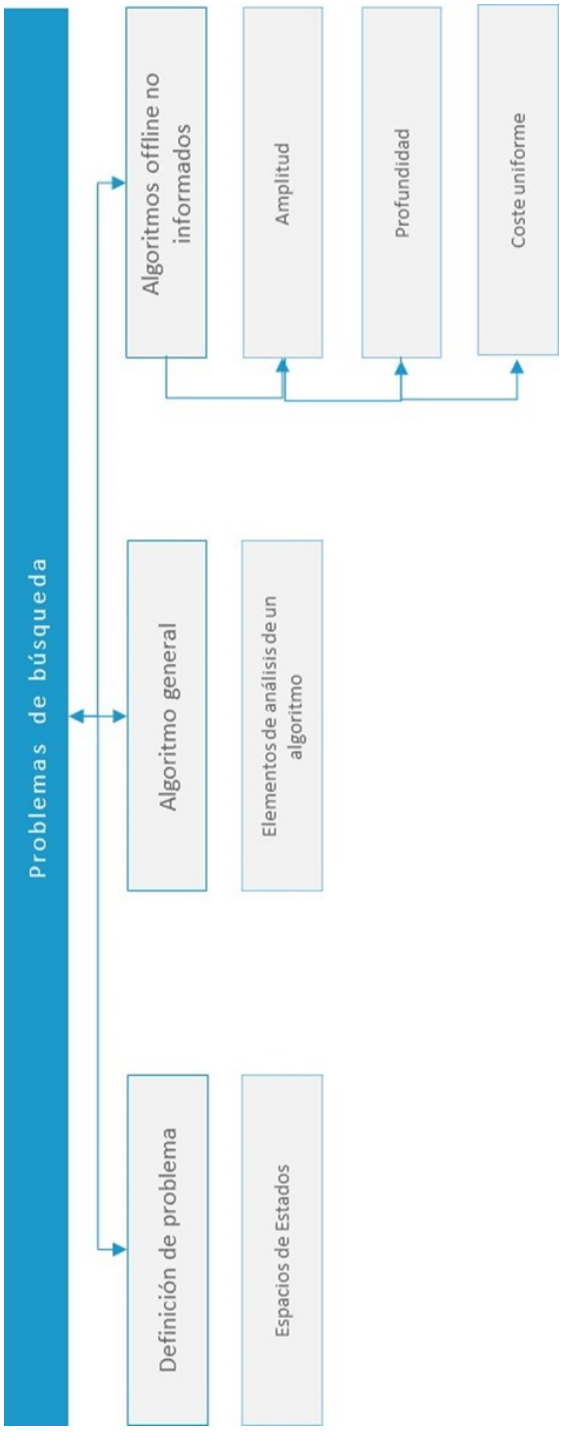
Razonamiento y Planificación Automática

Tema 4. Búsqueda no informada

Índice

Esquema



4.1. ¿Cómo estudiar este tema?

En este tema presentaremos la descripción general de los problemas de búsqueda en un espacio de estados no estructurado. Este tipo de problemas representa gran parte de los mecanismos de búsqueda autónoma de los agentes inteligentes.

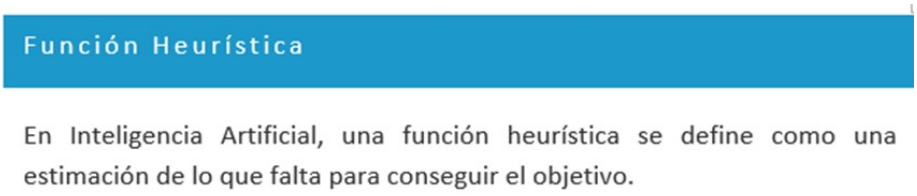
Los planificadores que veremos en los temas finales se apoyan en los conceptos de búsqueda que aparecen en los siguientes temas. Los agentes inteligentes deben enfrentarse muchas veces a problemas que les obligan explorar el entorno al que pueden acceder e intentar encontrar cuáles son las acciones que les permiten alcanzar sus objetivos y metas.

Principalmente este tema, así como alguno de los siguientes, se apoyan en el texto Inteligencia artificial: un enfoque moderno, de (Russell, 2004). Por lo tanto, se hace un requisito necesario durante el desarrollo de la asignatura leer este texto.

Puedes encontrar mucha información (en inglés) en la web de los autores: http://aima.cs.berkeley.edu/

4.2. Descripción general de un problema de búsqueda

Los problemas de búsqueda se pueden catalogar de distintas maneras. En este tema nosotros vamos a explicar dos categorías o divisiones. La primera división toma en cuenta una particularidad del algoritmo de búsqueda, si utiliza una función heurística o no.



En caso de no utilizar heurística, lo llamaremos una búsqueda no informada. Por el contrario, si el algoritmo utiliza una heurística, se considera una búsqueda informada.

Vamos a explicar un poco en más detalle estos dos términos:

- Búsqueda no informada: Decimos que una búsqueda es no informada cuando no emplea ningún tipo de heurística (el termino heurística lo explicaremos en más adelante con más detalle). Es decir, no tienen ningún modo de poder guiar la búsqueda, siempre se evalúa el siguiente estado sin conocer a priori si este es mejor o peor que el anterior.

- Búsqueda informada: representa aquellos algoritmos que emplean una función heurística para guiar la búsqueda y de esta manera llegar a soluciones optimas del problema.

La segunda división está más relacionada con el tipo de agente. Si es reactivo o deliberativo.

Si es deliberativo el agente realiza una búsqueda más offline, en la que puede emplear todo el tiempo necesario para encontrar un plan solución. Por el contrario, si es reactivo el agente realiza una búsqueda más online, en la que cuenta con un tiempo limitado para encontrar una solución parcial.

- Búsqueda *offline:* representa a aquellos agentes que realizan un proceso de búsqueda de la secuencia de acciones que deben desarrollar desde el principio (el estado actual en el que se encuentran) hasta el estado final (objetivo o meta que desean alcanzar) y, una vez realizado el proceso de búsqueda, empiezan a ejecutar el plan para conseguir el estado final del problema.

- Búsqueda online: engloba a aquellos agentes que realizan una búsqueda a «corto» plazo, que no llega necesariamente a encontrar la meta, pero nos pone en el camino para alcanzarla. Al contrario que la búsqueda offline, empieza a ejecutar acciones durante el proceso de búsqueda. Normalmente, se emplean algoritmos que realizan una búsqueda local.

Antes de empezar a discutir estos problemas y sus agentes asociados, debemos definir el problema en general al que se enfrentan los agentes basados en búsquedas.

¿Qué son los agentes basados en búsquedas?

Son aquellos que:

- Mantienen un **modelo simbólico** del entorno. Modelo que representa solo aquella parte de la información del entorno que resulta relevante para el problema en cuestión, definiendo aquellos parámetros que permiten diferenciar un estado de otro del entorno.

- Desean **modificar el estado del entorno** de acuerdo con sus objetivos. Es decir, aplicar acciones que permitan modificar el entorno, de tal modo que se acabe alcanzando un estado meta de los que satisfacen los objetivos del agente.

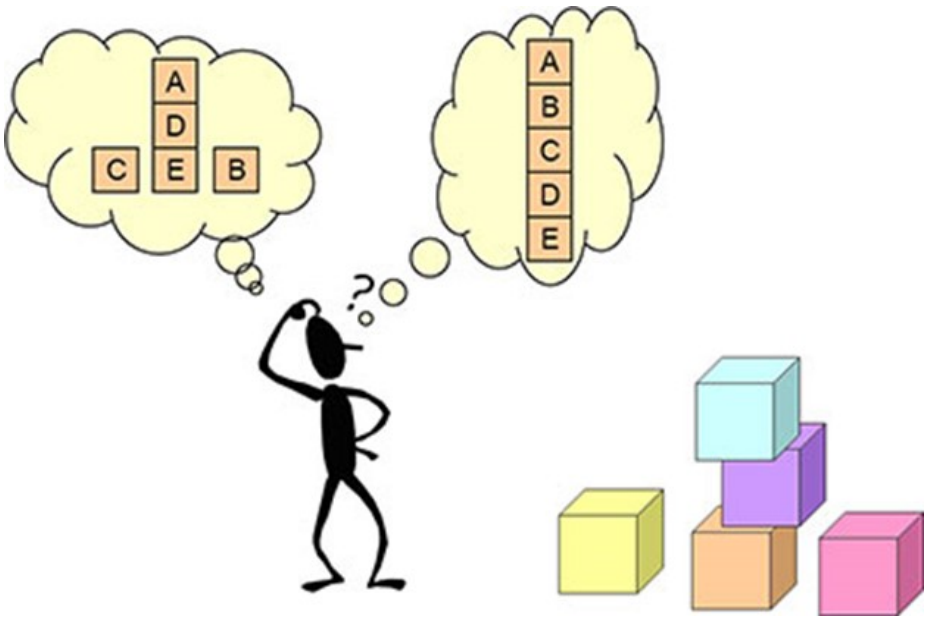


Figura 1. Dominio del mundo de bloques. El agente desea modificar el entorno para alcanzar un estado determinado (meta).

Por ejemplo, la figura 1, representa un agente que desea modificar un entorno del mundo de bloques. El entorno tiene 5 bloques, diferenciados por colores. El agente relaciona cada color con una letra, C=Amarillo, E=Naranja, B=rosa, D=Violeta, y A=Azul marino. Así, entonces en el entorno los bloques C, E y B se encuentran en la mesa. El bloque D se encuentra sobre el bloque E. Y el bloque A se encuentra sobre el bloque D. El objetivo del agente es colocar todos los bloques uno encima de otro. Más exactamente, el agente desea colocar el bloque A sobre el B, el B sobre el C, el C sobre el D, y el D sobre el E.

Ideas clave

Para modificar el entorno de acuerdo con sus objetivos, los agentes de búsqueda **anticipan** los efectos que tendrían sus acciones sobre el mundo (por medio de su modelo del entorno), generando **planes de actuación** a través de una secuencia de acciones que los llevan desde su estado actual hasta el objetivo buscado, tal como se muestra en la figura 2. Esto lo realizan por medio de un proceso de búsqueda.

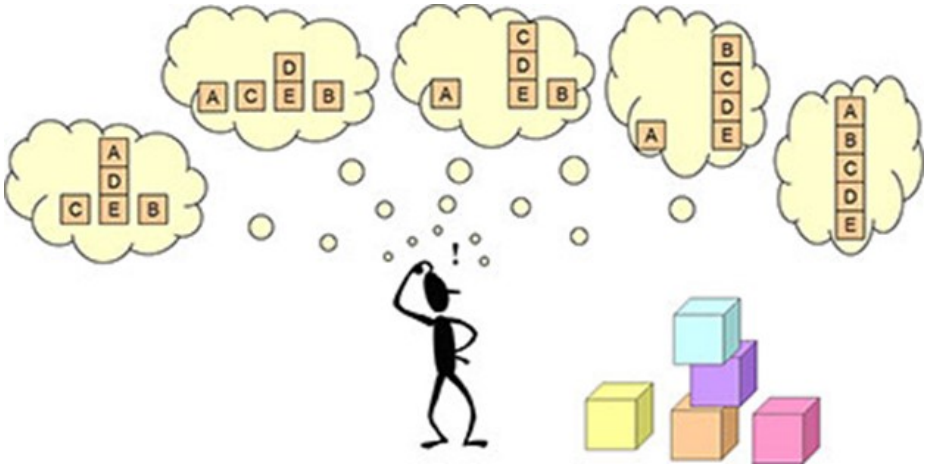


Figura 2. Dominio del mundo de bloques. Una búsqueda es la obtención de un plan de acciones para alcanzar la meta.

Si son agentes deliberativos, esta tarea la realizan antes de ejecutar las acciones del plan en el entorno real. Es decir, una vez encuentran el plan solución comienza la ejecución de este.

Si son agentes reactivos, esta tarea la van realizando a la vez que van ejecutando las acciones, corriendo el riesgo de equivocarse y tener que deshacer acciones que en muchos casos pueden resultar costosas.

Cualquiera de los dos tipos de agentes puede usar búsquedas no informadas e informadas.

Mecanismo para resolver estos problemas

Para resolver este tipo de problemas tenemos varios mecanismos que van desde aquellos que no permiten al agente ser autónomo de ninguna manera a aquellos que le permiten realizar procesos racionales de toma de decisiones.

Vamos a explicar algunos de estos mecanismos teniendo en cuenta el dominio de las torres de Hanoi mostrado en la figura 3.

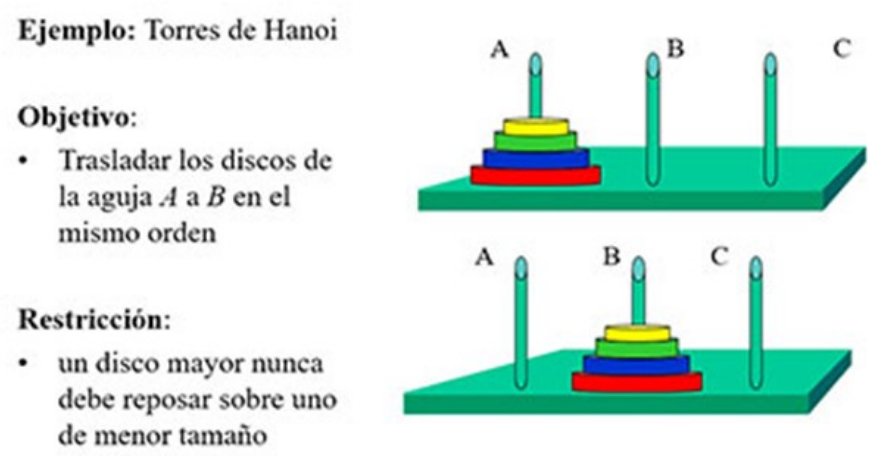


Figura 3. Domino de las Torres de Hanoi.

En este problema, se encuentran tres agujas A, B, y C. En la aguja A se encuentran 4 discos (1, 2, 3 y 4) de diferentes tamaños cada uno. Ubicados de mayor a menor, siendo 1 el de menor tamaño (el disco de color amarillo en la Figura 4). El objetivo es trasladar todos los discos a la aguja B. Los discos deben quedar en la aguja B en el mismo orden que estaban en la aguja A. Como restricción adicional se tiene que un disco mayor nunca puede estar sobre un disco de menor tamaño.

Tablas de actuación

En estos mecanismos, para cada situación hay una entrada en una tabla de actuación específica para el problema; dicha tabla presenta la secuencia de acciones

completa para llegar desde el estado inicial a uno de los estados finales.

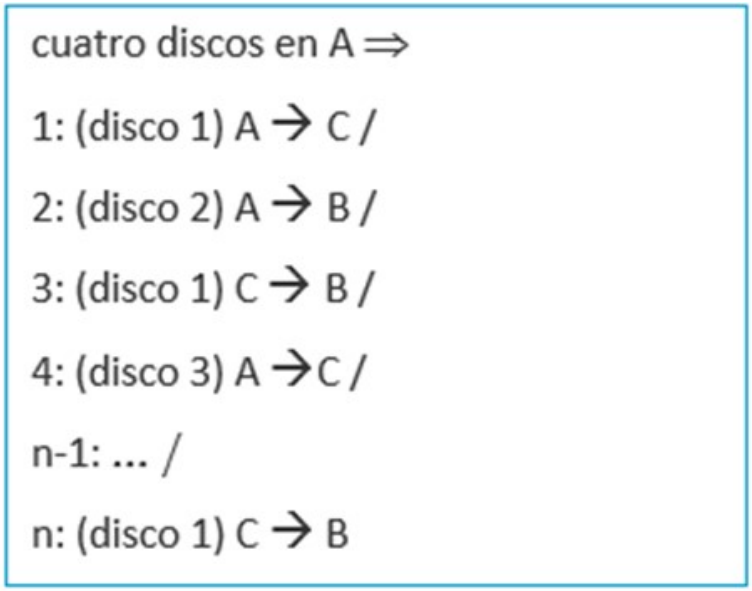


Figura 4. Ejemplo de fragmento de tabla para el dominio de las torres de Hanoi con cuatro discos.

Figura 4, muestra un ejemplo de tablas de actuación para el dominio de las torres de Hanoi de la figura 3. Por ejemplo, la línea 1 indica que lo primero que debemos hacer es mover el disco 1 desde la aguja A a la aguja C.

Se puede mejorar la flexibilidad del agente por medio de técnicas que le permitan **aprender** nuevas entradas, pero tiene un grave problema de escalabilidad dado que rápidamente tiene problemas de memoria.

Algoritmos específicos del dominio

En esta técnica de resolución, el diseñador del agente conoce un método para resolver problemas de un dominio concreto y codifica este método en un algoritmo particular para el dominio. Generamos, en resumen, un código específico que resuelve cualquier problema concreto que le planteamos de un dominio específico.

Se puede intentar mejorar su flexibilidad por medio de crear un código que admita parámetros que configuren el problema y otro código que los resuelva de modo general para cualquier valor admitido como parámetro.

El principal problema es que el diseñador de la solución debe prever todos los escenarios posibles. En entornos reales, suele ser demasiado complejo anticipar todas las posibilidades. Además, de que solo funciona para un dominio en particular.

En la figura 5, se puede ver un algoritmo específico para solucionar cualquier problema de las torres de Hanoi.

```
PROCEDURE MoverDiscos(n:integer;
                      origen,destino,auxiliar:char);
{ Pre: n > 0
  Post: output = [movimientos para pasar n
                  discos de la aguja origen
                   a la aguja destino] }
BEGIN
  IF n = 0 THEN  {Caso base}
    writeln
  ELSE BEGIN     {Caso recurrente}
    MoverDiscos(n-1,origen,auxiliar,destino);
    write('Pasar disco',n,'de',origen,'a',destino);
    MoverDiscos(n-1,auxiliar,destino,origen)
  END; {fin ELSE}
END; {fin MoverDiscos}
```

Figura 5. Dominio de las Torres de Hanoi. Ejemplo de código que resuelve problemas de modo recursivo.

Como podemos ver, este algoritmo permite resolver problemas de las torres de Hanoi con N discos. Es específico del problema porque solo puede ser usado para este dominio. El mismo algoritmo no es válido para el dominio del mundo de bloques, por ejemplo.

Métodos independientes del dominio

Son aquellos métodos que emplean un modelo simbólico del dominio y problema. Por ejemplo, para el caso de las torres de Hanoi, definiríamos de modo general el problema tal como se muestra en la siguiente figura 6.

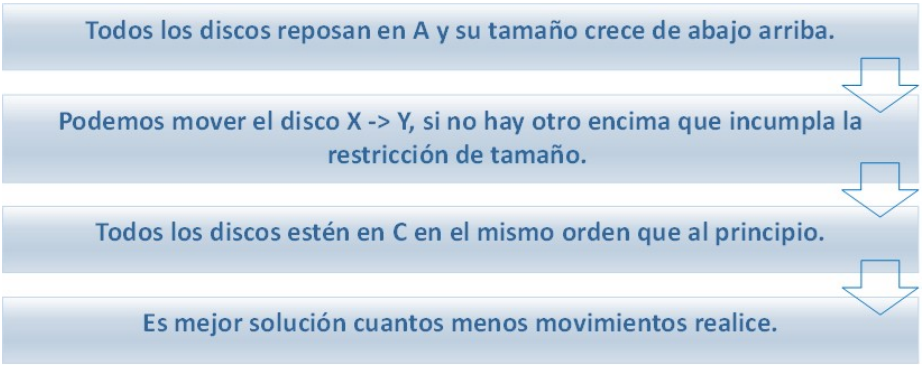


Figura 6. Dominio de las Torres de Hanoi. Definición general para un problema.

Para resolver los problemas, este método emplea un **algoritmo de búsqueda genérico**, representando el dominio y el problema mediante el modelo simbólico. Permite una mayor flexibilidad, ya que no necesitamos conocer la solución previamente y es fácil añadir nuevas características al problema.

Problemas de búsqueda en el espacio de estados

En estos problemas, nos encontramos con que el entorno se representa por medio de estados, que se deben diferenciar entre sí de modo unívoco, pero que no presentan características accesibles que los permitan diferenciar; es decir, que son distintos, pero para el agente solo son «etiquetas distintas» que representan estados distintos.

Para este tipo de problemas tenemos tres elementos que los definen:

- Espacio de estados: modelo del mundo representado por un grafo, en el cual tenemos un conjunto de elementos que representan componentes del mundo, que se traducen en elementos del modelo simbólico que simbolizaremos de una manera determinada en el grafo.

Figura 7. Espacio de estados.

- Problema de búsqueda: por medio de un mecanismo independiente del problema, exploramos el **espacio de estados** aplicando la **actitud del agente**, que representa el componente de racionalidad en el proceso de exploración.

Figura 8. Problema de búsqueda.

- Objetivo: queriendo encontrar el plan más eficiente que lleve del estado inicial a un estado meta.

Por tanto, un problema de este tipo trata de encontrar el mejor camino dentro de un grafo dirigido como el presentado en la figura 9. Pero, por desgracia, no conocemos el grafo. Si lo conociésemos, realizaríamos una búsqueda de camino mínimo en grafos como, por ejemplo, Dijkstra (Rosettacode, 2020).

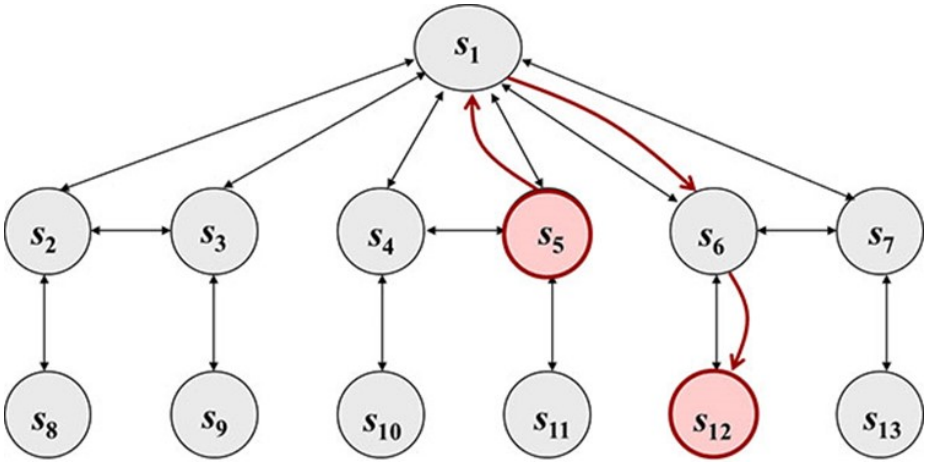


Figura 9. Grafo que representaría un espacio de estados de un problema.

Para este tipo de problemas disponemos de una serie de conocimientos *a priori* en el agente que le van a permitir realizar una estrategia de búsqueda sobre el problema, aun desconociendo el modelo completo del mismo. Así, tendremos una representación del conocimiento del problema de **búsqueda implícita**.

Conocimiento a priori del agente	
s_0	Estado *inicial*
expandir(s): $\{s_1, \ldots, s_n\}$	Conjunto de estados sucesores del estado *s*
meta(s): verdad | falso	Función de evaluación de *s* como estado meta
c(si, sj): v, $v \in \mathbb{R}$	Coste de aplicar un operador que lleva de si a sj
$c(s_1\, s_2 \ldots s_n) = \sum_{k=1}^{n-1} c(s_k, s_{k+1})$	Coste del plan completo

Tabla 1. Conocimiento *a priori* del agente.

Con esta información *a priori*, emplearemos una **estrategia** basada en el **método de búsqueda**, ver figura 10, que irá explorando el espacio de estados, **expandiendo** a cada paso un estado y creando de modo progresivo un **árbol de búsqueda** tal como el mostrado en la figura 11.

Figura 10. Método de búsqueda a partir de la elección de una hoja que representa un estado.

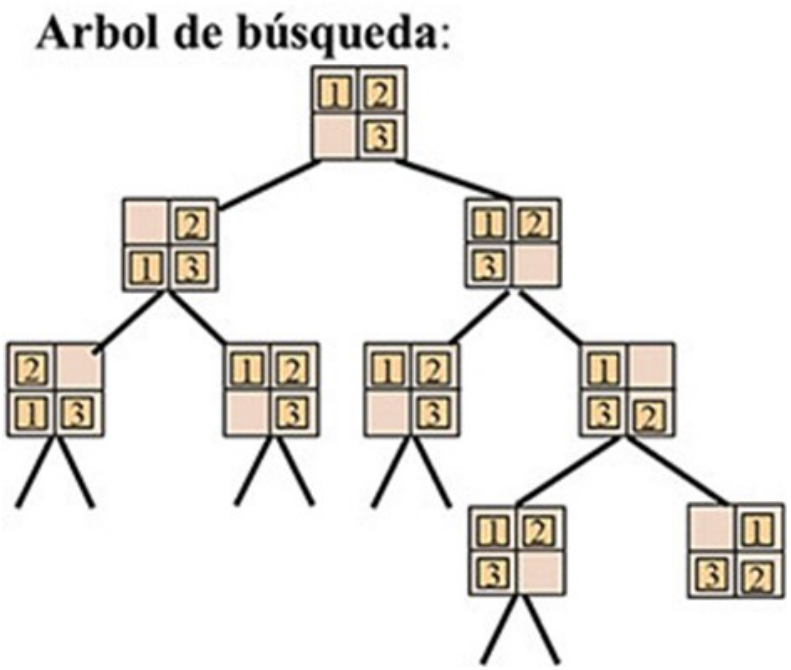


Figura 11. Árbol de búsqueda.

Algoritmo general de búsqueda

Podemos definir un algoritmo general de búsqueda tal como está en la figura 12:

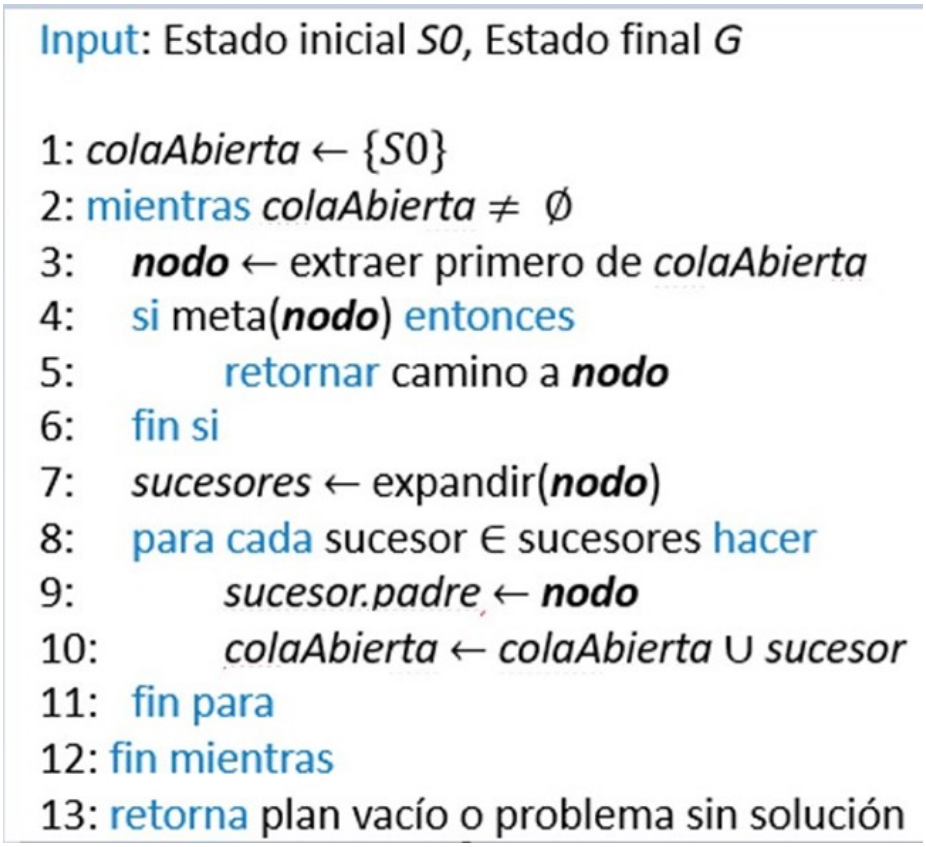


Figura 12. Algoritmo general de búsqueda.

Donde el árbol se representa con base a un registro **del tipo nodo**, en su representación más simple de un nodo enlazado a su antecesor (padre) por medio de una referencia (línea 9). En este nodo almacenaremos el estado que se alcanza en este instante de la exploración.

Existe una lista de nodos «abierta» (línea 1) con las hojas actuales del árbol, es decir, aquellos estados y caminos que hemos expandido para explorar.

Si la lista está vacía, nos encontraremos con un problema sin solución. Sino está vacía extraeremos el primer elemento de la lista de hojas abiertas.

Si el nodo es meta, entonces retornamos el camino haciendo un *backtracking* sobre los nodos padres del nodo meta encontrado (línea 4).

Por último, añadiremos los estados nuevos obtenidos de la expansión en la lista de abierto (línea 10).

En las siguientes secciones mostraremos el funcionamiento de los distintos algoritmos de búsqueda no informados, que son empleados como base en muchos problemas de agentes inteligentes.

En todos los mecanismos de búsqueda tenemos presente un posible problema que puede aparecer, que son los **estados repetidos**. Para resolver este problema, que puede causar en algunos casos errores graves como entrar en bucles infinitos, tenemos distintas estrategias:

- Ignorarlo: por extraño que parezca, algunos algoritmos no tienen problemas con esta solución debido a su propio orden de exploración.
- Evitar ciclos simples: evitando añadir el padre de un nodo al conjunto de sucesores.
- Evitar ciclos generales: de tal modo que ningún antecesor de un nodo se añada al conjunto de sucesores.
- Evitar todos los estados repetidos: no permitiendo añadir ningún nodo existente en el árbol al conjunto de sucesores.

Estas estrategias deben tener en cuenta el coste que conlleva tanto explorar de más como buscar elementos repetidos para explorar menos.

Ideas clave

En los siguientes temas y en las siguientes secciones presentaremos distintos algoritmos. Para todos ellos emplearemos un mecanismo de clasificación basado en los siguientes conceptos y características:

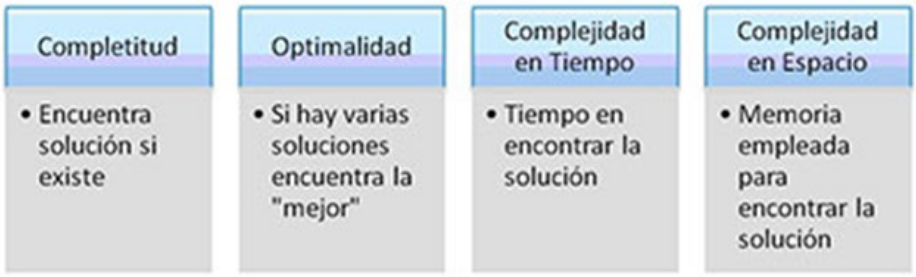


Figura 13. Características de un algoritmo.

4.3. Búsqueda en amplitud

La búsqueda en amplitud (BFS, por sus siglas en inglés) es una estrategia que genera el árbol de búsqueda por niveles de profundidad, expandiendo todos los nodos de nivel i antes de expandir los nodos de nivel i+1.

Considera, en primer lugar, todos aquellos estados que se encuentran en caminos de longitud 1 (es decir, aquellos caminos que solo requieren una acción), luego los de longitud 2 (caminos que solo requieren dos acciones), etc. De este modo, se encuentra aquel estado meta que esté a menor profundidad.

La figura 14 muestra un ejemplo de cómo se va generando el árbol con este algoritmo.

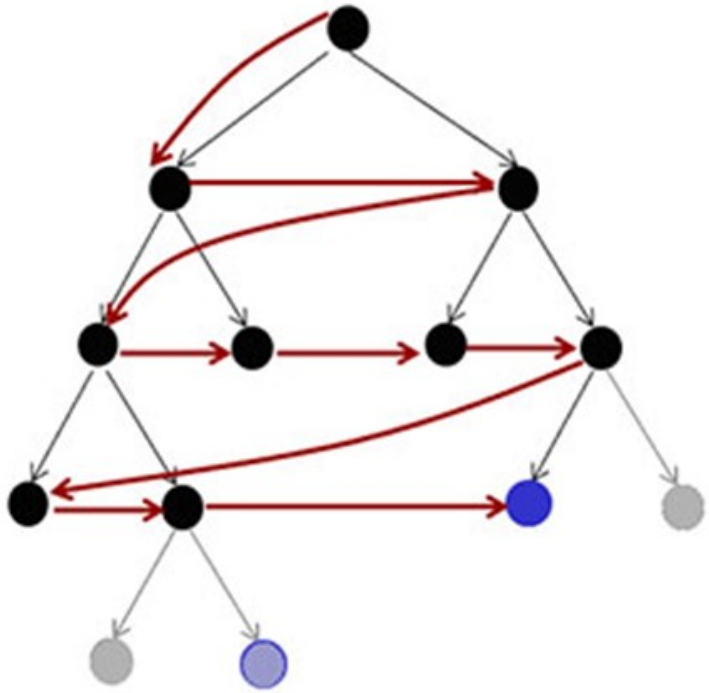


Figura 14. Esquema de búsqueda en amplitud.

El algoritmo desarrollaría un mecanismo de búsqueda que, por ejemplo, en el problema del puzzle-8, derivaría en el árbol de búsqueda mostrado en la figura 15.

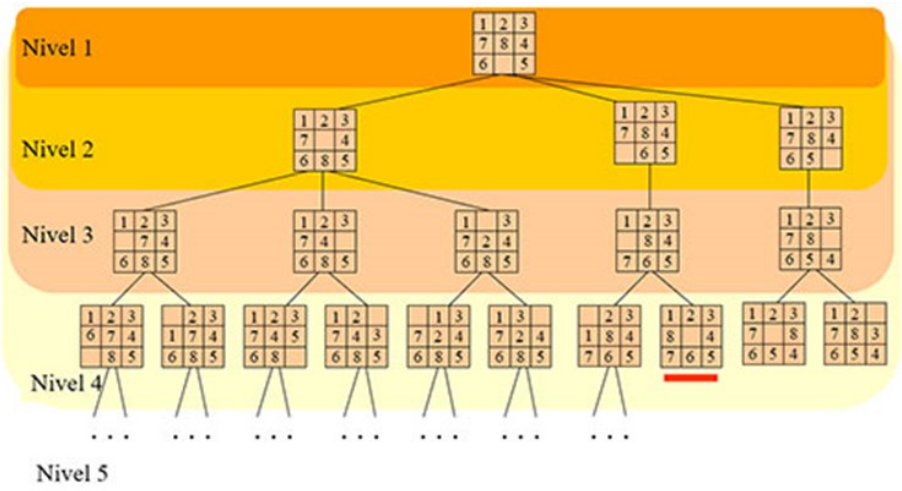


Figura 15. Árbol de búsqueda en amplitud para el puzzle-8.

Puzzle-8: es un popular juego que consiste en una matriz de 3x3, en la que se encuentran unas fichas enumeradas del 1 al 8. Adicionalmente, existe una pieza vacía o en blanco. El juego consiste en dado una configuración inicial llevar las piezas a una configuración final. En cada turno las únicas piezas que se pueden mover son las piezas adyacentes a la pieza vacía.

El algoritmo de búsqueda en amplitud lo podemos extraer del algoritmo de búsqueda general anteriormente expuesto en la figura 15. Matizando las siguientes cuestiones:

- Para añadir nuevos sucesores, lo haremos al final de la lista abierta.
- Por su parte, la lista abierta funciona como cola (insertando al final y recuperando al inicio), lo que conlleva que siempre se expandan primero aquellos nodos más antiguos (es decir, los menos profundos).
- Adicionalmente, controlaremos los nodos que se han visitado previamente.

Este algoritmo es, por tanto, **completo y óptimo**, pero presenta una complejidad en tiempo y espacio muy deficiente, ya que depende de modo proporcional al nivel de profundidad d de la solución y, por tanto, al número de nodos expandidos o factor de ramificación b.

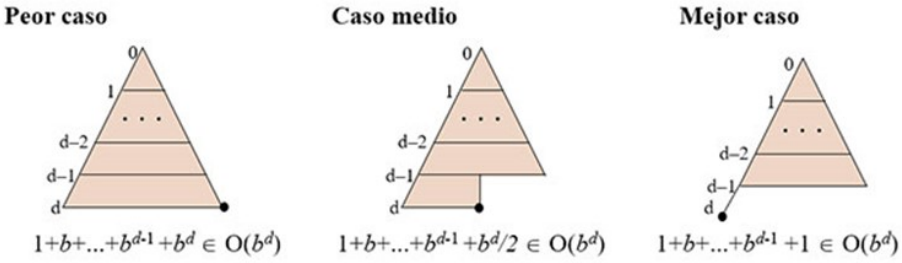


Figura 16. Complejidad espacial y temporal de un grafo sin eliminación de nodos duplicados. El factor de ramificación está en b y la solución está en una profundidad d.

En la figura 16 se puede ver un análisis más exhaustivo de la complejidad aproximada de este algoritmo para un problema general.

Ejemplo: recursos requeridos por la búsqueda en amplitud en el *peor caso*

- factor de ramificación efectivo: 10
- tiempo: 1.000.000 nodos/segundo
- memoria: 1.000 bytes/nodo

d	nodos	tiempo	memoria
2	110	0,11 ms	107 KB
4	11.110	11 ms	10,6 MB
6	10^6	1,1 s	1 GB
8	10^8	2 min	103 GB
10	10^{10}	3 horas	10 TB
12	10^{12}	13 días	1.000 TB
14	10^{14}	3,5 años	99 PB
16	10^{16}	350 años	10.000 PB

Figura 17. Análisis de la complejidad de un algoritmo de búsqueda en amplitud. Fuente: (Russell, 2004).

4.4. Búsqueda en profundidad

La búsqueda en profundidad (DFS, por sus siglas en inglés) es otra estrategia de búsqueda no informada (sin información adicional). En ella, al contrario de la búsqueda en amplitud, se intenta desarrollar un camino de longitud indeterminada, en el cual intentamos alcanzar metas profundas (aquellas que tienen un camino largo para alcanzarlas) desarrollando las menores ramificaciones posibles.

En general, es un algoritmo que funcionará bien mezclado con otras informaciones adicionales, pero que puede resolver problemas de la misma manera que un algoritmo en amplitud.

La figura 18 muestra un ejemplo de cómo se va generando el árbol con este algoritmo.

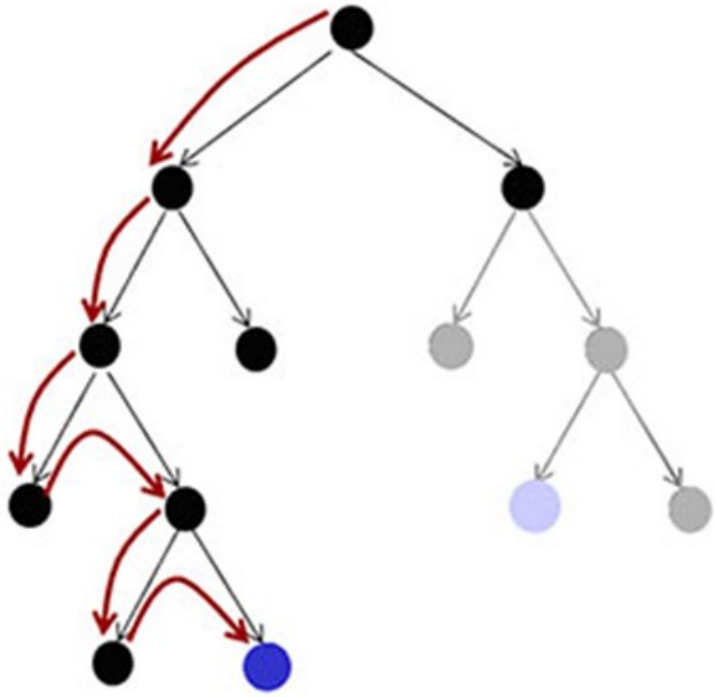


Figura 18. Esquema de búsqueda en profundidad.

En esta estrategia, se expande el árbol de «izquierda a derecha», por lo tanto, aquellos nodos más profundos se expanden primero. Si se llega a un nodo sin sucesores, se retrocede y se expande el siguiente nodo más profundo.

Como resultado, el método va explorando un «camino actual» y no siempre se encuentra el nodo de profundidad mínima. En este caso, el algoritmo general se ve adaptado teniendo en cuenta las siguientes consideraciones:

- Los nuevos sucesores se añaden al inicio de la lista abierta,

Ideas clave

- La lista abierta funcionará como una pila (insertando al principio y extrayendo también del principio) y siempre extraeremos el nodo más profundo. Al guardar todos los sucesores de un nodo expandido en abierta, se permite la «vuelta atrás» o *backtracking*.
- Solo procesaremos un nodo de la pila si este no ha sido visitado aún.

El análisis de este algoritmo nos muestra que es **completo** (si y solo si se garantiza la eliminación de los estados repetidos dentro de una misma rama), pero **no es óptimo** (para operadores de coste uno), dado que no garantiza que siempre se encuentre aquella solución que está a la menor profundidad.

4.5. Búsqueda de coste uniforme

En los casos anteriores de las búsquedas en amplitud y profundidad se empleaba una asunción de que el coste de aplicación de cualquier acción (por tanto, el coste de elegir una rama) era siempre igual a 1, por lo que el coste total del camino era el número de niveles en el que se encontraba un determinado nodo. Pero ¿qué sucede cuando el coste de transitar de un nodo a otro no es igual para todas las acciones dentro del entorno?

Por ejemplo, en el caso de la figura 19, que representa el problema de encontrar la ruta más corta en un grafo donde cada nodo es una ciudad y las aristas entre las ciudades contienen un número que representa el coste del operador, distancia por carretera de una ciudad a otra.

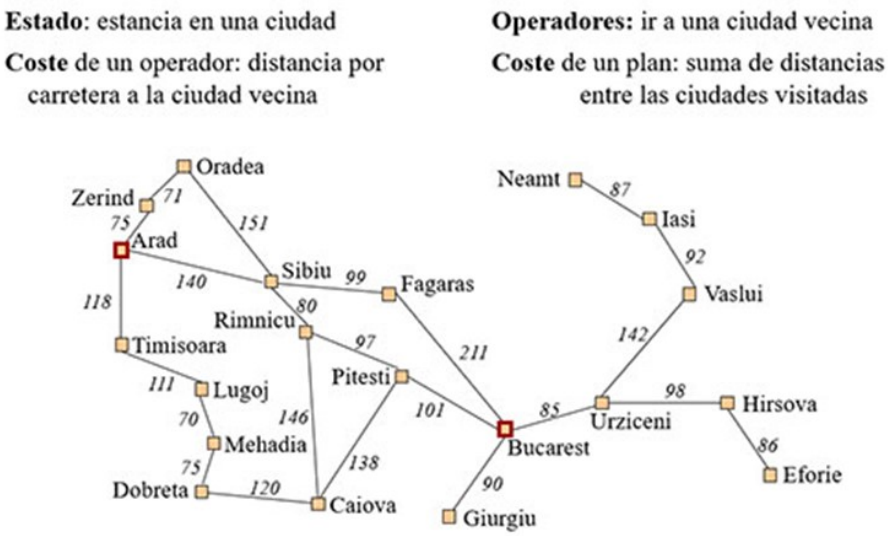


Figura 19. Grafo esquemático de carreteras.

En este caso, como se puede observar en la figura 20, la exploración de caminos empleando el algoritmo de búsqueda en amplitud que garantizaba la solución óptima falla. El algoritmo en amplitud ofrece la solución de color rojo, A - S - F – B. La solución óptima en este caso, es la que se encuentra en color verde, A – S – R – P – B.

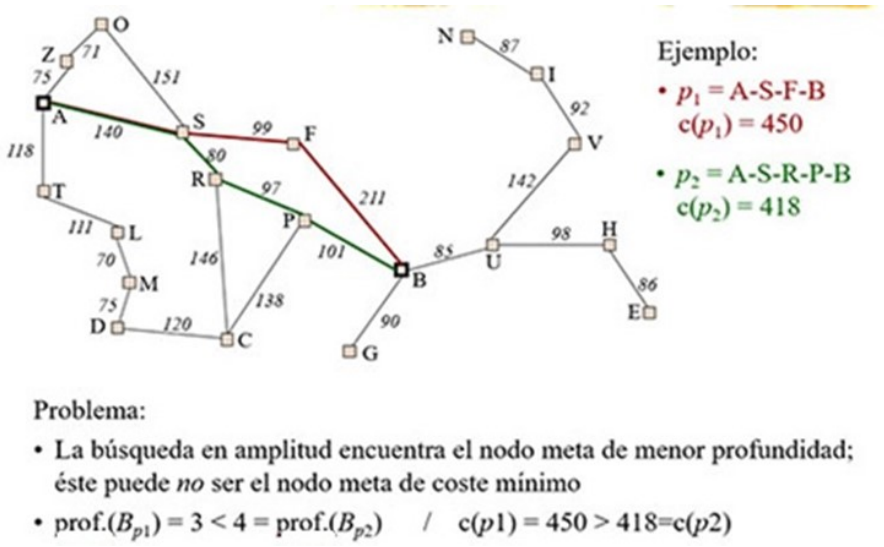


Figura 20. La búsqueda en amplitud, que asume coste 1, falla para encontrar el camino óptimo.

Para resolver este tipo de escenario donde el coste no es igual para todas las acciones (pero sí positivo en todos los casos), tenemos el algoritmo de búsqueda de coste uniforme (UCS por sus siglas en inglés).

Empleando el mismo algoritmo general de búsqueda, aplicamos la idea de dirigir la búsqueda por el coste de los operadores. Supondremos que existe una función de utilidad f(n) = g(n) que permite calcular el coste real para llegar del nodo inicial al nodo n. En cada iteración del algoritmo expandiremos primero el nodo de menor coste f.

Por simplicidad, en el resto de la explicación de este algoritmo, nos referiremos a la función de utilidad indistintamente como f o g.

Las modificaciones que se realizan del algoritmo general de búsqueda son:

- Almacenaremos cada nodo por prioridad con base a su valor de *g*, por lo tanto, la inserción de nuevos nodos en la lista abierta se ordenará de modo ascendente según su valor *g*.
- Lo anterior hace que la lista abierta este definida como una cola de prioridad ordenada por el valor de g.
- Solo agregamos un nodo a la lista abierta si este no se encuentra en la misma.
- En caso de encontrarse, reemplazamos el nodo si y solo si el coste f es menor.

La figura 21 muestra el resultado de aplicar el algoritmo UCS al problema planteado en la figura 19. En cada nodo se observa el valor real de g.

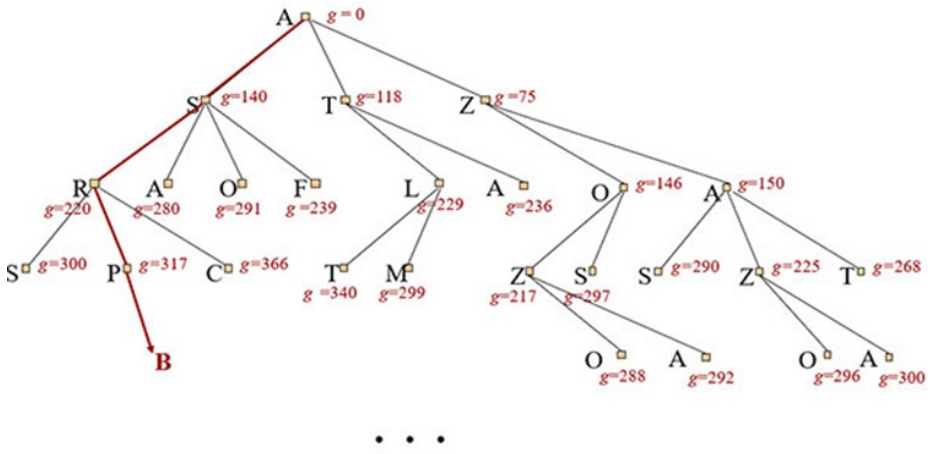


Figura 21. Árbol de exploración de una búsqueda de coste uniforme.

Este algoritmo es **completo** y óptimo al tener todos los costes con valores enteros positivos y, por tanto, la sucesión de valores de *g* no está acotada y siempre se expande de acuerdo con el orden de inserción que se basa en esta misma función.

4.6. Referencias bibliográficas

Rosettacode. (14 de 03 de 2020). Obtenido de Dijkstra's algorithm: http://rosettacode.org/wiki/Dijkstra%27s_algorithm

Russell, S. y. (2004). *Inteligencia Artificial: Un Enfoque Moderno*. Madrid: Pearson Educación.

Resolución de problemas

Russell, S. y Norvig, P. (2004). Resolver problemas mediante búsqueda. En. Russell, S. y Norvig, P. (Eds.), *Inteligencia artificial. Un enfoque moderno* (pp. 67-107). Madrid: Pearson Educación.

Este capítulo cubre los aspectos que fundamentan todos los inconvenientes de la resolución de problemas por medio de búsqueda.

Problemas resueltos de inteligencia artificial

Billhardt, H., Fernández, A. y Ossowski, S. (2015). *Inteligencia artificial. Ejercicios resueltos.* Madrid: Editorial Universitaria Ramón Areces.

Se presenta una amplia colección de ejercicios resueltos de varios temas tratados en el curso. En este tema conviene repasar los ejercicios de búsquedas no informadas (capítulo 1).

OpenNERO

Accede a la página web a través del aula virtual o desde la siguiente dirección web: https://github.com/nnrg/opennero/wiki

OpenNERO es una plataforma de software de código abierto diseñada para investigación y educación en Inteligencia Artificial. El proyecto se basa en el juego *Neuro-Evolving Robotic Operatives (NERO),* desarrollado por estudiantes de posgrado y pregrado del Grupo de Investigación de Redes Neuronales y del Departamento de Ciencias de la Computación de la Universidad de Texas, en Austin.

En particular, OpenNERO se ha utilizado para implementar varias demostraciones y ejercicios para el libro de texto de Russell y Norvig: *Inteligencia Artificial: Un Enfoque Moderno*. Estas demostraciones y ejercicios ilustran métodos de IA como la búsqueda de fuerza bruta, búsqueda heurística, *scripting*, aprendizaje de refuerzo y computación evolutiva, y problemas de IA como correr laberintos, pasar la aspiradora y la batalla robótica. Los métodos y problemas se implementan en varios entornos diferentes (o *mods*).

Grupo de desarrolladores de OpenNero en GoogleGroups

Accede a la página web a través del aula virtual o desde la siguiente dirección web: https://groups.google.com/forum/#!forum/opennero

Grupo de Google en el que los desarrolladores opinan y en el que se pueden resolver las dudas que vayan surgiendo.

Test

1. ¿Qué es una búsqueda *offline*?

A. Aquella que se realiza antes de empezar a ejecutar cualquier acción sobre el entorno del agente.

B. La que se hace sin conexión a Internet.

C. En la que solo existen dos acciones en cada estado.

D Todas son correctas.

2. ¿Qué es una búsqueda *online*?

A. La que emplea buscadores de Internet para alcanzar el estado meta.

B. La que no tiene estados meta bien definidos.

C. La que realiza búsquedas en tiempos cortos, aunque no consigan la meta.

D. Todas son correctas.

3. ¿Qué es una acción en un modelo de búsqueda?

A. Es una operación que cambia el estado de alguna manera.

B. Es una variable del entorno.

C. Es una operación que el agente puede estudiar para ver si le permite conseguir el estado meta.

D. La A y la C son correctas.

4. Un estado para este tipo de problemas es:

A. Una colección de variables que presentan estructura.

B. Un «identificador» sin estructura, pero diferenciable entre sí.

C. Un objetivo que no se puede alcanzar.

D. Todas son correctas.

5. La búsqueda en amplitud es:

A. Completa, pero no óptima.

B. Completa y óptima.

C. Incompleta.

D. Poco compleja.

6. Una búsqueda es óptima…

A. Si encuentra la solución en poco tiempo.

B. Si encuentra la solución que tiene asociado un menor coste (por ejemplo, número de acciones).

C. Si no comente errores al duplicar estados.

D. La A y la C son correctas.

7. La búsqueda en profundidad es:

A. Completa para todos los casos.

B. Óptima para todos los casos.

C. Muy eficiente si la profundidad de la solución es elevada y se toma dicha rama.

D. Incompleta y óptima.

8. La búsqueda de coste uniforme:

A. Asume un coste positivo para todas las acciones.

B. Encuentra la solución óptima.

C. Es completa.

D. Todas son correctas.

9. Los estados repetidos en un árbol de búsqueda, ¿son un problema?

A. No, nunca, todos los algoritmos los filtran automáticamente.

B. Sí, pero se pueden filtrar en muchos casos los bucles que se formen.

C. Sí, cuando el algoritmo explora siempre en el mismo orden los nodos pendientes.

D. B y C son correctas.

10. Los algoritmos de búsqueda no informada:

A. Indican el orden de las acciones para que se pueda alcanzar un estado meta.

B. Expanden nodos que representan estados alcanzados por medio de las acciones del agente.

C. Tienen en cuenta el conocimiento *a priori* de los expertos que los programan.

D. A y B son correctas.