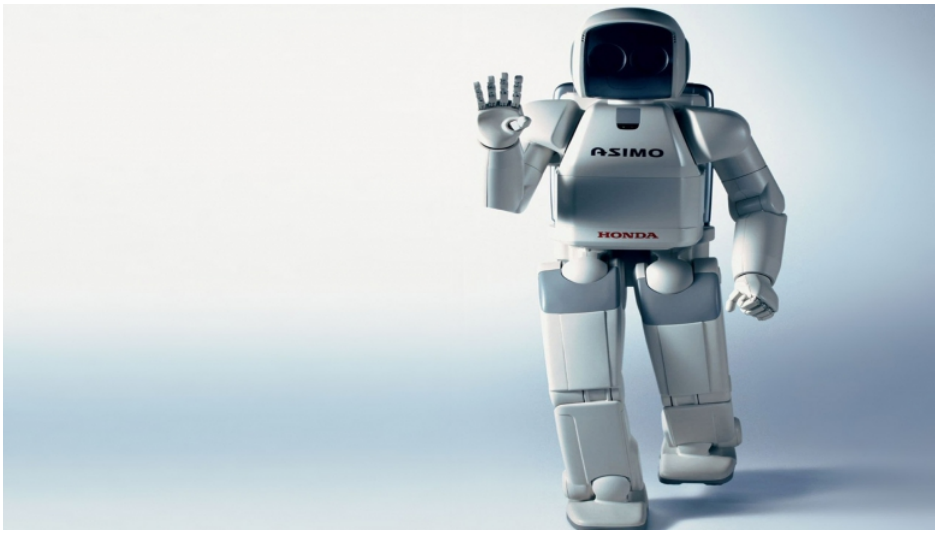
Inteligencia Artificial

Práctica 2



Enrique Ballesteros Horcajo
Ignacio Iker Prado Rujas

5 de Noviembre de 2014

Índice

Búsqueda	`pathCostl`	`nodesExpanded`	`queueSize`	`maxQueueSize`
Búsqueda en anchura	30	181058	365	24048
Método voraz	116	803	525	526
Método voraz(Manhattan)	66	211	142	143
A*(MisplacedTile)	30	95920	23489	23530
A*(Manhattan)	30	10439	5101	5102

Cuadro 1: Tabla del puzzle de 8 extremo, comparando distintos algoritmos.

1. Introducción: AimaDemoApp

Para esta primera parte, tras haber importado el proyecto a Eclipse, hemos ejecutado el fichero `AimaDemoApp.java`, probando las distintas posibilidades que ofrece. Tiene dos modos: *Applications* y *Demos*, y la principal diferencia está en que en la primera disponemos de una interfaz gráfica, útil para comprender mejor qué está haciendo el algoritmo correspondiente.

La primera aplicación que podemos usar es para el coloreado de mapas mediante *backtracking* con distintas heurísticas. Es muy interesante ejecutar la aplicación paso a paso, para ver como se desarrolla el árbol. Además, hay algunos juegos como el puzzle de 8, el 3 en raya, las n-reinas y el conecta 4. Los algoritmos que utilizan son los habituales, con algunas variaciones en la heurística: minimax, poda $\alpha - \beta$, búsqueda en anchura, búsqueda en profundidad, A^*, voraz... Además, incluye una aplicación para encontrar el camino mínimo entre dos nodos de un grafo, que aquí es un mapa de Rumanía y otro de Australia.

Luego, en cuanto a las demos, aparecen algunas aplicaciones nuevas, pero creemos que tiene menos interés porque sólo puedes ver el resultado final y tratar de interpretarlo, y no puedes modificar parámetros como en las aplicaciones anteriores, o incluso interferir en los juegos haciendo el movimiento que quieras.

2. Puzle de 8 extremo

Debido a que en este problema el coste es uniforme, la búsqueda en anchura encuentra la solución en mínima profundidad. Pero como puede verse en `maxQueueSize` el coste en memoria es muy elevado debido a que debemos mantener todo un nivel de nodos abiertos. Esto hace que si la solución estuviese a más distancia sería imposible almacenar todos los nodos necesarios. El método voraz en este caso encuentra una solución en muchos pasos, pero a cambio expande pocos nodos y almacena pocos en memoria. Al ser un problema sencillo, el método voraz puede ser una buena opción, basta añadir una buena heurística como la Manhattan, para que encontremos la solución en pocos pasos y expandiendo pocos nodos. La búsqueda A^* encuentra la solución en 30 pasos expandiendo muchos nodos y almacenando en la cola muchos estados. El uso de la heurística Manhattan mejora el funcionamiento pero aún así hay que almacenar demasiados estados.

En este tipo de juego, que es sencillo, el método voraz parece ser el mejor. A esta conclusión se llega también jugando al juego, pues si haces numerosos movimientos acercando cada número a su posición, sin pensar demasiado se acaba llegando a la solución en relativamente pocos pasos. En juegos más complicados, o buscando la mejor solución, el uso del algoritmo voraz ya no nos resultará tan útil.

Arad-Bucarest	**Depth First**	**Breadth First**	A^*
Step 1	Timisoara - 118	Sibiu - 140	Sibiu - 140
Step 2	Lugoj - 229	Fagaras - 239	RimnicuVilcea - 220
Step 3	Mehadia - 299	Bucarest - 450	Pitesti - 317
Step 4	Dobreta - 374	-	Bucarest - 418
Step 5	Craiova - 494	-	-
Step 6	Pitesti - 632	-	-
Step 7	Bucharest - 733	-	-
Total	733 in 7 steps	450 in 3 steps	418 in 4 steps

Cuadro 2: Para la búsqueda de caminos en el mapa desde Arad hasta Bucarest, comparación entre la búsqueda en profundidad, en anchura y el A^* con SLD.

Búsqueda	`pathCostl`	`nodesExpanded`	`queueSize`	`maxQueueSize`
Búsqueda en profundidad	304	10365	1288	1290
Coste uniforme	31	94454	192	300
A*	31	23136	395	521

Cuadro 3: Tabla sobre el camino entre Altheim y Leibi, comparando algoritmos.

3. AimaDemoApp: Desde Arad hasta Bucharest

SLD es la heurística de línea recta, que elige la ciudad que está más cerca en línea recta del objetivo. En el cuadro adjunto podemos ver como se desarrolla el algoritmo en cada caso. Cabe destacar también que en el primero se expanden 10 nodos, mientras que en el segundo y en el tercero son 5.

El algoritmo de la búsqueda en anchura no encuentra el camino mínimo porque está a nivel 7, y la búsqueda en anchura termina cuando encuentra la solución en el nivel 3. Podría reprogramarse para que encontrara la solución óptima pero para ello tendría que recorrer todo el árbol de soluciones. También podría buscarse la solución óptima hasta un cierto nivel.

4. SearchDemoOsmAgentApp: Desde Altheim hasta Leibi

La búsqueda en profundidad como podemos ver, encuentra un camino largo y que da un rodeo innecesario. Esto se debe a la forma que sigue la búsqueda, a pesar del SLD, extiende el nodo que encuentra primero, por lo que va dando un rodeo hacia la ciudad destino. No obstante, esto hace que expanda pocos nodos pues es una búsqueda muy directa. Evidentemente no encuentra una solución ni mínimamente óptima, que está muy lejos de las obtenidas por el resto de algoritmos. Además el coste en espacio que obtenemos es bastante elevado, lo que es extraño en la búsqueda en profundidad que expande pocos nodos al mismo tiempo.

La búsqueda en coste uniforme y en A^* obtienen resultados similares en longitud, sin embargo el coste uniforme expande muchos más nodos debido a que no expande el nodo que más opciones tiene de encontrar el destino sino el que menos se ha expandido desde el origen. Esto hace que se expandan muchos nodos innecesarios, aunque ambas búsquedas tienen poco coste en espacio, por lo que son viables con un problema mayor.

5. Librerías y definiciones de estado para el puzzle de 8

Para el problema del puzle de 8 se utiliza un array de enteros de tamaño 9, sencillo pero efectivo. El rango de valores para el vector es $0 \leq i \leq 8$ (sin repeticiones), y de este modo el cero corresponde al hueco en el puzle. El resto de casillas se corresponden 1-1 con su representación gráfica, leído de izquierda a derecha y de arriba a abajo. Además, se define en `EightPuzzleBoard.java` en el paquete `aima.core.enviroment.eightpuzzle` como array privado de enteros. También es importante notar que en función de desde donde se llame al constructor de esta clase, se pasa como parámetro un vector diferente (en función de la dificultad).

Figura 1: Puzle de 8 correspondiente a `state = new int`$[]\{1, 2, 5, 3, 4, 0, 6, 7, 8\}$.

En cuanto a los operadores que trabajan con el estado, *se centran en el hueco* y hay cuatro: `moveGapLeft()`, `moveGapUp()`, `moveGapRight()`, `moveGapDown()`. Hacen lo propio, tras comprobar que pueden hacer el movimiento pedido con `CanMoveGap()` calculan en qué posición está el hueco y cambian sus coordenadas en el mapa hacia la izquierda, arriba, derecha o abajo respectivamente. Para ello simplemente hacen un *"swap"*, guardando (por ejemplo para el primer caso) el valor situado a la izquierda del cero, poniéndolo a la derecha y escribiendo un cero a la izquierda. Es interesante ver como transforman un tablero 3×3 en un vector lineal de 9 posiciones. Dadas las coordenadas (x, y) la posición correspondiente en el vector es $3x + y$ (contando como (1,1) la esquina inferior izquierda donde está el 6 en la imagen), y así por ejemplo, en la figura vemos que 7 está en las coordenadas $(2, 1) \implies 3 \cdot 2 + 1 = 7$, luego `state[7] = 7`. Esto se consigue mediante las funciones `getAbsPosition()` y `setValue()`, en la misma clase que la comentada anteriormente.

En el paquete `aima.core.environment.eightpuzzle` se encuentra la clase donde se calcula el número de fichas descolocadas: `MisplacedTilleHeuristicFunction.java`.

6. Librerías y definiciones de estado para el mapa

El programa concreto lo tenemos en el paquete `aima.core.environment.map`, además de las clases abstractas que comparten todos. En cuanto a su parte gráfica, se encuentra en `aima.gui.applications.search.map`. El estado se guarda en la clase `MapEnvironment` que hereda de `AbstractEnvironment`. En `MapEnvironment` se guarda el mapa, la localización, la distancia recorrida y una instancia de `MapEnvironmentState` que tiene los datos sobre la localización concreta en el mapa.

El operador es el `MoveToAction`, que se encarga de actualizar la localización del `Agent`, `MapFunctionFactory` es la clase que gestiona el movimiento, usando `Search` y `MapAgent`, cambiando así el estado en `MapEnvironment`.

7. Conclusiones

Como hemos visto, hay algoritmos que son mejores para un cierto tipo de problema mientras que otros lo son para un problema diferente. Por ejemplo, el voraz para el puzle de 8 nos es útil para resolverlo de manera sencilla, sin necesidad de un algoritmo complejo, mientras que el A^* en el puzle de 8 consume mucho más espacio. En cambio, en problemas mas complicados como en la búsqueda en mapas, con el A^* obtenemos buenos resultados y con el voraz es inviable.

En cuanto a AIMA nos parece una herramienta bastante completa, útil para la comprensión de estos algoritmos. Por último, nos parece una muy buena idea que se desarrolle una alternativa *open source* a Google Maps.

Bibliografía

[1] Russell, S.; Norvig, P,
Artificial Intelligence, a modern aproach.
New Jersey: Pearson, 2010.

[2] Apuntes y transparencias de Inteligencia Artificial,
Doble Grado Matemáticas - Ing. Informática, U.C.M., 2014-2015.