Práctica 1

Técnicas de los Sistemas Inteligentes
Universidad de Granada
Carmen Azorín Martí

12 de abril de 2025

Algoritmo	**Mapa**	**Runtime (acumulado)**	**Tamaño de la ruta calculada**	**Nodos expandidos**	**Nodos Abiertos / Cerrados**
Labyrinth Dual (id 59)					
Dijkstra	Pequeño	10	68	157	
Dijkstra	Mediano	32	60	337	
Dijkstra	Grande	74	184	4202	
A*	Pequeño	9	68	156	1/ 155
A*	Mediano	25	60	240	9/ 243
A*	Grande	118	184	1628	44/ 1627
RTA*	Pequeño	27	242	242	
RTA*	Mediano	39	350	350	
RTA*	Grande	83	1014	1014	
LRTA*	Pequeño	73	1330	1330	
LRTA*	Mediano	121	1858	1858	
LRTA*	Grande	75	932	932	

Cuadro 1: Comparativa de algoritmos en Labyrinth Dual (id 59)

Pregunta 1

Si solo usamos la posición (las coordenadas X e Y) como estado, sin tener en cuenta la capa que llevamos ni las capas que quedan por recoger, nos estamos dejando fuera información muy importante. El principal problema es que en Labyrinth Dual hay muros que solo se pueden pasar con una capa concreta (azul o roja), y si no sabemos qué capa llevamos, el agente no va a saber si puede pasar o no por ahí.

Por ejemplo, puede que la única forma de llegar a la meta sea pasando por un muro azul, pero como no guardamos si llevamos la capa azul, el algoritmo simplemente no va a poder planear ese camino, aunque sea el mejor (o el único) posible. Incluso si hay otro camino sin muros, puede que sea mucho más largo, y el agente lo elija solo porque no entiende que puede usar capas para atajar.

También está el tema de las capas que quedan por recoger. Dos estados con la misma posición y misma capa actual podrían parecer iguales, pero en realidad no lo son si uno ya ha recogido todas las capas posibles y el otro todavía tiene opciones de coger otra. Eso puede marcar la diferencia entre quedarse atascado o poder seguir avanzando.

Entonces, solo la posición como estado puede hacer que el agente no encuentre soluciones o tome decisiones poco eficientes, simplemente porque le falta contexto sobre su situación real en el laberinto.

Una posible ventaja de solo guardar la posición como estado es que se usa menos memoria, ya que no se está guardando información extra como la capa actual o las que quedan por recoger. Esto hace que el número total de estados posibles sea mucho menor, así que el algoritmo puede ir más rápido al explorar, porque tiene menos combinaciones que considerar.

También podría hacer que la implementación sea un poco más simple, ya que no hace falta preocuparse por actualizar el estado de las capas en cada movimiento. Pero claro, esa simplicidad tiene un precio, porque se pierde información clave y el agente puede tomar malas decisiones o directamente no encontrar la solución.

Pregunta 2

No, RTA* y LRTA* no siempre son capaces de llegar al portal en todos los mapas. Un ejemplo claro de esto es el mapa 4 del juego, donde se pueden quedar atrapados.

Esto pasa porque estos algoritmos toman decisiones basadas en información local (lo que ven justo en ese momento), sin planificar todo el camino como lo haría A* o Dijkstra. Entonces, puede que entren en una zona atravesando un muro marrón, usando la capa roja, y después cojan la capa azul. El problema es que al hacer eso, ya no pueden volver por el mismo muro marrón, porque la capa roja ya la usaron y no se puede recuperar. Quedan atrapados sin salida.

En esos casos, al no tener una visión completa del mapa ni poder "pensar a largo plazo", RTA* y LRTA* pueden meterse en caminos sin retorno, y se quedan atascados aunque haya una solución si hubieran tomado otras decisiones antes.

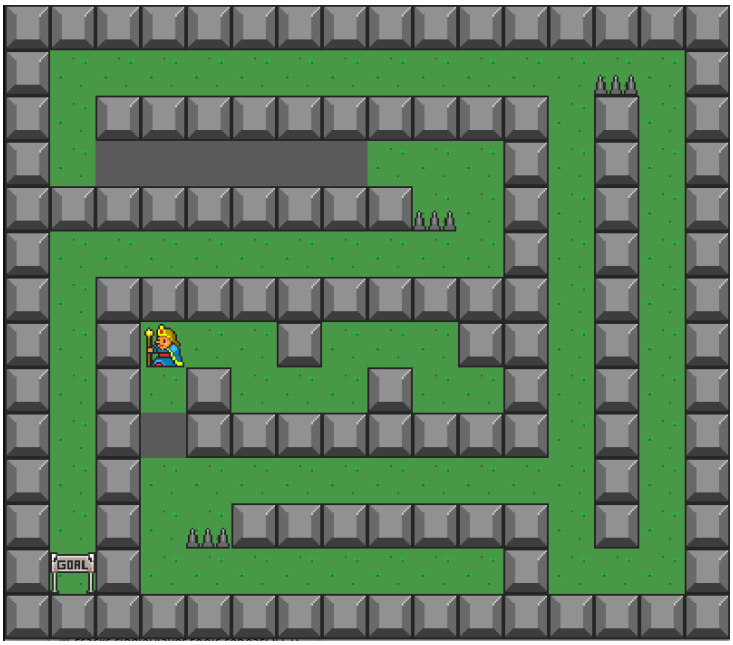

Figura 1: Mapa 4 del juego Labyrinth Dual con RTA*

Vemos en la imagen que el agente tiene la capa azul y está rodeado de muros grises y un muro rojo que ya no puede traspasar. Ya no podrá llegar al portal.

Pregunta 3

Además de cambiar la cola por una pila para hacer búsqueda en profundidad, otra forma sería implementarla usando recursividad. Es decir, dejar que el programa cree una especie de pila automáticamente mediante llamadas recursivas a funciones. De esta forma, la pila de llamadas del sistema se usa para guardar los nodos abiertos.

Las ventajas de hacerlo así son que el código suele quedar más limpio y fácil de leer, sobre todo si el problema no es muy profundo. Además, en problemas que ya tienen una estructura recursiva, como laberintos o árboles, la recursividad encaja de forma muy natural.

Sin embargo, si el espacio de búsqueda es muy profundo, usar recursión puede consumir mucha memoria y provocar errores de desbordamiento de pila, ya que cada llamada ocupa espacio en la memoria. También puede ser más difícil de controlar si necesitas llevar mucha información adicional en cada nodo, porque cada llamada puede implicar muchos parámetros y es más fácil cometer errores al actualizar o restaurar esos datos.

Pregunta 4

LRTA*(k) es una mejora del algoritmo LRTA* que introduce una estrategia más avanzada para actualizar las heurísticas. Mientras que LRTA* solo actualiza la heurística del estado actual en cada iteración, LRTA*(k) puede actualizar hasta k estados, lo que le permite propagar mejor la información aprendida. Esta técnica se llama propagación acotada, y

permite que el algoritmo aprenda estimaciones heurísticas más precisas y de forma más rápida, manteniendo la propiedad de admisibilidad de la heurística.

Gracias a esta mejora, LRTA*(k) suele encontrar mejores soluciones desde el primer intento y converge más rápido a caminos óptimos, con menos pasos y menos repeticiones que LRTA*. Además, sus soluciones son más estables. La principal desventaja es que requiere más cálculos en cada paso, lo cual puede ser un problema si se necesita una respuesta muy rápida. Sin embargo, como el valor de k es configurable, se puede ajustar el equilibrio entre calidad de solución y coste computacional.

Pregunta 5

Ese bloque del algoritmo A* se encarga de detectar si hemos encontrado un camino mejor a un nodo que ya habíamos cerrado. Si eso ocurre, reabrimos ese nodo para reconsiderarlo.

Una heurística es monótona cuando para cada nodo n y n', se cumple que: $h(n) \leq c(n, n') + h(n')$ donde $c(n, n')$ es el coste de ir de n a n'. Si estamos usando una heurística monótona, podemos estar seguros de que nunca entraremos en ese if, porque A* no encontrará caminos mejores a nodos ya cerrados.

En nuestro juego estamos usando la distancia Manhattan como función heurística, que es admisible (nunca sobreestima el coste real al objetivo) y monótona (cumple la propiedad de que $h(n) \leq c(n, n') + h(n')$, que es la propiedad triangular). Por tanto, A* no necesitaría reabrir nodos en este caso, y podríamos prescindir de ese bloque del if sin problemas.

Por el contrario, si se usara una heurística que no fuese consistente, sí podría ocurrir que se encontraran mejores caminos a nodos ya cerrados, y en ese caso sí sería necesario mantener ese if para garantizar que A* funcione correctamente y encuentre soluciones óptimas.