Práctica 2

Inteligencia Artificial

Borja Aday Guadalupe Luis
Eduardo Bryan de Renovales
3ºA, Grupo 5

Índice

1. Problema del 8 Puzzle.

1.1. Tabla comparativa de algoritmos.

Tipo de búsqueda	Coste de ejecución	Nodos expandidos	Tamaño de la cola	Tamaño máximo de la cola
Breadth First Search	30	181058	365	24048
Greedy Best First Search (MisplacedTile)	116	803	525	526
Greedy Best First Search (Manhattan)	66	211	142	143
A* Search (MisplacedTile)	30	95920	23489	23530
A* Search (Manhattan)	30	10439	5101	5102

1.2. Optimalidad de los algoritmos.

Los métodos voraces no son capaces de encontrar la solución óptima porque las heurísticas no son consistentes, esto provoca que se desprecie la rama de la solución óptima. Aunque a cada paso el algoritmo calcule la rama que más se acerca a la solución, este no tiene en cuenta el coste completo del camino que realiza y por ello no encuentra la solución óptima.

Los algoritmos A* encuentran la solución óptima con ambas heurísticas porque las dos son admisibles y al llamar al A* hace uso de una búsqueda GraphSearch, lo que nos asegura encontrar la mejor solución. Si decidimos ejecutar el A* con las mismas heurísticas pero con TreeSearch, el algoritmo no llega a terminar.

Por otra parte, el algoritmo de búsqueda en anchura, si existe solución para el problema, encuentra siempre una de longitud mínima ya que recorre todos los nodos posibles cerrando aquellos por los que ya ha pasado (un nodo por el que ya has pasado debe cerrarse porque si te vuelves a encontrar ese nodo más adelante, el coste de hasta llegar a él ha sido mayor y por tanto debemos descartarlo) y explorando el resto que todavía pueden esconder una solución.

1.3. Uso de memoria de los algoritmos.

Los métodos de búsqueda más eficientes respecto a la memoria son los voraces ya que descartan las respuestas menos prometedoras, y con ello mantienen menos nodos en la cola. Entre estos dos se aprecia que la heurística manhattan es mejor con relación a la memoria ya que facilita encontrar la mejor solución y con ello reduce la cantidad de nodos en la cola.
Por otra parte, los algoritmos A* son peores de acuerdo con la memoria pues estos no descartan los nodos que no son prometedores. Al igual que con los algoritmos voraces, se observa que la heurística Manhattan es mejor que la heurística de fichas descolocadas respecto a la memoria, por el mismo motivo que antes.

En cuanto al algoritmo de búsqueda en anchura es el peor de todos ya que intenta hallar el resultado por mera fuerza bruta y con ello es necesario explorar y expandir muchos más nodos y tener muchos más en la cola. A pesar de que al final de la ejecución el tamaño de la cola no

era el mayor de todos, durante la ejecución fue el que generó una cola más grande y el que más nodos expandió por lo que fue el que más memoria utilizó.

1.4. Mejor algoritmo para el 8 Puzzle.

El mejor algoritmo para solucionar el problema del 8 puzzle es el A* con la heurística Manhattan, a pesar de que ha expandido 10.439 nodos y haber alcanzado 5.102 elementos en la cola, su uso de memoria y tiempo de cálculo han estado en un rango más que aceptable para habernos proporcionado la solución óptima.

La búsqueda del primero en anchura también nos ha dado la solución óptima pero como podemos comprobar su tiempo de ejecución y memoria ha sido mucho mayor.

Por tanto para resolver este problema debemos usar la búsqueda A* con la heurística Manhattan.

1.5. Definición de estados y operadores.

La representación de un estado del tablero se define en la clase EightPuzzleBoard y la podemos encontrar en …\aimacore\src\main\java\aima\core\environment\eightpuzzle, esta clase contiene un atributo de tipo int[] que nos servirá para representar el estado del tablero.

Las acciones que podemos ejecutar sobre dicho tablero, también se definen en esa misma clase, se crean los Action correspondientes a cada acción posible sobre un tablero y se crea una función que será asociada con dicho Action. Esta asociación se lleva a cabo en la clase EPResultFunction que está dentro de la clase EightPuzzleFunctionFactory.

1.6. Definición del estado objetivo.

La clase donde se define el estado objetivo y la función para saber si has llegado hasta él está el directorio ...\aimacore\src\main\java\aima\core\environment\eightpuzzle, en concreto en la clase EightPuzzleGoalTest.

Lo primero que se hace es definir el estado final en una instancia de EightPuzzleBoard, que es la que nos permite representar el estado. Por último, necesitas una función que te compare si el estado dado es igual al estado final, para ello se debe hacer la conversión del State a la clase que representa nuestro estado y sobrescribir la función equals de la clase EightPuzzleBoard para comprobar que efectivamente ambos estados son iguales.

Lo primero que realiza el equals es comprobar si ambos son exactamente el mismo estado, es decir, si ambos apuntan a la misma dirección de memoria. Si esto no ocurre, entonces va posición por posición comprobando que todo está en su sitio, si encuentra alguna pieza descolocada entonces devuelve false y la búsqueda sigue avanzando hasta hallar la solución.

1.7. Heurística Manhattan.

La heurística Manhattan está definida en la clase ManhattanHeuristicFunction, que se encuentra en el directorio ...\aimacore\src\main\java\aima\core\environment\eightpuzzle.

Esta clase consta de dos funciones, una de ellas se encarga de calcular la distancia de una pieza hasta su posición final diciéndole el número de la pieza y la localización actual de la pieza.

Sabiendo esto solo tiene que realizar operaciones aritméticas con valor absoluto pues las posiciones finales son siempre las mismas. Para obtener el valor final de la heurística tenemos la función public double h(Object state) que va recorriendo los números del 1 al 8 obteniendo sus posiciones y pasándole estos datos a la función anterior, al final del bucle tendrá la suma total con el valor final de la heurística en ese estado.

2. Problema de búsqueda de caminos.

2.1. Tabla comparativa de algoritmos.

Tipo de búsqueda	Coste de ejecución	Nodos expandidos	Tamaño de la cola	Tamaño máximo de la cola
Depth First Graph Search	1002	16	3	5
Breadth First Tree Search	719	118	202	203
Breadth First Graph Search	719	16	1	5
Uniform Cost Tree Search	687	693	1	4
Uniform Cost Graph Search	687	17	1	4
Greedy Best First Tree Search	719	6	12	13
Greedy Best First Grap Search	719	6	7	8
A* Tree Search	687	16	29	30
A* Graph Search	687	13	4	6

2.2. Optimalidad de los algoritmos.

Solo los algoritmos de A* y de Coste Uniforme son capaces de encontrar la solución óptima, esto es debido a que la búsqueda de coste uniforme encuentra la solución correcta siempre que los costes de todos los nodos sean positivos y por otra parte los A* son capaces de encontrar la solución óptima debido a que SLD es una heurística consistente.

Por otra parte, los métodos voraces no son capaces de encontrar la solución óptima ya que estos no lo garantizan. Los demás algoritmos no encuentran la solución óptima debido a que tampoco garantizan que la vayan a encontrar, ya que la descartan porque proviene de expandir un nodo que se descarta por no ser prometedor.

2.3. Uso de memoria de loa algoritmos.

Los métodos de búsqueda más eficientes respecto a la memoria son los de coste uniforme, aunque ha necesitado expandir muchos nodos en comparación al resto, la cola nunca superó un tamaño de 4 nodos y acabó con una cola de un solo nodo por lo que durante el proceso no ha utilizado mucha memoria al ir descartando nodos inservibles.

Por otro lado, los algoritmos A* son peores de acuerdo con la memoria pues estos no descartan los nodos que no son prometedores y por ello su cola llega a alcanzar un tamaño

mayor. Esto no quiere decir que hayan usado mucha memoria, pero sí en comparación con la búsqueda de coste uniforme. A pesar de ello, son las dos únicas búsquedas que han conseguido encontrar la solución óptima con un tiempo de ejecución y memoria más que adecuados.

En cuanto al algoritmo de búsqueda en profundidad es el peor de todos si se usa con TreeSearch pues este necesita de la expansión de muchos nodos para encontrar la solución y con ello llega a contener una cola de 203 nodos por expandir, muy superior al resto donde su predecesor solo llega a tener una cola de 30 elementos.

Algo destacable también en cuanto al uso de memoria es que el uso de GraphSearch mejora con creces el tamaño máximo de la cola, en todos los algoritmos de búsqueda utilizados, GraphSearch mejora al TreeSearch por lo que podemos decir que el GraphSearch (a pesar de guardar los nodos cerrados en una lista) nos ayuda a no desperdiciar tanta memoria.

2.4. Mejor algoritmo para la búsqueda de caminos.

Los mejores algoritmos para la búsqueda de caminos entre 2 ciudades son la búsqueda de coste uniforme, si se focaliza en la memoria utilizada y A* si se prioriza la rapidez de la búsqueda. Siendo siempre mejor las versiones GraphSearch de ambos algoritmos. Esto es debido a que la heurística SLD es consistente y esto provoca que no se tengan que comprobar los nodos que ya habían sido cerrados previamente.

Estos dos algoritmos son los mejores debido a que son los únicos capaces de encontrar la solución óptima, realizando ese cálculo en un periodo de tiempo aceptable y con un consumo de memoria razonable.

2.5. Definición de estados y operadores.

La definición de los estados de la búsqueda de caminos se encuentra en el directorio .../aima/core/src/main/java/aima/core/environment/map/ en la clase MapEnvironmentState y los operadores de la búsqueda de caminos se definen entre la clase MoveToAction y MapEnvironment. Los estados están implementados por medio de un Map donde se guarda un agente y un par, formado por la ubicación y la distancia recorrida.

Los operadores están implementados por medio del cambio de la ubicación y el cálculo de la nueva distancia recorrida, que se realiza en MapEnviroment en la función executeAction, aunque el operador de mover al agente se encuentra definido en MoveToAction y es MapEnviroment el que llama a MoveToAction para realizar el cambio de posición.

3. TreeSearch vs. GraphSearch

Los algoritmos de búsqueda generales TreeSearch y GraphSearch se encuentran en el directorio ...aima-core\src\main\java\aima\core\search\framework donde podemos encontrar las clases llamadas TreeSearch y GraphSearch.

TreeSearch es una clase muy sencilla con una única función que recibe un nodo y lo expande generando una nueva lista de nodos. Por el contrario, la búsqueda GraphSearch es un poco más compleja y consta de 6 funciones diferentes que se combinan para obtener una búsqueda

más compleja y eficiente. Mientras que TreeSearch no tiene control de repetidos y cualquier nodo que le llega lo expande, GraphSearch no pasa dos veces por el mismo nodo. Para ello, al expandir un nodo, agrega este a una lista de nodos cerrados y para expandir uno nuevo siempre comprueba que este no esté en la lista de nodos cerrados. Esto lo hace debido a qué si un nodo ya fue explorado anteriormente y no encontró solución, esta vez tampoco la tendrá, y si la tuvo, su coste fue menor porque el nodo fue encontrado anteriormente.

Si realizamos una búsqueda A* con TreeSearch esperando la solución óptima, nos bastará con que la heurística utilizada sea admisible, pero si por el contrario usamos GraphSearch, tendremos que utilizar una heurística no solo admisible sino también consistente.

Por estos motivos, la búsqueda GraphSearch es un poco más compleja que la TreeSearch pero es más eficiente en cuanto a tiempo y memoria.

4. Primero en anchura.

La búsqueda del primero en anchura la podemos encontrar definida en el directorio del proyecto ...aima-core\src\main\java\aima\core\search\uninformed en la clase BreadthFirstSearch. En esta clase podemos encontrar en su constructora la respuesta a la pregunta.

Lo primero que se encarga la constructora es de activar el TestGoal antes de expandir con esta instrucción: search.setCheckGoalBeforeAddingToFrontier(true).

Además para dejarlo más claro aún, los creadores del AIMA han dejado un comentario diciendo: “Goal test is to be applied to each node when it is generated rather than when it is selected for expansion.”, es decir, que se comprueba si un nodo es estado final antes de que este sea seleccionado para expandirse.

Esto se hace de esta forma porque como la propia búsqueda dice, encuentra el primero en anchura y una vez ha encontrado un nodo que es solución, deja de expandir más nodos innecesarios, pues ya tiene una solución que dar.

5. Búsqueda informada: A*.

Para este apartado hemos decidido analizar la búsqueda informada A*, esta búsqueda la podemos encontrar en el directorio ...\aima-core\src\main\java\aima\core\search\informed, en concreto en dos clases que se complementan: AStarSearch y AStarEvaluationFunction.

- AStarEvaluationFuncition: esta clase se encarga de darle un valor estimado del coste mínimo total desde el inicial hasta algún estado objetivo de cualquier solución que pase por el nodo en el que estamos comprobando.

  Para ello hace uso de una función que realiza la siguiente operación: g(n) + h’(n) donde g(n) representa el coste real del camino que hay que seguir hasta llegar al nodo actual y h’(n) nos da un valor estimado del coste mínimo hasta encontrar un estado objetivo.
  Mientras que g(n) nos aporta realidad del coste que hemos tenido hasta llegar a ese nodo, h’(n) nos da una estimación optimista de lo que nos puede faltar para alcanzar

nuestro objetivo y la combinación de ambos nos ayuda a encontrar un camino prometedor.

- AStarSearch: Esta clase es realmente la búsqueda en sí, por eso extiende a otra clase de búsqueda, y hace uso de la anterior clase a la que se le pasa la heurística a seguir para calcular h'(n).

Este tipo de búsqueda combina la búsqueda de primero en anchura con la búsqueda de primero en profundidad ya que g(n) hace que la búsqueda vuelva a atrás cuando h'(n) está dominada. Por eso esta búsqueda es bastante directa ya que se cambia de camino cada vez que encuentra uno más prometedor. Si la heurística utilizada en esta búsqueda es admisible y usa TreeSearch, entonces podemos asegurar que nos encontrará la solución óptima, si por el contrario usamos GraphSearch, nos exigirá que esta heurística además de admisible sea consistente para hallar la solución óptima.