# Práctica 2

# Representación

# y búsqueda de la

# librería AIMA

Javier Pellejero Ortega & Zhaoyan Ni

Inteligencia Artificial

Grupo 11

Doble grado Matemáticas e ingeniería informática

1. **Puzle de 8.** 
   1. **Tabla comparativa de los resultados obtenidos con los 5 algoritmos.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Búsqueda en anchura | Método voraz (Fichas descolocadas) | Método voraz (Distancia Manhattan) | Búsqueda A\* (Fichas descolocadas) | Búsqueda A\* (Distancia Manhattan) |
| Coste del camino | 30 | 94 | 68 | 30 | 30 |
| Nodos explorados | 181.058 | 672 | 269 | 113.020 | 9.627 |
| Tamaño de la cola | 1.082 | 424 | 187 | 31.602 | 5.134 |
| Máximo tamaño de la cola | 32.766 | 425 | 188 | 32.264 | 5.135 |

* 1. **Optimalidad de las soluciones encontradas. Explicad qué algoritmos encuentran la solución óptima y cuáles no y analizad por qué ocurre esto.**

La *búsqueda en anchura* encuentra la solución óptima, que es 30 movimientos. Este algoritmo explora todo el árbol de nodos hasta llegar a una solución y, además, si el coste de los operadores de búsqueda es aproximadamente uniforme la solución es óptima. Como *A\** con ambas heurísticas cifran el camino en coste 30, también son soluciones óptimas.

El *método voraz* encuentra la solución, pero no una óptima con ninguna de las dos heurísticas. Esto es debido a que el método voraz prioriza el nodo más prometedor en función únicamente del coste estimado para llegar a la solución objetivo desde el estado escogido y no tiene en cuenta el coste previo para llegar a dicho estado. Es decir, podemos operar sobre un estado aparentemente prometedor y obtener nuevos estados igual de prometedores, pero no estamos teniendo en cuenta que la realización de las operaciones tiene ya un coste implícito.

* 1. **Coste de memoria. Analizad las diferencias, respecto al coste de memoria, de los 5 algoritmos y explicad a qué se deben estas diferencias**

El método voraz

* 1. **Indicad cuál es el mejor de los 5 algoritmos para la resolución del puzle de 8. Justificad vuestra respuesta.**

El mejor de los 5 algoritmos es *A\** con la heurística *distancia Manhattan* puesto que es uno de los tres que encuentran la solución óptima y además es el que menos memoria ocupa y visita menos nodos de los tres.

Sin embargo, en el caso de que no nos preocupe la optimalidad de la solución, sino una solución cualquiera, podemos quedarnos con el método voraz con heurística *Manhattan* pues es método que visita menos nodos y menos memoria necesita.

* 1. **Indicad donde se definen los estados y los operadores del puzle de 8 y explicad como están implementados.**

Los estados y operadores están definidos en:

**aima/core/agent/environment/eightpuzzle/EightPuezzleBoard.java**

Los estados están implementados como un array de enteros de tamaño 9, la posición *i* del array simboliza la celda *i*, numeradas de izquierda a derecha y de arriba a abajo, cuyo contenido del array es el número del 1 al 8 correspondiente a las piezas del puzle o 0 en representación del hueco.

Los operadores están definidos como *arriba*, *izquierda*, *arriba*, y *abajo* y representan a donde se mueve el hueco en cada cambio de estado.

* 1. **Indicad donde se define el test de estado objetivo en el puzle de 8 y explicad como está implementado.**

Podemos encontrar el test de estado objetivo en:

**aima/core/agent/environment/eightpuzzle/EightPuezzleGoalTest.java**

Su implementación es sencilla: se define un array de enteros como los mencionados en el anterior apartado correspondiente al estado objetivo y se comprueba si el array que identifica al estado actual es idéntico al definido.

* 1. **Indicad donde se define la heurística Manhattan y explicad como está implementada.**

Encontramos la implementación de la heurística *Manhattan* en:

**aima/core/agent/environment/eightpuzzle/ManhattanHeuristicFunction.java**

La implementación de esta heurística

1. **Búsqueda de caminos.** 
   1. **Tabla comparativa de los resultados obtenidos con los 4 algoritmos (cada uno con *TreeSearch* y *GraphSearch*).**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Coste del camino | Nodos explorados | Tamaño de la cola | Tamaño máximo de la cola |
| Breadth First Search  *(TreeSearch)* | 719 | 118 | 203 | 203 |
| Breadth First Search  *(GraphSearch)* | 719 | 16 | 2 | 7 |
| Coste uniforme  (*TreeSearch)* | 687 | 693 | 1.143 | 1.144 |
| Coste uniforme  *(GraphSearch)* | 687 | 17 | 1 | 5 |
| Método Voraz  (*TreeSearch)* | 719 | 6 | 12 | 13 |
| Método Voraz  *(GraphSearch)* | 719 | 6 | 7 | 8 |
| Búsqueda A\*  (*TreeSearch)* | 687 | 16 | 29 | 30 |
| Búsqueda A\*  *(GraphSearch)* | 687 | 13 | 6 | 9 |

* 1. **Optimalidad de las soluciones encontradas. Explicad qué algoritmos encuentran la solución óptima y cuáles no y analizad por qué ocurre esto.**

Las soluciones óptimas son encontradas por *Coste uniforme* (tanto con *TreeSearch* como con *GraphSearch*) y *Búsqueda A\** (tanto con *TreeSearch* como con *GraphSearch*).

La *búsqueda en anchura* no encuentra una solución óptima puesto que los costos de los caminos no son uniformes; encuentra el camino más corto, pero no el menos costoso.

El *método voraz* tampoco encuentra una solución óptima al igual que en el caso del *Puzle 8*. Las razones son las mismas, el método prioriza sobre el camino más eficiente al siguiente nodo, pero no tiene en cuenta el coste previo que supone llegar al primer nodo en cuestión.

* 1. **Coste de memoria. Analizad las diferencias, respecto al coste de memoria, de los 4 algoritmos y explicad a qué se deben estas diferencias.**
  2. **Indicad cuál es el mejor de los 4 algoritmos para la búsqueda de caminos entre dos ciudades. Justificad vuestra respuesta.**

Podemos suponer que no nos vale con obtener una ruta solución cualquiera, sino la más corta. Luego nuestro algoritmo debe encontrar la solución óptima recorriendo el menor número de nodos posible y minimizando el uso de memoria. En este caso podemos optar entre dos algoritmos que han arrojado soluciones óptimas con costes en memoria y vista de nodos muy similares: *A\** con *GraphSearch* y *Coste uniforme* con *GraphSearch*. El primero visita menos nodos que el segundo, pero el segundo que un ligero menor uso de memoria.

* 1. **Indicad donde se definen los estados y los operadores de la búsqueda de caminos y como están implementados.**

Los estados están definidosen:

**aima/core/agent/environment/map/MapEnvironmentState.java**

1. **Esquemas generales de búsqueda. Indicad en que paquete se encuentran los esquemas generales de búsqueda *TreeSearch* y *GraphSearch* y explicad la diferencia entre ambos.**

La implementación de la búsqueda *TreeSearch* se encuentra en la clase *TreeSearch* y la de *GraphSearch* se encuentra en la clase *GraphSearch*. Ambas clases están en la carpeta:

**aima/core/search/framework/qsearch.**

La mayor diferencia entre ambas búsquedas es que la búsqueda *TreeSearch* no tiene control de repeticiones mientras que la búsqueda *GraphSearch* sí lo tiene. Por tanto, podemos observar que en la implementación de la búsqueda *GraphSearch* hay un atributo con nombre *explored* donde guardamos los nodos explorados. Después de extraer el nodo actual dentro del conjuntos de nodos vivos y comprobar que no contenga el estado final, añadimos el nodo actual al conjunto de los nodos cerrados y añadimos los hijos del nodo actual al conjunto de nodos abiertos sólo si no están ni en abiertos ni en cerrados.

1. **Algoritmo primero en anchura. Indicad en que paquete se encuentra el algoritmo de búsqueda primero en anchura e indicad en qué momento se comprueba si un estado es objetivo y por qué creéis que se ha tomado esa decisión.**

La implementación de la búsqueda primero en anchura se encuentra en la clase *BreadthFirstSearch* que está situada en la carpeta:

**aima-core/src/main/java/aima/core/search/uniformed**

La comprobación de si un estado es objetivo se realiza cuando se llama a la función *findState* y en la función *findActions*. (no estoy muy segura de esto)

1. **Algoritmo de búsqueda informada. Indicad cual es el algoritmo qué habéis elegido, en que paquete y clase se encuentra su implementación y cómo esta implementado.**

Entre los algoritmos de búsqueda informada implementados, hemos elegido el método voraz que está en:

**aima/core/search/informed**

Para implementar el algoritmo, se crea la clase *GreddyBestFirstSearch* que se encuentra en la dirección anteriormente citada. Esta clase se hereda de la clase *BestFirstSearch* y utiliza la interfaz *HeuristicFunction* y la clase abstracta *QueueSearch.*

La clase *BestFirstSearch* hereda de la clase *PrioritySearch* que está en aima/core/search/framework. Para entender la implementación de la clase *BestFirstSearch,* tenemos que saber qué es búsqueda best-first. La búsqueda best-first es un algoritmo en el que se selecciona el nodo con menor estimación de coste según una función de evaluación para explorar. Por tanto, para crear un objeto de la clase *BestFirstSearch* hay que pasar una función de evaluación y una estrategia de búsqueda de exploración espacial. La función de evaluación pasada como argumento se guarda en el atributo *evalFunc* de la clase *EvaluationFunction* y con este atributo creamos un comparador para poder comparar el coste estimado de los nodos.

La interfaz *HeuristicFunction* sólo tiene un atributo que guarda el menor coste estimado necesario para pasar del estado actual hasta llegar al estado objetivo.

La clase abstracta *QueueSearch*

La clase *GreddyBestFirstSearch* sólo contiene una función: la función constructora de la clase que recibe dos argumentos, uno de la clase *QueueSearch* que se llama *impl* que es una estrategia de búsqueda de exploración espacial (por ejemplo, *TreeSearch* y *GraphSearch*) y otro de la clase *HeuristicFunction* con nombre *hf*. Esta función llama a la función constructora de la clase padre pasando como argumento *impl* y un argumento de la clase *GreedyBestFirstEvaluationFunction* creada usando el argumento *hf.* Como el nombre indica, en la clase *GreedyBestFirstEvaluationFunction* está implementada la función de evaluación asociada al método voraz que coincide con la función heurística.