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.**

   a) **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

   b) **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.

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

Para empezar cabe resaltar que los algoritmos que usan la heurística *Manhattan* obtienen mejores resultados en memoria que los que usan *fichas descolocadas*. De aquí podemos obtener la conclusión de que la primera heurística es mejor, posiblemente porque es más ***informada*** que la segunda.

Sin entrar en distinciones de heurísticas, podemos decir que el algoritmo más eficiente en cuanto a uso de memoria es el *método voraz*. Recordar que este algoritmo sacrifica la búsqueda de una solución óptima y elige el camino más prometedor sin tener en cuenta el coste previo de llegar al estado en cuestión.

La *búsqueda A** es el segundo en consumo de memoria. Consume más que el algoritmo anterior pues este sí tiene en cuenta el coste previo de llegar al siguiente estado a inspeccionar, lo que nos garantiza en muchos casos una solución óptima.

Por último, la *búsqueda en anchura* recorre sin ningún tipo de criterio de elección, todos los hijos de un estado, previa visita previa de todos los hermanos del primero. Esto provoca una visita masiva de nodos.

d) 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.

e) 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/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.

f) 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/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.

g) Indicad donde se define la heurística Manhattan y explicad como está implementada.

Encontramos la implementación de la heurística *Manhattan* en:

aima/core/environment/eightpuzzle/ManhattanHeuristicFunction.java

La implementación de esta heurística es sencilla. Visitamos cada elemento del array de enteros que representa el estado a tratar, correspondiente con una celda, vemos que elemento aloja y a qué distancia está de su posición original (hay que tener en cuenta que no existen movimientos en diagonal sólo verticales y horizontales a la hora de medir dichas distancias). Si la pieza está en su celda, la distancia es cero. Así, se suman todas las distancias obtenidas de las 9 celdas y se devuelve el total correspondiente con la heurística.

2. Búsqueda de caminos.

a) 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

b) 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.

c) Coste de memoria. Analizad las diferencias, respecto al coste de memoria, de los 4 algoritmos y explicad a qué se deben estas diferencias.

Para empezar cabe la pena destacar la diferencia de memoria usada por el mismo algoritmo con el uso de *TreeSearch* y *GraphSearch*. Este último previene la repetición de estados; es decir, memoriza los estados que ya ha comprobado para no comprobarlo sucesivas veces, esto produce un uso de memoria mucho menor.

Teniendo en cuenta el algoritmo utilizado, el *método voraz* es el que menos memoria utiliza. Recordemos que busca una solución directa cogiendo los mejores candidatos sin tener en cuenta el coste de llegar a los estados previos. Así logra una solución rápida, visitando pocos nodos, pero pocas veces óptima.

En segundo lugar, tenemos la *búsqueda A**, recordemos que busca candidatos óptimos, pero siempre teniendo en cuenta el coste previo de llegar al estado en cuestión. Esto hace considerar más estados que el algoritmo anterior a cambio de encontrar una solución óptima.

A continuación, tenemos el *coste uniforme*. Este algoritmo considera primero los nodos que tienen menor coste hasta llegar a ellos y continua hasta encontrar la solución óptima. En algunos casos, si alguna de las primeras soluciones encontradas es óptima, el algoritmo puede realizarse de manera rápida visitando pocos nodos y utilizando poca memoria, como en el caso con *GraphSearch*; sin embargo, podemos ver que puede ser realmente costoso en otros casos, sin ir más lejos, con *TreeSearch*.

Por último, tenemos la *búsqueda en anchura*, que no sólo no encuentra la solución óptima, sino que añade muchos nodos a visitar tras analizar alguno de ellos (sobre todo con *TreeSearch*) y además no hay un criterio de elcción del siguiente. Esto provoca que sea el más costoso en memoria.

d) 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 c*oste uniforme* con *GraphSearch*. El primero visita menos nodos que el segundo, pero el segundo que un ligero menor uso de memoria.

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

Los estados están definidos en:
aima/core/agent/environment/map/SimplifiedRoadMapOfPartOfRomania.java

La implementación es más simple de lo que parece. Tenemos un *Map* que hace las veces de matriz de *Strings*. Están definidas las ciudades como constantes (*Strings*) y desde la constructora se inicializa el mapa con los enlaces (en ambos sentidos) entre las ciudades correspondientes y el coste del camino entre ambas. Podemos resumir que el mapa hace las veces de grafo, donde los vértices hacen de estados (ciudades) y las aristas de operadores (con sus costes).

3. 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.

4. 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.

5. **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.