# 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.
   2. Optimalidad de las soluciones encontradas. Explicad qué algoritmos encuentran la solución óptima y cuáles no y analizad por qué ocurre esto
   3. Coste de memoria. Analizad las diferencias, respecto al coste de memoria, de los 5 algoritmos y explicad a qué se deben estas diferencias
   4. Indicad cuál es el mejor de los 5 algoritmos para la resolución del puzle de 8. Justificad vuestra respuesta.
   5. Indicad donde se definen los estados y los operadores del puzle de 8 y explicad como están implementados.
   6. Indicad donde se define el test de estado objetivo en el puzle de 8 y explicad como está implementado.
   7. Indicad donde se define la heurística Manhattan y explicad como está implementada.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Breadth First Search | 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 |



*Bread First Search* y *A\** con ambas heurísticas encuentran la solución óptima, que es de coste 30 asumiendo cada movimiento con coste una unidad.



Fj



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.



En aima-core/src/main/java/aima/core/agent/environment/eightpuzzle/EightPuezzleBoard.java

Y en aima-core/src/main/java/aima/core/agent/environment/eightpuzzle/EightPuezzleFuncionFactory.java



En aima-core/src/main/java/aima/core/agent/environment/eightpuzzle/EightPuezzleGoalTest.java

En aima-core/src/main/java/aima/core/agent/environment/eightpuzzle/ManhattanHeuristicFunction.java

1. **Búsqueda de caminos.** 
   1. Tabla comparativa de los resultados obtenidos con los 4 algoritmos (cada uno con *TreeSearch* y *GraphSearch*).
   2. Optimalidad de las soluciones encontradas. Explicad qué algoritmos encuentran la solución óptima y cuáles no y analizad por qué ocurre esto.
   3. Coste de memoria. Analizad las diferencias, respecto al coste de memoria, de los 4 algoritmos y explicad a qué se deben estas diferencias.
   4. Indicad cuál es el mejor de los 4 algoritmos para la búsqueda de caminos entre dos ciudades. Justificad vuestra respuesta.
   5. Indicad donde se definen los estados y los operadores de la búsqueda de caminos y como están implementados.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 |



Las soluciones óptimas son encontradas por *Coste unirforme* (tanto con *TreeSearch* como con *GraphSearch*) y *Búsqueda A\** (tanto con *TreeSearch* como con *GraphSearch*). Poeque son mazo de listos.



Sf



fgf

Los estados están definidosen aima-core/src/main/java/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/search/uniformed**

La comprobación de si un estado es objetivo se realiza cuando se llama a la función *findState* y la función *findActions*.

Se ha tomado esta decisión porque si ya hemos llegado al estado objetivo, no hace falta explorar más nodos vivos y comprobando si un estado es objetivo o no en las funciones *findState* y *findActions* nos da más eficiencia.

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* tiene 4 constantes estáticos del tipo string que pueden ser reiniciados a 0 o ser renovados, 4 atributos: *nodeExpander, frontier, earlyGoalTest, metrics* y 11 funciones, de las cuales 8 están implementadas. Con el atributo *nodeExpander* podemos crear y explorar nodos, también podemos saber el número de nodos explorados y el coste del camino. En el atributo *frontier* guardamos los nodos abiertos que pueden ser añadidos, eliminados de *frontier* por las funciones de la clase, también podemos preguntar si un nodo es nodo vivo o no. En *metrics* están guardados los clave-valor para análisis de eficiencia. Además, la función *findNode* recibe un problema y una cola de nodos vivos e intentar encontrar un nodo que hace referencia a un estado de objetivo.

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.