Técnicas de los Sistemas Inteligentes
Práctica 1: Desarrollo de agentes basado en técnicas de búsqueda heurística dentro del entorno GVGAI

David Muñoz Sánchez
07256819C - Grupo A1

16 de abril de 2023

Índice

1. Ejercicios propuestos

1.1. Ejercicio 1

Describa un caso para el que sería más recomendable/beneficioso utilizar el algoritmo de Dijkstra en lugar de A*, y viceversa.

El algoritmo de Dijkstra es una excelente opción para encontrar la ruta más corta en un grafo ponderado, donde todas las aristas tienen pesos no negativos. Este algoritmo explora todos los nodos en el grafo, lo que lo hace eficaz para grafos con un número relativamente pequeño de nodos y aristas.

En contraste, A* es un algoritmo heurístico que combina la búsqueda primero el mejor con una heurística que estima la distancia restante hasta el objetivo. A* es más eficiente que Dijkstra en grafos grandes, ya que se enfoca en los nodos más prometedores en lugar de explorar todos los nodos.

Entonces, ¿cuándo se recomienda usar uno u otro algoritmo? Si enfocamos la pregunta a esta práctica, donde contamos con una heurística admisible para A* como lo es la **distancia Manhattan**, es más conveniente usar la búsqueda heurística antes que Dijkstra. En general, A* se centrará en los nodos más prometedores y realizará menos expansiones que Dijkstra, además de asegurarnos el camino mínimo dada la heurística admisible que se usa.

Ahora bien, si nos encontramos en un problema para el cual no se cuenta con una heurística admisible (es decir, A* no garantiza encontrar el camino óptimo), y los mapas no son demasiado grandes, conviene usar Dijkstra, que no usa información heurística para hallar el camino más corto del grafo. Dado que Dijkstra tiene tiempos de ejecución más altos que A*, conviene no usarlo con mapas muy grandes, ya que probablemente tarde en encontrar la solución.

1.2. Ejercicio 2

¿Cuáles son las principales diferencias entre RTA* y LRTA*? En la práctica, ¿en qué afectan estas diferencias a la resolución del problema?

Ambos algoritmos son heurísticos en tiempo real. Para tomar una decisión, se basan en el espacio local de búsqueda, que en el caso de la práctica, son los vecinos del nodo en el que se encuentre la ejecución actualmente. La principal diferencia entre ambos algoritmos radica en la forma de actualizar el valor heurístico h de un nodo. Mientras que RTA* usa el segundo mínimo para la actualización de la heurística en el espacio local de aprendizaje (que en nuestro caso es el nodo actual), LRTA* usa el primero.

Como se puede observar en la tabla de resultados que hay más adelante en este documento, ni uno ni otro logran conseguir el camino mínimo en una sola ejecución, pero si que es cierto que RTA* converge a una solución mucho más rápido de lo que lo hace LRTA*, que incluso no consigue una solución en los mapas grandes de ambos juegos porque se pasa de 10000 ticks.

Entonces, podemos afirmar que RTA* es mucho mejor para una sola ejecución, ya que la sobreestimación de h eligiendo el segundo mínimo hace que escape de mínimos locales más rápido de lo que lo hace LRTA*, que si se visualiza una ejecución, se puede observar como da más vueltas por las mismas casillas, mientras que RTA* sale rápido de esas zonas. No obstante, LRTA* en varias ejecuciones y, teniendo en cuenta que cada ejecución toma la matriz de h's anterior, converge a la solución óptima, por lo que, a la larga, obtiene mejores resultados que RTA*, por la no sobreestimación de la función heurística.

1.3. Ejercicio 3

¿Cómo aplicaría el algoritmo de Dijkstra y el algoritmo A* para la resolución del juego "Labyrinth extended"?

Tanto Dijkstra como A* son dos métodos de búsqueda offline. Según se ha planteado en la práctica, estos algoritmos se ejecutan, obtienen un camino, y a partir de ahí nuestro agente hace los movimientos que se indican según la ruta obtenida. El agente siempre llega a su destino por el hecho de que tiene un conocimiento perfecto del entorno y de que este no cambia durante la ejecución, además de que siempre es posible hacerlo.

Sin embargo, el juego "Labyrinth extended" sí que puede cambiar durante la ejecución si el agente pasa por una de las casillas invisibles que hay en el mapa. Es por ello que este juego se prueba con algoritmos en tiempo real

que calculan una acción cada vez que se ejecutan, no calculan una ruta completa.

Para aplicar Dijkstra o A* a este juego, habría que tener en cuenta que muy probablemente, la ruta que se calcule al principio no funcione correctamente porque el agente se encuentre con un muro o una trampa a la hora de ejecutar el siguiente movimiento. Para controlar este hecho, habría que tener un método que cuando tengamos un plan, compruebe si el siguiente movimiento nos deja en una casilla no transitable. Si esto ocurre, se borra el plan actual, la variable booleana HayPlan se pone a False y se vuelve a recalcular una ruta llamando al algoritmo en cuestión, pero esta vez como casilla de inicio donde nos hemos quedado en la ruta anterior. Para ello, se vacían todas las listas, tanto de no visitados en el caso de Dijkstra como de Cerrados y Abiertos en A* y se vuelve a ejecutar de 0. Esto justamente ha sido lo implementado en la primera versión de mi agente de competición.

También hay más formas de proceder, como por ejemplo, la versión final del Agente de Competición que se ha explicado más adelante. En mi caso, está pensado para A*, pero con Dijkstra se hace de forma análoga. Justo al principio del *act*, se comparan los muros y las trampas con los muros y las trampas de la ejecución anterior que se guardan en una variable de clase. Si alguno de ellos difiere o ambos difieren con respecto a la ejecución anterior, se recalcula directamente. Esto se ha pensado para ahorrar los ticks que costaba, al menos en los mapas de prueba, recorrer la ruta que tuviéramos hasta no poder ejecutarla más para recalcular.

1.4. Ejercicio 4

Realice un script que, dado un mapa, genere uno nuevo posicionando al avatar en una localización distinta (siempre que en ésta no haya muros, trampas o casillas objetivo). Usando este script, ejecute el algoritmo RTA* múltiples veces sobre el mapa pequeño de labyrinth (original), de forma que la función h(n) sea inicializada con los valores almacenados al final de la ejecución anterior. Finalmente, represente usando un mapa de calor (heatmap), el valor de h(n) al finalizar la última ejecución de RTA*. Genere un segundo heatmap repitiendo el proceso con LRTA*. Por último, calcule el coste mínimo h*(n) para todas las casillas de este mapa usando A* (puede usar el script anterior) y represente esta función en un tercer heatmap. ¿En qué se diferencian los heatmaps obtenidos? ¿Qué conclusión se puede sacar de estos resultados?

A continuación, se presentan los tres heatmaps pedidos para el ejercicio:

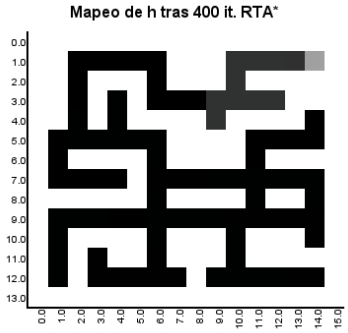

Figura 1: Experimento RTA*

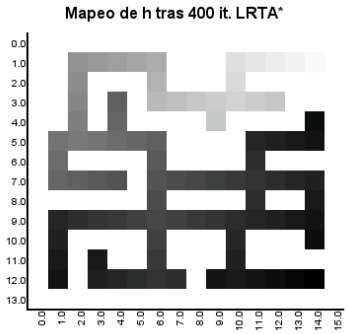

Figura 2: Experimento LRTA*

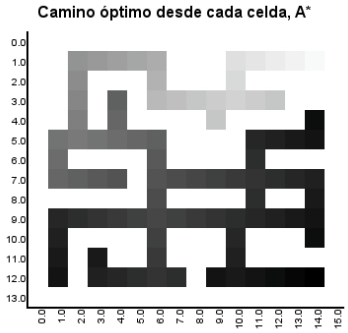

Figura 3: Coste óptimo A*

```
PARA LRTA*:
Valor de h máximo graficado: 39
PARA RTA*:
Valor de h máximo graficado: 3392
PARA A*:
Valor de h* máximo graficado: 39
```

Figura 4: Valores máximos graficados en cada algoritmo (lo que se representa con mayor intensidad de negro)

Como bien se puede observar tanto en la figura 4 como en las figuras 2 y 3, las múltiples ejecuciones de LRTA* convergen al coste mínimo h* que se ha obtenido con la longitud del plan que devuelve A* desde cada casilla de juego transitable (que sabemos que es el coste óptimo al usar una heurística admisible). RTA* no da buenos resultados por lo comentado anteriormente, al usar el segundo mínimo para la actualización de la h, sobreestima esta, haciendo que para una ejecución encuentre una solución más rápido, pero para varias no nos proporcione información valiosa. LRTA* sin embargo, acaba dando muy buenos resultados tras las 400 iteraciones que se le han asignado.

En este caso, como la clase HeatChart (de un repositorio de GitHub) que he usado para los HeatMap, no me daba la opción de poner una leyenda con los distintos valores de gris, pero sabiendo lo que dice la figura 4 y que a ese valor se le asigna el máximo valor de negro, se puede intuir por donde van los disintos valores, ya que se usa una división lineal entre el mínimo representado y el máximo para asignarlos. Esta cuestión ha tenido más importancia en la figura 1, ya que los valores de h de RTA* son muchos más altos y los muros se están representando con -1, la meta al tener h = 0, no se visualizaba, así que en este caso concreto, los muros se han puesto a -2000. Añadir también que probablemente, otra ejecución cambie los valores máximos de RTA*, ya que no se usa semilla aleatoria.

La forma de proceder para realizar los experimentos en RTA* y LRTA* ha sido la siguiente:

1. Se usa una función para cambiar de posición la A del mapa que está en nuestro mismo directorio.

2. Se llama a esta función y a la de jugar propia de GVGAI las veces que se desee.

3. En cada ejecución se va guardando y actualizando un fichero de texto con el mapeo de la función h, que servirá de realimentación para la siguiente ejecución.

En el caso de A* ha sido parecido, pero el mapeo del coste óptimo no realimenta el algoritmo, si no que solo se va actualizando al final de este. Además, la casilla de inicio no es random, si no que se va recorriendo todo el mapa hasta que no se pueda cambiar a ninguna casilla transitable nueva.

Por último, para hacer los heatmaps, se ha usado una clase tomada de GitHub y ha habido que tener en cuenta la orientación de los mapeos, ya que en cada algoritmo, la x representa las filas de la matriz de h's y la y las columnas, ídem con los valores guardados de A*. Esto es, justo al contrario del juego, así que para el heatmap, cada fila de la matriz que se tiene es la columna de la matriz que graficamos

2. Resultados de los algoritmos

A continuación, se presenta la tabla pedida con los resultados de las distintas ejecuciones del algoritmo en los distintos mapas proporcionados en Prado.

Algoritmo	Mapa	Runtime (acumulado)	Tamaño de la ruta calculada	Nodos expandidos
Labyrinth (original)				
Dijkstra	**Pequeño**	1.6434	36	88
	Mediano	8.7046	114	565
	Grande	12.7729	808	2144
A*	**Pequeño**	0.4465	36	83
	Mediano	1.4339	114	556
	Grande	2.2569	808	2127
RTA*	**Pequeño**	0.9447	40	40
	Mediano	4.3347	338	338
	Grande	24.6693	3386	3386
LRTA*	**Pequeño**	1.1981	60	60
	Mediano	19.3965	5658	5658
	Grande	>10000 TICKS	>10000 TICKS	>10000 TICKS
Labyrinth extended				
RTA*	**Pequeño**	1.7099	85	85
	Mediano	4.5165	338	338
	Grande	23.0187	3402	3402
LRTA*	**Pequeño**	2.4985	151	151
	Mediano	23.6618	5658	5658
	Grande	>10000 TICKS	>10000 TICKS	>10000 TICKS

Tabla 1: Resultados Dijkstra, A*, RTA* y LRTA*

3. Resumen algoritmos

A continuación, se hará un resumen de lo más reseñable de cada algoritmo así como de algunos problemas derivados de su implementación.

Todos los algoritmos se han apoyado en una matriz inicialmente con valores a infinito (que al ser una matriz de enteros, infinito es Integer.MAX_VALUE). Dependiendo del algoritmo la matriz contiene una cosa u otra, que en el caso de Dijkstra y A* sería la g actualizada del nodo en cuestión y en el caso de los algoritmos en tiempo real, la h. Nótese que la matriz tiene las mismas dimensiones que el grid de juego.

En cuanto a la implementación, se ha hecho siguiendo lo más fielmente posible los distintos pseudocódigos de los que se disponían, y todo está debidamente comentado en las distintas clases java. Además, el proyecto se ha apoyado en una clase nodo con el fin de hacer más entendible la estructura de los algoritmos y de encapsular debidamente la información. No obstante, en el algoritmo A*, no se ha usado la parte del pseudocódigo de uno de los ifs para ver que hacer con cada nodo. Se recomendaba comprobar que el nodo no estuviera ni en cerrados ni abiertos para actualizar su g, es decir, un filtro para saber que ese nodo hijo generado nunca había sido visitado. No obstante, en un principio se optó por comprobar que la casilla correspondiente al nodo en la matriz de g tenía un valor de infinito. En esencia, es la misma comprobación, pero con una complejidad menor, ya que un *contains* en Java es O(n) y una consulta a una matriz es O(1).

Por último, se va a comentar uno de los mayores problemas que surgieron mientras se implementaban los distintos algoritmos, que es la ordenación de la cola con prioridad con un comparador. Si bien es cierto que claramente, los dos primeros criterios para el orden son la f y posteriormente la g, tenía más dudas con el criterio histórico y de expansión de los nodos. Nótese, antes de hablar de este hecho, que por la naturaleza de los algoritmos en tiempo real, se puede obviar la comparación de la cola de sucesores por su g, ya que esta en constante y solo se necesita la f, que en este caso es la h. No obstante, en cuanto a la actualización de la h se refiere, si que hay que tener en cuenta el costo g, aunque sea constante.

Volviendo al problema con la expansión de los nodos, inicialmente, el orden de expansión no importaba y el criterio histórico se trataba asignando a cada hijo de un nodo una variable numérica con el mismo valor. En caso de empate (todos los hijos de un nodo siempre empataban), se desempataba según la expansión, para lo cual se contaba con un método que a cada nodo asignaba un valor numérico según la acción por la que fueron generados. Aunque esta forma de proceder no presentaba ningún problema, si que era un poco más liosa y menos transparente que la que finalmente se ha implementado. En la última versión, no es neceario desempatar según la acción porque los hijos de un nodo nunca van a tener la misma variable numérica, si no que tendrán una diferente lo cual, además de propiciar

un orden histórico, propicia que por la propia construcción del algoritmo (más concretamente, el momento en el que se generan los nodos), ya se tenga en cuenta el orden de expansión.

A continuación, se incluye una captura que ilustra el comportamiento de la cola con prioridad en Dijkstra y en el mapa pequeño que se proporciona:

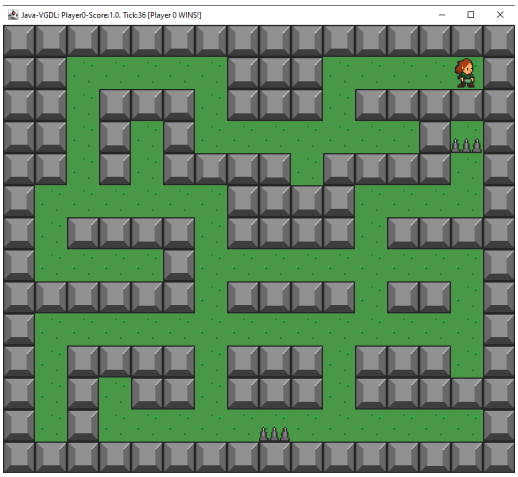

Figura 5: Mapa pequeño proporcionado

El agente empieza en la única casilla libre en la esquina inferior izquierda del mapa, por lo que está en una situación donde, si se imprime la cola con prioridad mostrando los diferentes atributos de un nodo, se ilustra a la perfección como se tiene en cuenta el criterio histórico y de expansión de nodos.

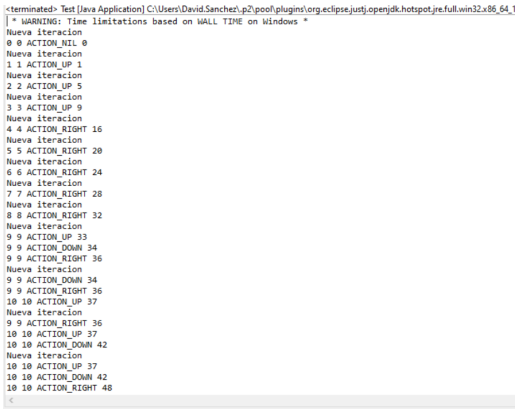

Figura 6: Datos contenidos en la cola con prioridad en cada iteración del algoritmo de Dijkstra

Lo que interesa es la acción y el número que sale a la derecha, que es la variable contador. Como se puede observar, entre una acción y su siguiente del mismo tipo, hay 4 unidades de diferencia. También hay que observar que, por ejemplo, entre la última acción UP y la siguiente RIGHT, hay más diferencia, puesto que la acción RIGHT que se hubiera generado al mismo tiempo que la acción UP mencionada (es decir, con un mismo padre), hubiera tenido contador con valor 12, por lo que la siguiente acción RIGHT, que es la última que se genera, tiene contador con valor 16.

De esta forma, los nodos de un mismo padre están ordenados y los nodos más antiguos tienen un contador menor que los más nuevos.

4. Agente de competición

Para el agente de competición, se ha implementado una modificación del algoritmo A* de forma que pueda ejecutarse en un entorno cambiante como lo es Labyrinth Extended.

La primera versión que se implementó, tenía en cuenta que dada una ruta para llegar al objetivo, había que comprobar siempre si la siguiente casilla era transitable, y, en caso contrario, recalcular la ruta al objetivo tomando la casilla donde nos quedamos como nodo inicial de partida para nuestro algoritmo.

Lo único que cambia con respecto a la versión primera que se hizo de A* es cuando tenemos un plan. Si tras comprobar vemos que el siguiente movimiento a ejecutar nos lleva a una casilla que no es transitable (ya sea porque es un muro o una trampa), se borra el plan entero, se vuelve a inicializar la matriz de costes g todo a infinito, se indica que no hay plan con la variable booleana que se tiene para comprobar este hecho y se devuelve la acción nula. A partir de ahí, se vuelve a calcular una ruta esta vez siendo la casilla inicial donde se quedó el agente en la ejecución de la ruta anterior.

La versión definitiva mantiene esa filosofía pero es un poco diferente. El hecho de recalcular ya no se hace porque el plan actual nos mueva a una casilla intransitable, si no que se recalcula cuando en el mapa se produce un cambio en las trampas o en los muros, es decir, cuando se activa una casilla invisible propia del juego extendido. Si se añaden muros o trampas o ambos, directamente el agente recalculará la ruta. Si al pasar por una casilla no pasa nada (podría darse el caso), sirve la ruta que se había calculado inicialmente.

Esto es fácil de implementar ya que se cuenta con una variable de clase donde se guarda la lista de posiciones *Immovable*, que se actualiza al final del método *act* justo antes de devolver una acción. Ahora, antes de entrar a valorar en el método si tenemos o no plan para hacer una cosa u otra, se ve si las posiciones *Immovables* actuales difieren con la que tenemos en la variable y, si es así, se recalcula, si no, el método continúa.

No tenemos conocimiento de donde están las casillas invisibles, pero si tenemos conocimiento completo de nuestro entorno, así que podemos evaluar si ha habido cambio con respecto a mapas anteriores.

En resumen, si el agente detecta en alguna ejecución del método *act* que las trampas o los muros han cambiado con respecto a la ejecución anterior, recalcula tomando como inicio la casilla donde esté en ese momento del cambio. Esto, muy probablemente, pase siempre que el agente atraviese una casilla invisible que cambie el mapa, aunque no tiene por qué, podría ocurrir que no cambiara nada y el agente podría continuar con la ruta que ya tenía.

A continuación, se muestra una tabla con los resultados de ejecutar el agente en los distintos mapas proporcionados de Labyrinth Extended:

Agente Competición	
Mapa	**Ticks**
Pequeño	58
Mediano	215
Grande	1049

Tabla 2: Timesteps gastados en competición

4.1. Problemas con la implementación

Tal y como se ha comentado en la sección 3 de este documento, desde un principio se optó por comprobar si un nodo no había sido visitado de la forma descrita. No obstante, tras comprobar que acción devolvía el plan de A* modificado, se comprobó que justo siempre al recalcular devolvía la acción nula. En un principio se pensó que era por el hecho de que el método act tardaba más de 40ms, y se procedio a cambiar el código para que hiciera exactamente lo que pone el pseudocódigo de las transparencias. No obstante, esto daba peores resultados aún. Al final todo era que al terminar de calcular una ruta no devolvía ninguna acción y por defecto se devuelve la nula, así que se solucionó rápidamente.