

Tema-3.-Busqueda-y-planificacion...



Anónimo



Sistemas Inteligentes



4º Grado en Ingeniería de Computadores



**Facultad de Informática
Universidad Complutense de Madrid**

Formamos
talento para un futuro
Sostenible



MÁSTER EN

**Big Data &
Business Analytics**

EOI Escuela de
organización
industrial

[saber más](#)

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

TEMA 3: BÚSQUEDA Y PLANIFICACIÓN.

Los agentes reactivos basan sus acciones en una aplicación directa desde los estados a las acciones.

Sin embargo, los agentes basados en objetivos pueden planificar sus acciones para conseguir sus objetivos a largo plazo.

Conceptos de computabilidad.

- Diseño de un algoritmo: hay que estudiar su funcionamiento y su complejidad (eficiencia).
 - Funcionamiento se puede comprobar con validación matemática si es posible o si no mediante traza.
 - La eficiencia: estimar el tiempo y espacio físico que necesitará el algoritmo para ejecutarse ante una entrada concreta.

Principio de invarianza.

Dadas dos implementaciones distintas del mismo algoritmo de tiempos $t1(n)$ y $t2(n)$ unidades de tiempo, siendo n el tamaño del problema, siempre existen constantes positivas a y b tales que:

$$t1(n) \leq a \cdot t2(n)$$

$$t2(n) \leq b \cdot t1(n)$$

Órdenes de complejidad.

$$O(c) < O(\log n) < O(n) < O(n^2) < O(n^3) < \dots < O(2^n) < O(n^n)$$

Un problema decidible es aquel que no puede ser resuelto por una computadora (existe una máquina de Turing que lo resuelve).

Hay problemas decidibles que son intratables porque son muy complejos de resolver.

Problemas no tratables.

No se pueden resolver algorítmicamente usando la fuerza bruta. Se necesita alguna forma de conseguir resolver el problema de forma eficiente. De esto se encarga la IA.

Muchos problemas de IA son de búsqueda. Un problema de búsqueda consiste en encontrar un conjunto de soluciones, aquella que es más eficiente. Si no podemos encontrar la mejor, hay que intentar conseguir una lo suficientemente buena.

Representación de un problema.

Se necesita codificar el problema que se pretende resolver para poder resolverlo por métodos de búsqueda. Para ello necesitamos algunas definiciones:

- Espacio de estados: conjunto de todos los estados posibles entre los que podemos encontrar una solución.
- Espacio de búsqueda: se puede representar en forma de árbol y sólo contiene los nodos generados por la búsqueda.
 - Los nodos del espacio de búsqueda pueden estar repetidos y puede haber ciclos.

Definición de problema.

Un problema puede definirse por cuatro componentes:

- Estado inicial en el que el agente comienza.
- Las acciones que puede tomar el agente en cada estado. Se suele definir una función "Sucesor" que devuelve el conjunto de pares (acción, sucesor) donde cada acción es una de las legales en el estado y cada sucesor es el estado al que llega en agente al aplicar la acción.
- Un test de objetivo para saber si se han cumplido los objetivos.
- Una función de coste del camino que asigna un costo numérico a cada camino (rendimiento del agente).

Ejemplos de definición de un problema

- 2 garrafas vacías con capacidades de 4 y 3 litros, respectivamente
- El objetivo es que la garrafa de 4 litros ha de contener exactamente 2 litros
- Tenemos un grifo para rellenarlas, un lavabo para vaciarlas, y posibilidad de trasvasar líquido de una garrafa a la otra, hasta que la 1ª se vacíe o la 2ª se llene



Solución

- Inicial: (0,0)
- Objetivo: (2,Y) => lo que sea en el segundo y 2 en el primero
- Acciones (operadores): Llenar-4L, Llenar-3L, Vaciar-4L, Vaciar-3L, Vaciar-4L-4L, Vaciar-4L-3L
- Coste de cada operación: número de litros desperdiciados o número de pasos que hacemos.

pierdo espacio



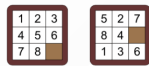
Necesito concentración

ali ali oohh
esto con 1 coin me
lo quito yo...

WUOLAH

WUOLAH 1

8 puzzle deslizante (taken)



- Números del 1-8 y un hueco
- Puedes mover una pieza al hueco
- En las esquinas solo puedes mover dos piezas. En el centro puedes mover 4 piezas. En los laterales 3 piezas.

Solución

- Estado inicial: El que se defina como configuración inicial del problema desde el que parte el puzzle.
- Estado final: Puzzle totalmente ordenado con el espacio en la esquina inferior derecha.
- Coste de los operadores: 1 por cada operador independientemente de la acción realizada

Búsqueda.

Tenemos que tener en cuenta algunos factores como su complejidad computacional en tiempo y en espacio, su completitud, su optimalidad y su coste total (coste de buscar la solución + coste de ejecutarla).

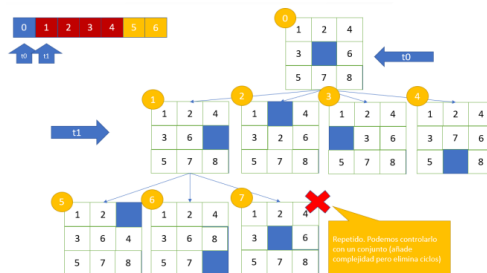
Tipos de búsqueda.

- **Búsqueda no informada o a ciegas:** búsqueda sin añadir información del dominio.

Son algoritmos que pueden ser aplicados a cualquier tipo de problema, pero tiene una capacidad para resolverlos muy limitada, sobre todo en problemas de complejidad exponencial

- Primero en Anchura: se explora el espacio de búsqueda procesando todos los sucesores de un nodo (la frontera de exploración es una cola). Es completo, si tiene coste uniforme la solución es óptima. Su complejidad en tiempo y espacio es

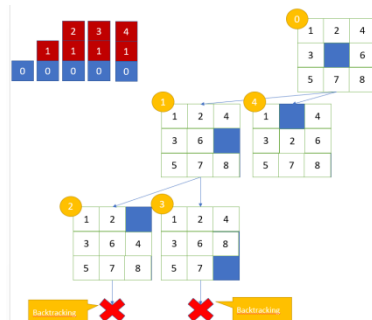
$O(r^p)$ donde r =sucesores y p =profundidad media a la que se encuentre la solución.



- Búsqueda con coste uniforme: si el coste del camino no es siempre el mismo (cada operador no vale lo mismo) podemos modificar el algoritmo de primero en anchura para que siga siendo óptimo. Para ello expandiremos primero aquellos nodos con el menor coste acumulado. Si no hay ciclos y todos los sucesores tienen un coste positivo mayor que 0, entonces seguirá siendo completo y óptimo.

- Primero en profundidad: se exploran los nodos que se acaban de expandir primero en vez de explorar primero todos sus hermanos (la frontera de exploración es una pila). Si un nodo no tiene sucesores, entonces volvemos para atrás y continuamos la búsqueda. NO es completo, no es

óptimo y si complejidad en tiempo y espacio es $O(r * p)$



- Variantes de primero en profundidad:

















- Profundidad limitada: completo si la solución está antes del límite, no óptimo.
- Profundidad iterativa (completo y óptimo): el coste en tiempo es similar a anchura, pero el coste en memoria es mucho menor.
- Búsqueda bidireccional: desde el estado inicial hasta el estado final y otra desde el estado final al estado inicial.

- **Búsqueda informada o heurística:** añade conocimiento del dominio para guiar a la búsqueda y que esta no expanda nodos innecesariamente.

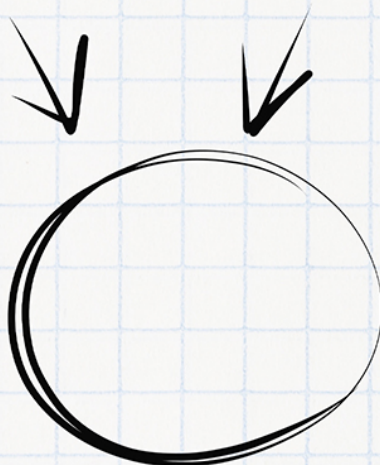
- Hay múltiples formas de implementar una heurística. Normalmente es una función que \hat{u} ntúa un camino en base a cuan prometedor es.

Imagínate aprobando el examen

Necesitas tiempo y concentración

Planes	 PLAN TURBO	 PLAN PRO	 PLAN PRO+
 Descargas sin publi al mes	10 	40 	80 
 Elimina el video entre descargas			
 Descarga carpetas			
 Descarga archivos grandes			
 Visualiza apuntes online sin publi			
 Elimina toda la publi web			
 Precios Anual <input type="checkbox"/>	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo,
¿Qué nota vas a sacar?



WUOLAH

Sistemas Inteligentes



Comparte estos flyers en tu clase y consigue más dinero y recompensas



Banco de apuntes de la

WUOLAH

1

Imprime esta hoja

2

Recorta por la mitad

3

Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes

4

Llévate dinero por cada descarga de los documentos descargados a través de tu QR



- Buscar una función heurística adecuada es clave para que los algoritmos basados en heurísticas tengan buenos resultados.
- El coste de calcular la función heurística idealmente debería ser el menor posible y nunca mayor que el coste de explorar el camino o no tendría sentido.

Características de las funciones heurísticas.

- Una función heurística h_1 es más informada que otra h_2 si ambas son admisibles y además para cualquier nodo.

$$h(n) \geq h_1(n) \geq h_2(n)$$
- Una función heurística es consistente si y sólo si para todos los nodos n_i y n_j (siendo n_j descendiente de n_i) se cumple que:

$$h'(n_i) - h'(n_j) \leq h(n_i, n_j)$$

El coste de ir del padre al descendiente es siempre mayor que la función heurística.

Ejemplo misioneros y caníbales

3 misioneros y 3 caníbales en la orilla de un río junto con 1 bote
El objetivo es que pasen todos a la otra orilla pero:
- Deben cruzar usando el bote
- En el bote sólo pueden ir 1 o 2 personas
- En ninguna de las orillas puede haber más caníbales que misioneros



Relajamos el problema

Asumir que hay infinitas barcas tanto en un lado como en otro.

Hemos eliminado restricciones => simplificado el problema a una solución más sencilla.

función heurística asumiendo que C son los caníbales y M los misioneros:

$$h_1 = \frac{C + M}{2}$$

Para C=2 y M=2 => 2

Te llevas 1 y 1 en el viaje 1 y 1 y 1 en el viaje 2 (no hace falta que

Pero ojo con la admisibilidad

Problema relajado en el que los caníbales nunca se comen a los misioneros.

La solución es que en cada viaje de ida y vuelta podemos transportar a una persona (la otra tendrá que volver con la barca).

$$h_2 = 2 \cdot (C + M) - 1$$

Asumimos que la barca siempre será 1 y estará a la izquierda

Si tenemos 2 Cs y 2 Ms => 7, pero su coste real es 5, **estamos sobreestimando el coste**

¿Cual es mejor la primera heurística o la segunda?

La más informada será mejor en este caso:

$$h_2(n) \geq h_1(n) \forall n$$

Por lo tanto la segunda heurística es mejor y obtendrá mejores resultados.

¿Como podemos arreglarla la heurística?

Debe ser -3 no -1 ¿Por qué?

Analicemos:

- En el primer viaje movemos a 2 cualesquiera en un viaje.
- Luego en la orilla quedarán (C+M) - 2 y hacemos **2 viajes**
- Hasta que haya (C+M) = 1 se llevará a 1 por cada 2 viajes, por tanto, se lleva **(C+M-3)-2 viajes**
- Y después hay **un viaje adicional** para llevarse al último

2 + (C+M-3)*2 + 1, desarrollando: 2 + 2C + 2M - 6 + 1 = 2C + 2M - 3.

$$h_2 = 2 \cdot (C + M) - 3$$

- **Búsqueda primero el mejor (voraz):** selecciona de los sucesores el primero a expandir siendo este el más prometedor (el que se estime que tenga menor coste).
 - Se usa una cola de prioridad y siempre elegimos el que sea mejor.
 - En la búsqueda voraz no tenemos en cuenta el coste de llegar a un nodo, sólo miramos al futuro (nunca replanificar salvo que lleguemos a un callejón sin salida).
 - Es muy rápida.
 - No es completa ni admisible.

A*

A* minimiza el coste total de la solución. De forma que puede abandonar un camino ya explorado si detecta que este ya no es el más prometedor y continuar con otro.

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

- Es completo si f es admisible
- Es óptimo si f es admisible
- Si no se puede calcular h' admisible se utiliza A* con heurísticas ligeramente no admisibles para obtener soluciones ligeramente sub-óptimas.

Algoritmo A*

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

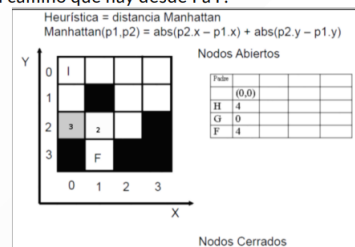
A* mantiene el mejor camino a todos los nodos procesados. Guarda dos listas o colecciones, la colección de nodos procesados (cerrada) y una cola de prioridad ordenada (abierta).

Para cada nodo:

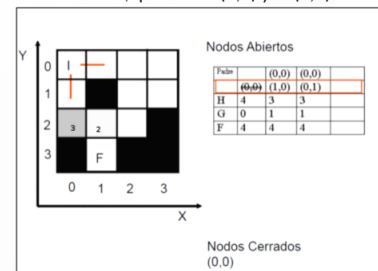
1. Genero sus hijos.
2. Exploro sus hijos: el orden (heurística + coste acumulado).
3. Si el hijo no abierta -> calculo el coste total previsto y guardo el camino.
4. Si el hijo (está ya en abierta) y encuentro un camino mejor -> guardo el mejor y deshecho el otro.

Ejemplo de A*

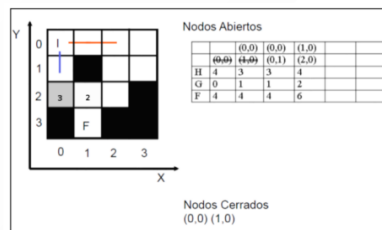
Buscamos el camino que hay desde I a F.



Generamos los sucesores, que son el (1,0) y el (0,1)

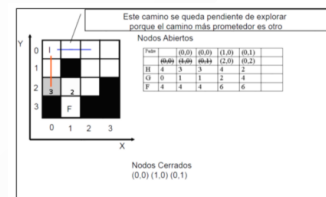


Metemos en la lista de cerrados el nodo procesado.
Generamos los sucesores de (1,0) => (2,0).

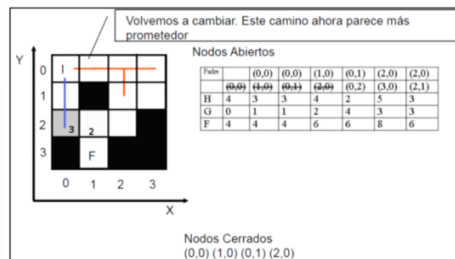


Metemos (1,0) en la lista de cerrados.
Generamos los sucesores de (0,1) ya que es el nodo con menos fitness. => (0,2)

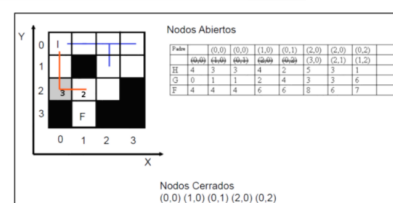
Metemos en la lista de cerrados el nodo (0,1)



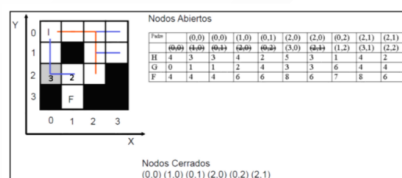
Generamos sucesores del nodo (2,0) => (3,0), (2,1)



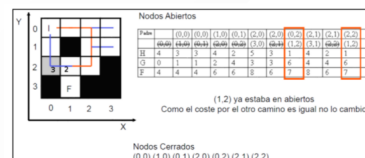
Generamos sucesores del nodo (0,2) => (1,2)



Seleccionamos el nodo (2,1) como el más prometedor y generamos sus sucesores => (3,1), (2,2)

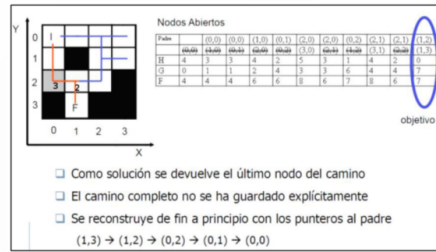
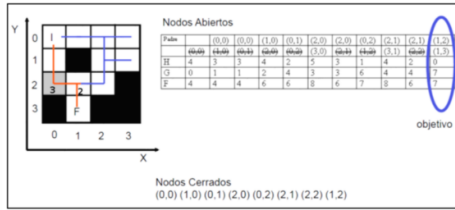


El siguiente nodo a procesar es (2,2) que genera como sucesor el (1,2) que ya está en abierta. Tenemos que comprobar si ese nodo por el camino anterior era más costoso llegar a no.



A partir del destino podemos reconstruir el camino para llegar a ese nodo moviéndonos por sus padres hasta llegar al nodo inicial.

Generamos los sucesores del nodo (1,2) => (1,3) que es el destino.



Búsqueda con adversarios.

Son sistemas de búsqueda para más de un agente. En concreto vamos a ver ejemplos para 2 adversarios:

Por convenio se dice que el jugador que empieza la partida es MAX y el segundo es MIN por lo que las posiciones pares del árbol de búsqueda corresponden a max y las impares a min.

Podemos por tanto asumir que el jugador max intenta maximizar la función objetivo y el jugador min minimizarla. Esto se aplica a problemas de suma cero donde las acciones de uno de los jugadores perjudican al resto y viceversa.

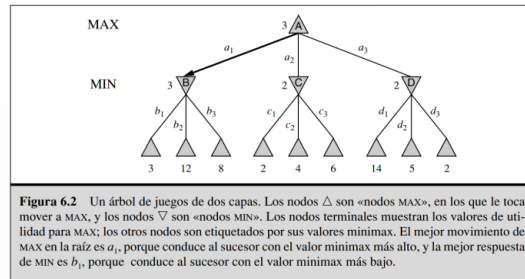
Ejemplo de Mini-MAX

¿Como sabemos si una acción me beneficia?

Cada nodo max le asignamos el máximo de los valores de sus hijos
Cada nodo min le asignamos el mínimo de los valores de sus hijos

$$Valor(n) = \begin{cases} utilidad(n) & \text{si } n \text{ es terminal} \\ MAX_{s \in S} Valor(s) & \text{si } n \text{ es un estado max} \\ MIN_{s \in S} Valor(s) & \text{si } n \text{ es un estado min} \end{cases}$$

El sub-árbol solución es aquel que representa para max la mayor rentabilidad y para min la menor penalización. Esto establece una estrategia o un plan que llevará el agente para vencer al oponente.



Optimizaciones de minimax.

No podemos calcular todo ya que es exponencial en tiempo. Lo que debemos es establecer un límite de nodos o de profundidad. En función de ese límite, el agente será más o menos inteligente.

El problema en estos casos es que la función de utilidad en los terminales no se alcanza y se tomarán acciones que pueden no ser totalmente óptimas.

También podemos explorar hasta cierto nivel y a partir de ahí estimar con una heurística cual sería el mejor camino o usar jugadas completas probabilísticas.

Poda Alfa-Beta.

El problema de la búsqueda minimax es que el número de estados que tiene que examinar es exponencial en el número de movimientos. No podemos eliminar el exponente, pero podemos dividirlo.

alpha es el valor de la mejor opción (el más alto= que hemos encontrado hasta ahora para MAX.

beta es el valor de la mejor opción (más bajo) que hemos encontrado hasta ahora para MIN.

La búsqueda alfa-beta actualiza el valor de Alpha y Beta se va recorriendo el árbol y poda las ramas restantes cuando el valor del nodo a podar es peor que Alpha o Beta.