermedios) están en el camino óptimo de la solución original.

El número de variantes que se han dado para mejorar la casuística de las búsquedas ciegas es enorme, y como nuestro objetivo es solo conocer las aproximaciones clásicas a la resolución de

problemas, no entraremos en más detalles y pasaremos a estudiar otros tipos de búsquedas que resulten más eficientes en general.

BÚSQUEDAS INFORMADAS³

Es evidente que los **algoritmos de búsqueda ciega** (o **no informada**) serán incapaces de encontrar soluciones en problemas en los que el tamaño del espacio de búsqueda sea grande, ya que todos ellos tienen un coste temporal que es exponencial en el tamaño del dato de entrada, por lo que cuando se aplican a problemas reales el tiempo para encontrar la mejor solución a un problema no es asumible.

Una de las formas para intentar reducir este tiempo de búsqueda a cotas más razonables pasa por hacer intervenir dentro del funcionamiento del algoritmo de búsqueda conocimiento adicional sobre el problema que se está intentando resolver. En consecuencia, perderemos en generalidad, a cambio de ganar en eficiencia temporal.

En estas líneas nos centraremos en aquellos algoritmos que introducen mecanismos para manipular esa información adicional que puede suponer la diferencia entre tardar un tiempo impracticable para hallar la solución óptima y poder encontrarla en tiempos razonables.

CALCULANDO COSTES PARA OPTIMIZAR

Al igual que se hace con la búsqueda ciega, hemos de definir qué entendemos por **búsqueda del óptimo**, por lo que hemos de dar alguna medida del coste de obtener una solución. En general, este coste se medirá sobre el camino que nos lleva desde el estado inicial del problema hasta el estado final. No siempre es una medida factible ni deseable, pero nos centraremos en problemas en los que esta medida del óptimo tenga sentido. Para poder calcular este coste de forma efectiva, tendremos un coste asociado a los operadores que permiten pasar de un estado a otro.

Los algoritmos que veremos utilizarán el coste de los caminos explorados para saber qué nodos merece la pena explorar antes que otros. De esta manera, perderemos la sistematicidad de los algoritmos de búsqueda no informada, y el orden de visita de los estados no vendrá determinado por su posición en el grafo de búsqueda, sino por su coste respecto del problema real que intentan resolver.

En la mayoría de los problemas, debido al tamaño del espacio de búsqueda, no podemos plantearnos generar todo el grafo asociado de una vez, sino que deber ser generado a medida que es explorado, por lo que se tienen dos elementos fundamentales que intervienen en el coste del camino que va desde el estado inicial hasta la solución que buscamos:

1. En primer lugar, tendremos **el coste del camino recorrido**, que podremos calcular simplemente sumando los costes de los operadores aplicados desde el estado inicial hasta el nodo actual. En consecuencia, este coste es algo que se puede calcular con exactitud.
2. En segundo lugar, tendremos un coste más difícil de determinar, **el coste del camino que nos queda por recorrer hasta el estado final**. Dado que lo desconocemos, tendremos que utilizar el conocimiento del que disponemos del problema para obtener una aproximación. Es aquí donde interviene el adjetivo de "**heurística**" que se aplica a este tipo de algoritmos de búsqueda.

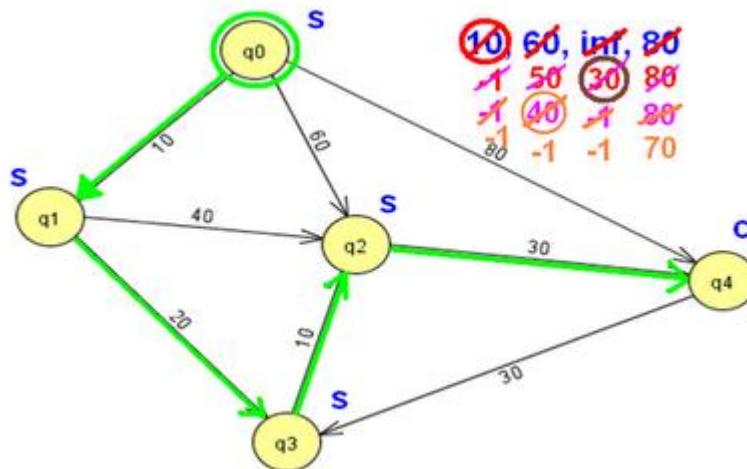
³ <http://www.cs.us.es/~fsancho/?e=62>

Evidentemente, la calidad de este conocimiento que nos permite aproximar el coste futuro hará más o menos exitosa nuestra búsqueda. Si nuestro conocimiento fuera perfecto, podríamos dirigirnos rápidamente hacia el objetivo descartando todos los caminos de mayor coste, eligiendo en cada momento el mejor estado posible, y dando la solución en tiempo lineal. En el otro extremo, si estuviéramos en la total ignorancia, tendríamos que explorar muchos caminos antes de hallar la solución óptima. Una situación similar a la que se encuentra en la búsqueda ciega.

Así pues, esta función heurística de coste futuro se vuelve fundamental en la resolución eficiente del problema, de forma que, cuanto más ajustada esté al coste real, mejor funcionarán los algoritmos que hagan uso de ella. El fundamento de los algoritmos de búsqueda heurística será el modo de elegir qué nodo explorar primero, para ello podemos utilizar diferentes estrategias que nos darán diferentes propiedades.

ALGORITMO VORAZ

Una primera estrategia, que representa una heurística muy limitada pero muy extendida, es utilizar como estimación futura el coste del siguiente paso que vamos a dar, que lo podemos calcular fácilmente desde el nodo actual en el que nos encontramos. En este caso, la heurística usada se resume en suponer que, en cada paso, el mejor camino se consigue al minimizar el coste inmediato.



Esta estrategia se traduce en el **algoritmo voraz (greedy best first)**, cuya única diferencia con respecto a los algoritmos ciegos es que utiliza alguna estructura ordenada de datos para almacenar los nodos abiertos (por ejemplo, una cola con prioridad), de forma que aquellos que supongan un coste inmediato menor se coloquen primero. Una posible implementación podría ser la siguiente:

Algoritmo: Voraz

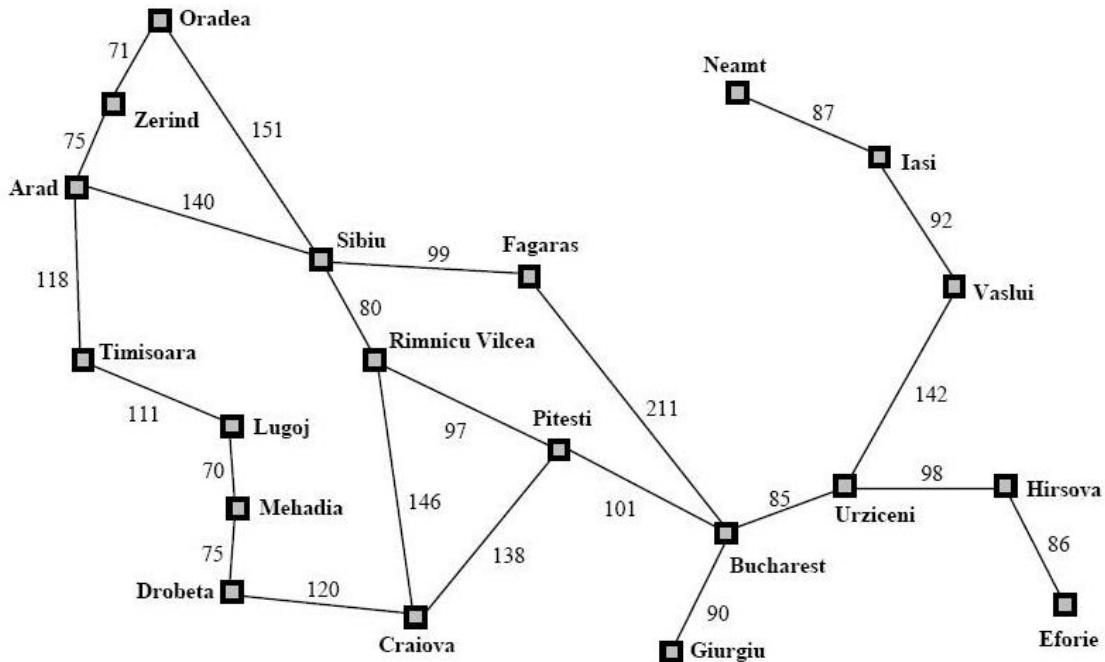
```

Insertar Estado_inicial Est_abiertos
Actual ← Primero Est_abiertos
mientras Actual no es_final? y Est_abiertos no vacía? hacer
    Quitar_primeros Est_abiertos
    Insertar Actual Est_cerrados
    hijos ← generar_sucesores_ordenados_por_peso (Actual)
    hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
    Insertar Hijos Est_abiertos
    Actual ← Primero Est_abiertos
fin
    
```

Aunque explorar antes los nodos vecinos más cercanos puede ayudar a encontrar una solución antes, es fácil dar ejemplos en los que no se garantice que la solución encontrada sea óptima.

EL ALGORITMO A*

Dado que nuestro objetivo no es solo llegar lo más rápidamente a la solución, sino encontrar la de menor coste tendremos que tener en cuenta el coste de todo el camino y no solo el camino recorrido o el camino por recorrer.



Para poder introducir el siguiente algoritmo y establecer sus propiedades es necesario conocer las siguientes definiciones:

- El **coste de una arista** entre dos nodos ni y nj es el coste del operador que nos permite pasar de un nodo al otro, y lo denotaremos como $c(ni,nj)$. Este coste siempre será positivo.
- El **coste de un camino** entre dos nodos ni y nj es la suma de los costes de todos los arcos que llevan desde un nodo al otro y lo denotaremos como:

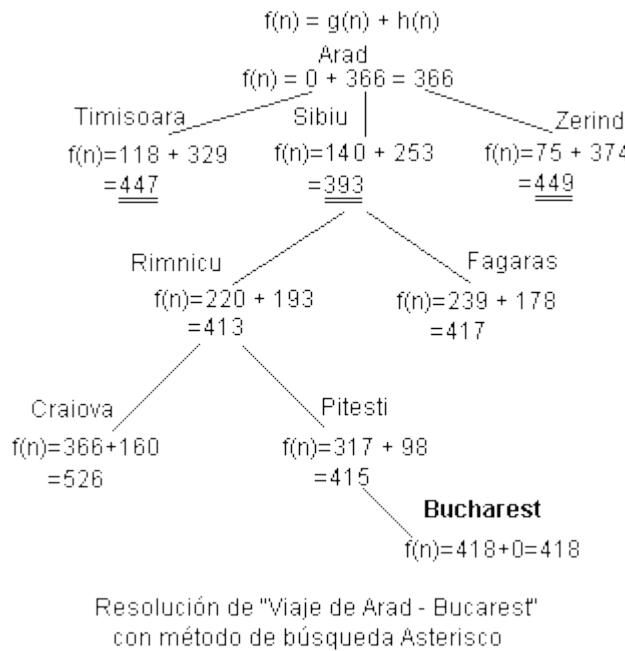
$$C(ni,nj)=\sum_{x=ij} c(nx,nx+1)$$

- El **coste del camino mínimo** entre dos nodos ni y nj (el del camino de menor coste de aquellos que llevan desde un nodo al otro) se denominará por:

$$K(ni,nj)=\min_k C_k(ni,nj)$$

- Si nj es un nodo terminal, para cada nodo ni notaremos $h^*(ni)=K(ni,nj)$, es decir, el coste del camino mínimo desde ese estado a un estado solución.

- Si n_i es un nodo inicial, para cada nodo n_j notaremos $g^*(n_j)=K(n_i,n_j)$, es decir, el coste del camino mínimo desde un estado inicial a ese estado.



Esto nos permite definir **el coste del camino mínimo** que pasa por cualquier nodo como la suma del coste del camino mínimo desde el nodo inicial y el coste del camino mínimo nodo hasta el nodo final:

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

El problema habitual suele ser que para un nodo cualquiera, n , el valor de $h^*(n)$ es desconocido, por lo que este valor hemos de **estimarlo** por una función que nos lo aproximará, a esta función la denominaremos $h(n)$ y le daremos el nombre de **función heurística**, que debe ser siempre un valor mayor o igual que cero. Denominaremos $g(n)$ al coste del camino desde el nodo inicial al nodo n , que en este caso sí es conocido ya que lo hemos recorrido en nuestra exploración. De esta manera tendremos una estimación del coste del camino mínimo que pasa por cierto nodo:

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

Este será el valor que utilizaremos para decidir en nuestro algoritmo de búsqueda cuál es el siguiente nodo que explorar de entre todos los nodos abiertos disponibles por medio del siguiente algoritmo, que denominaremos A^* :

Algoritmo: A^*

```

Insertar Estado_inicial Est_abiertos
Actual ← Primero Est_abiertos
mientras Actual no es_final? y Est_abiertos no vacía? hacer
    Quitar_primeros Est_abiertos
    Insertar Actual Est_cerrados
    hijos ← generar_sucesores_ordenados_por_heurística (Actual)
    hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
    Insertar Hijos Est_abiertos
    Actual ← Primero Est_abiertos
fin

```

Se puede observar que el algoritmo es prácticamente el mismo que el algoritmo voraz visto anteriormente, pero ahora la ordenación de los nodos se realiza utilizando el valor de la función heurística f general, y no el caso particular del coste inmediato. Por convención, suele considerarse que, a igual valor de f los nodos con h más pequeña se explorarán antes (simplemente, porque si la h es más pequeña es porque esos nodos parecen estar más cerca de la solución), y a igual h se consideran en cualquier orden (por ejemplo, el orden en el que se introdujeron en la cola), ya que no tenemos criterios para tomar una decisión.

Nunca está de más el remarcar que el algoritmo solo acaba cuando se extrae una solución de la cola. Es posible que en cierto momento ya haya en la estructura de nodos abiertos algunos nodos que sean solución, pero hasta que no se hayan explorado los nodos por delante de ellos, no podemos asegurar que realmente sean soluciones buenas. Debe recordarse que los nodos están ordenados por el coste estimado del camino total, si esta estimación en un nodo es menor es que podría pertenecer a un camino que representa una solución mejor. Si no tomáramos esta precaución, podríamos tener el mismo problema que presenta el algoritmo voraz.

En el peor de los casos el coste temporal de este algoritmo es también $O(r^p)$. Si, por ejemplo, la función $h(n)$ fuera siempre 0 el algoritmo se comportaría como una búsqueda en anchura gobernada por el coste del camino recorrido. De hecho, una posible interpretación de las funciones g y h es que gobiernan el comportamiento en anchura o profundidad del algoritmo.

Evidentemente, lo que hace que este algoritmo pueda tener un coste temporal inferior es la bondad de la función h . Cuanto más cercana sea al coste real (que habíamos denotado h^*), mayor será el comportamiento en profundidad del algoritmo, pues los nodos que aparentemente están más cerca de la solución se explorarán antes. En el momento en el que esa información deje de ser fiable, el coste del camino ya explorado hará que otros nodos menos profundos tengan un coste total mejor y, por lo tanto, se abrirá la búsqueda en anchura.

Si pensamos un poco en el coste espacial que tendrá el algoritmo, llegamos a la conclusión de que, debido a que puede llegar a comportarse como una búsqueda en anchura, el peor caso sería $O(r^p)$. Un resultado teórico que se deberá tener en cuenta es que el coste espacial del algoritmo A^* crece exponencialmente a no ser que se cumpla que:

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

Este resultado nos da una cota para posibles actuaciones: Si no podemos encontrar una función heurística suficientemente buena, deberíamos usar algoritmos que no buscasen el óptimo pero garantizasen el encontrar una buena solución.

El tratamiento de nodos repetidos en este algoritmo se realiza de la siguiente forma:

- Si el nodo que se repite está en la estructura de nodos abiertos:
 - Si su coste es menor, sustituimos el coste por el nuevo, lo que podrá variar su posición en la estructura de nodos abiertos.
 - Si su coste es igual o mayor, nos olvidamos del nodo, ya que no aporta nada mejor que lo que teníamos.
- Si el nodo que se repite está en la estructura de nodos cerrados:
 - Si su coste es menor, reabrimos el nodo insertándolo en la estructura de nodos abiertos con el nuevo coste, no hacemos nada con sus sucesores, ya que se reabrirán si hace falta.
 - Si su coste es mayor o igual, nos olvidamos del nodo.

¿CUÁNDO PODEMOS ENCONTRAR EL ÓPTIMO?

Hasta ahora no hemos hablado de cómo garantizar la optimalidad de la solución. Hemos visto que en el caso degenerado ($h=0$) tenemos una búsqueda en anchura guiada por el coste del camino explorado, eso nos debería garantizar el óptimo en este caso, aunque a costa de un mayor tiempo de búsqueda. Pero como hemos comentado, la función h nos permite introducir un comportamiento de búsqueda en profundidad, donde sabemos que no se garantiza el óptimo, pero a cambio podemos realizar la búsqueda en menor tiempo, porque podemos evitar explorar ciertos caminos.

Para saber si encontraremos o no el óptimo mediante el algoritmo A^* hemos de analizar las propiedades de la función heurística:

ADMISIBILIDAD

La propiedad clave que garantizará hallar la solución óptima es la que denominaremos **admisibilidad**:

Diremos que una función heurística h es **admisible** cuando su valor en cada nodo sea menor o igual que el valor del coste real del camino que falta por recorrer hasta la solución, es decir:

$$\forall n \ (0 \leq h(n) \leq h^*(n))$$

Esto quiere decir que la función heurística ha de ser un **estimador optimista** del coste que falta para llegar a la solución. Esta propiedad implica que, si podemos demostrar que la función de estimación h que utilizamos en nuestro problema la cumple, siempre encontraremos una solución óptima. El problema radica en hacer esa demostración, pues cada problema es diferente.

Para una discusión sobre técnicas para construir heurísticos admisibles se puede consultar el capítulo 4, sección 4.2, del libro “Inteligencia Artificial: Un enfoque moderno” de S. Russell y P. Norvig.

CONSISTENCIA

La **consistencia** se puede ver como una extensión de la admisibilidad. A partir de la admisibilidad se deduce que si tenemos el coste $h^*(ni)$ y el coste $h^*(nj)$ y el coste óptimo para ir de ni a nj , o sea $K(ni,nj)$, se ha de cumplir la desigualdad triangular:

$$h^*(ni) \leq h^*(nj) + K(ni,nj)$$

La **consistencia** exige pedir lo mismo al estimador h :

$$h(ni) - h(nj) \leq K(ni,nj)$$

Es decir, que la diferencia de las estimaciones sea menor o igual que la distancia óptima entre nodos. Que esta propiedad se cumpla quiere decir que h es un estimador uniforme de h^* . Es decir, la estimación de la distancia que calcula h disminuye de manera uniforme.

Si h es consistente además se cumple que el valor de $g(n)$ para cualquier nodo es $g^*(n)$, por lo tanto, una vez hemos llegado a un nodo sabemos que hemos llegado a él por el camino óptimo desde el nodo inicial. Si esto es así, no es posible que encontrar el mismo nodo por un camino alternativo con un coste menor y esto hace que el tratamiento de nodos cerrados duplicados sea innecesario, lo que ahorra tener que almacenarlos.

Suele ser un resultado habitual que las funciones heurísticas admisibles sean consistentes y, de hecho, hay que esforzarse bastante para conseguir que no sea así.

HEURÍSTICO MÁS INFORMADO

Otra propiedad interesante es la que permite comparar heurísticas entre sí, y permite saber cuál de ellos hará que A^* necesite explorar más nodos para encontrar una solución óptima:

Se dice que el heurístico h_1 es **más informado** que el heurístico h_2 si se cumple la propiedad:

$$\forall n \ (0 \leq h_2(n) < h_1(n) \leq h^*(n))$$

En particular, poder verificar esta propiedad implica que los dos heurísticos son admisibles. Si un heurístico es más informado que otro, al hacer uso de él en el algoritmo A^* se expandirán menos nodos durante la búsqueda, ya que su comportamiento será más en profundidad y habrá ciertos nodos que no se explorarán.

Esto podría hacer pensar que siempre hay que escoger el heurístico más informado, pero se ha de tener en cuenta que la función heurística también tiene un tiempo de cálculo que afecta al tiempo de cada iteración, y que suele ser más costoso cuanto más informado sea el heurístico. Podemos imaginar por ejemplo que, si tenemos un heurístico que necesita 10 iteraciones para llegar a la solución, pero tiene un coste de 100 unidades de tiempo por iteración, tardará más en llegar a la solución que otro heurístico que necesite 100 iteraciones pero que solo necesite una unidad de tiempo por iteración. Lo que lleva a buscar un equilibrio entre el coste del cálculo de la función heurística y el número de expansiones que ahorramos al utilizarla. En este sentido, es posible que una función heurística peor acabe dando mejor resultado.

VARIANTE CON MENOS MEMORIA: EL ALGORITMO IDA*

Como hemos visto, el algoritmo A^* tiene limitaciones de espacio por poder acabar degenerando en una búsqueda en anchura si la función heurística no es demasiado buena. Además, al igual que ocurría en la búsqueda ciega, si la solución del problema está a mucha profundidad o el tamaño del espacio de búsqueda es muy grande, podemos encontrarnos con necesidades de espacio prohibitivas. Esto lleva a buscar algoritmos alternativos que tengan menores necesidades de espacio.

La primera solución viene de la mano del algoritmo en profundidad iterativa de búsqueda ciega, pero añadiéndole el uso de la función f para controlar la profundidad a la que llegamos en cada iteración. En concreto, se realiza la búsqueda imponiendo un límite al coste del camino que se quiere hallar (en la primera iteración, la f del nodo raíz), explorando en profundidad todos los nodos con f igual o inferior a ese límite y reiniciando la búsqueda con un coste mayor si no encontramos la solución en la iteración actual:

Algoritmo: IDA* (límite entero)

```
prof ← f(Estado inicial)
Actual ← Estado inicial
mientras Actual no es_final? y prof
    Inicializa Est_abiertos
    Insertar Estado_inicial Est_abiertos
    Actual ← Primero Est_abiertos
    mientras Actual no es_final? y Est_abiertos no vacía? hacer
        Quitar_primeros Est_abiertos
        Insertar Actual Est_cerrados
        Hijos ← generar_sucesores_acotados (Actual, prof)
        Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
        Insertar Hijos Est_abiertos
        Actual ← Primero Est_abiertos
fin
```

```
prof ← prof+1
fin
```

La función `generar_sucesores` solo retorna los nodos con un coste inferior o igual al de la iteración actual.

Este algoritmo, al necesitar solo una cantidad de espacio lineal, permite hallar soluciones a más profundidad, aunque hay que pagar el precio de reexpandir nodos ya visitados. Este precio extra dependerá de la conectividad del grafo asociado al espacio de estados, si existen muchos ciclos éste puede ser relativamente alto ya que a cada iteración la efectividad de la exploración real se reduce con el número de nodos repetidos que aparecen.