Práctica 1: Desarrollo de agentes basado en técnicas de búsqueda heurística dentro del entorno GVGAI

UNIVERSIDAD DE GRANADA

Técnicas de los Sistemas Inteligentes

Elena Torres Fernández
Curso académico 2024-2025

11 abril 2025

Tabla de resultados

A continuación se muestra una tabla con los resultados obtenidos para cada ejecución de los cuatro algoritmos (Dijkstra, A*, RTA* y LRTA*) en los tres mapas de prueba correspondientes. El mapa pequeño es *"labyrinthdual_lvl0.txt"*; el mediano, *"labyrinthdual_lvl5.txt"*; y el grande, *"labyrinthdual_lvl6.txt"*. Cabe destacar que he usado ChatGPT para que dada una imagen de la tabla vacía de los resultados, me proporcione el código en LaTeX necesario para crearla con el mismo formato. A excepción de algunos ajustes posteriores, me sirvió de gran ayuda.

Algoritmo	**Mapa**	**Runtime ms (acumulado)**	**Tamaño ruta calculada**	**Nodos expandidos**	**Nodos Abtos/ Cerrados**
Labyrinth Dual (id 59)					
Dijkstra	Pequeño	7	68	157	
	Mediano	13	60	337	
	Grande(*)	63	184	4202	
A*	Pequeño	8	68	156	1/156
	Mediano	12	60	244	9/244
	Grande	36	184	1628	44/1628
RTA*	Pequeño	6413	242	242	
	Mediano	9204	350	350	
	Grande	26724	1014	1014	
LRTA*	Pequeño	34017	1330	1330	
	Mediano	47651	1858	1858	
	Grande	24598	932	932	

Cuadro 1: Tabla de resultados

(*) En la ejecución del algoritmo de Dijkstra en el mapa Grande obtengo el siguiente mensaje de error por tiempo excedido:

```
$  cd /home/usuario/Escritorio/TSI/practica1/GVGAI-master ; /usr/bin/env /usr/lib/
    jvm/java-21-openjdk-amd64/bin/java @/tmp/cp_co2nlp23t8rcyfddnqnedgajf.argfile
    tracks.singlePlayer.Test
Runtime Dijkstra (ms): 63
Tamano de la ruta calculada: 184
Numero de nodos expandidos: 4202
Too long: 0(exceeding 26ms): controller disqualified.
Result (1->win; 0->lose): Player0:-100, Player0-Score:-1000.0, timesteps:0
```

Preguntas

1. Imaginemos que la representación del estado en nuestro juego (*Labyrinth Dual*) solo incluyese la posición de la casilla en que nos encontramos (coordenadas X e Y del *grid*). Por ejemplo, no se incluiría la capa como parte del estado. ¿Qué problemas y/o ventajas podría tener esta representación?

   Al considerar sólo la posición del avatar, el algoritmo **está tratando nodos que serían diferentes como iguales**, pues no es lo mismo estar en una casilla teniendo una capa azul

puesta que estar en esa misma casilla sin nada puesto. Esto influye directamente en el plan que se calcula porque se está perdiendo información esencial al no tener en cuenta el consumo de recursos y las capas que quedan disponibles. Sí llegaría a la meta en un camino óptimo cuando no se necesiten capas para llegar a la misma o cuando el camino óptimo pase por una sola capa y no haya más capas en el mapa. Sin embargo, podría no llegar a la meta si se necesitara una capa específica para cruzar o si el número y la ubicación de las capas restantes afectara a la ruta futura (las capas desaparecen una vez cogidas).

Como **ventajas**, al usar solo la posición como parte del nodo estamos gastando menos memoria por cada nodo, las comparaciones entre nodos son más rápidas (comparar vectores es más rápido que comparar estructuras más complejas) y el algoritmo es más simple de implementar. No obstante, no sirven de nada estas ventajas porque **el algoritmo sería incompleto con esta configuración de nodos**; no siempre alcanzaría la meta, aunque fuera posible.

2. ¿RTA* y LRTA* son capaces de alcanzar el portal en todos los mapas? Si la respuesta es que no, ¿a qué se puede deber esto?

   En nuestros mapas sí lo alcanzan pero **no está garantizado que siempre lo hagan**. Al ser algoritmos en tiempo real podrían quedarse **atrapados en mínimos locales** y nunca llegar a la meta, ocasionando ciclos o bloqueos. Esto podría darse en casos donde sea necesario alejarse temporalmente de la meta o tomar un camino largo que parezca peor al principio, a falta de tener una visión global del mapa. Además, para mapas muy complejos estos algoritmos podrían tardar un tiempo excesivo y no llegar a la solución por falta de memoria o tiempo de cómputo.

3. Imaginemos que debemos implementar búsqueda en profundidad como algoritmo de búsqueda para resolver este u otro problema. La forma más evidente de hacerlo es partir de una implementación de búsqueda en anchura y sustituir por una pila la cola que se emplee para almacenar los nodos abiertos. ¿Qué otra alternativa tendríamos a esta estrategia naive, y qué pros y contras tendría dicha alternativa?

   Lo ideal sería usar **llamadas recursivas al algoritmo de búsqueda en profundidad**. En cuanto a las ventajas, esta implementación conlleva un código más compacto y legible, no hace falta gestionar manualmente ninguna cola o pila de nodos abiertos y se alinea con la naturaleza recursiva del problema.

   Como desventajas, podemos destacar el posible desbordamiento de la pila porque se supere el límite de recursión y la **menor trazabilidad o depuración del código**. En cuanto a lo segundo, siendo todo llamadas anidadas es más difícil pausar la ejecución y ver qué hay en la "pila"de búsqueda de cara a hacer un seguimiento de los nodos visitados. Por otro lado, el problema del límite de recursión está ligado al límite de profundidad que se establece para el algoritmo de búsqueda en profundidad, necesario en espacios de búsqueda grandes. Al poner este límite, es posible que no se alcance la meta o que lo haga de forma no óptima para algunos casos. Este es un problema existente en la búsqueda en profundidad en general.

4. ¿De qué modo LRTA*(k) se diferencia de LRTA*? ¿En qué consiste el algoritmo y cuáles son sus pros y contras en relación a algoritmos previos (como LRTA*)?

   En primer lugar, el **algoritmo LRTA*(k)** se basa en la misma idea que el LRTA* pero

actualizando la estimación de la heurística hasta k veces, no necesariamente de nodos distintos. Dicho de otra forma, el algoritmo LRTA* actualiza la estimación de la heurística para un único nodo en cada iteración; mientras que el LRTA*(k) lo hace hasta k veces por iteración, siguiendo una **estrategia de propagación acotada**. De hecho, LRTA* es un caso particular de LRTA*(k) para $k = 1$. Esta actualización mantiene la admisibilidad de la heurística.

Algunas **mejoras** del algoritmo LRTA*(k) frente a LRTA* son las que se mencionan a continuación. Por un lado, se ha probado que el primero **converge más rápido** a la solución porque sus estimaciones del valor de la heurística para cada nodo se acercan más a los valores exactos. Por otro lado, el algoritmo LRTA*(k) **aprovecha mejor la regla de aprendizaje del primer mínimo**. Si el primer mínimo no involucra ciclos, LRTA*(k) podría comportarse como LRTA*, pero no suele ocurrir. Así, cuando LRTA*(k), dentro de una misma iteración, revisita un nodo y encuentra una mejor estimación de la heurística para el mismo, esta se actualiza y se selecciona una mejor acción, la cual no hubiera sido encontrada por el algoritmo LRTA* porque se hubiera quedado con la primera. De esta forma, LRTA*(k) **encuentra soluciones más cortas** y con menor tiempo computacional.

Como **desventaja** del algoritmo LRTA*(k) frente a LRTA* está el tiempo de cálculo del plan en cada iteración. Aunque a nivel global el primero proporciona un camino más corto que el segundo, **el segundo tiene un menor tiempo de respuesta en cada iteración**, pues sólo actualiza la heurística de un nodo. Esto puede ser beneficioso para aplicaciones en las que sea necesario tener una respuesta rápida en cada paso y no tanto el que la solución sea la más corta posible. No obstante, el tiempo de planificación en LRTA*(k) puede adaptarse según el valor de k. A mayor k, obtenemos una mejor solución (camino más corto) pero con mayor tiempo de planificación en cada iteración y viceversa.

5. Dentro del pseudocódigo del algoritmo A*, tenemos el siguiente bloque de código. ¿Bajo qué condiciones podemos estar seguros de que no vamos a entrar nunca dentro de este *if*? En otras palabras, ¿es siempre necesario implementar esa parte del algoritmo?

```
1     if ( cerrados.contains(sucesor)
2               and mejorCaminoA(sucesor) ): // menor g(n)
3          cerrados.remove(sucesor)
4          abiertos.add(sucesor) // actualizar g(n)
5
```

Este *if* comprueba si ya tenemos un nodo en cerrados para el cual hemos encontrado un sucesor con menor coste. En ese caso, se actualizan los valores del nodo mejor en cerrados. Podemos estar seguros de que no vamos a entrar nunca a este *if* si la **heurística es monótona**. En este caso, A* nunca encontrará un camino con menor coste hacia un nodo que ya esté en cerrados, por lo que no hace falta revisar en cerrados.

Una heurística es monótona si cumple la siguiente condición:

$$h(n) \leq c(n, s) + h(s),$$

donde $h(n)$ es el valor heurístico del nodo n; s es un sucesor de n y $c(n, s)$ es el coste de ir de n a s (en nuestro caso, siempre 1). Como nuestra heurística es la distancia de Manhattan, que sabemos que es monótona, nunca entramos en ese *if*. De hecho, no he incluido ese bloque en la implementación.

Yendo más allá, **monotonía implica admisibilidad**, luego la heurística de Manhattan también es admisible y podemos decir que esta implementación del algoritmo A* es **óptima y completa**, es decir, si existe solución, siempre la encuentra y será la óptima.