Práctica 1:
TSI

UNIVERSIDAD
DE GRANADA

Pablo Martín Palomino
DNI: 7593097H
E-mail: pablomarpa@correo.ugr.es

Tabla de resultados

Algoritmo	Mapa	Runtime (acumulado)	Tamaño de la ruta calculada	Nodos expandidos	Nodos Abiertos / Nodos Cerrados
Labyrinth Dual (id 59)					
Dijkstra	Pequeño	3 ms	68	157	
	Mediano	2 ms	60	337	
	Grande	16 ms	184	4202	
A*	Pequeño	6 ms	68	156	1/156
	Mediano	4 ms	60	244	9/244
	Grande	18 ms	184	1628	44/1628
RTA*	Pequeño	16 ms	242	242	
	Mediano	20 ms	350	350	
	Grande	41 ms	1014	1014	
LRTA*	Pequeño	41 ms	1330	1330	
	Mediano	57 ms	1858	1858	
	Grande	40 ms	932	932	

Apartado de preguntas

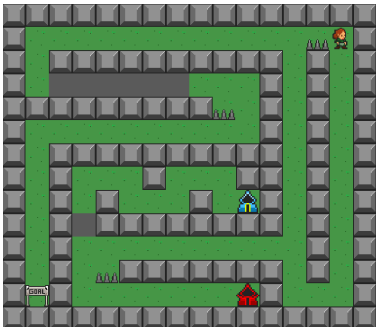
1) ***Imaginemos que la representación del estado en nuestro juego (Labyrinth Dual) solo incluyese la posición de la casilla en que nos encontramos (coordenadas X e Y del grid). Por ejemplo, no se incluiría la capa como parte del estado. ¿Qué problemas y/o ventajas podría tener esta representación?***

Ventajas: Principalmente en mapas que no haya obstáculos relacionados con tener capas se llegará a solución óptima igualmente (Básicamente si el mapa no tiene obstáculos relacionados con capas calcula la ruta como si fuera camino en un laberinto sin obstáculos). La carga en memoria será mucho menor pues no hay que almacenar la información de las capas.

Desventajas: En el momento que haya obstáculos relacionados con capas como en el estado no controlamos ni si tenemos una capa o no, ni posición y tipos de capas que quedan,la mayoría de veces no encontrará solución o si la encuentra seguramente no será una solución válida.¿A qué se debe? Pues, si hay un obstáculo sea “muro azul “ que se necesita pasar para llegar a meta , directamente no podemos saber cuando podemos pasarlo pues no controlamos que capa tenemos.

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?*

No, de hecho en el mapa 4 de la práctica no encuentra solución RTA, esto es debido a que una vez toman una decisión irreversible están condicionados, me explico puede ocurrir decidir coger una capa azul/roja porque era el mejor vecino en ese momento pero luego resulta que con la capa azul/roja me he quedado encerrado en un lugar del mapa impidiendo llegar a meta.Imagen que ejemplifica:



Si cogemos la capa roja y después entramos al centro del mapa y cogemos la capa azul nos quedamos encerrados y esto puede pasar en RTA y LRTA (en este caso solo ocurre con RTA).

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?*

Otra alternativa sería usar recursión donde la pila de llamadas a funciones manejaría los nodos por explorar.El pseudocódigo sería algo así (dfs=depth-first search):

```
dfs(nodo):
si nodo es solucion devolver nodo
para cada posible nodo hijo:
    resultado = dfs(hijo)
    si resultado devolver resultado
```

Ventajas

-Es un código muy legible y fácil entender
-No hay que gestionar manualmente una pila
-Bueno en problemas de Backtracking

Desventajas

-En grafos muy profundos puede haber desbordamiento de pila por la recursión (problema general de la recursividad)
-Dificultad para añadir restricciones en la búsqueda del tipo profundidad, reinicios etc.…
-Las llamadas recursivas pueden llegar a ser más lentas

4) *El alumnado debe revisar el siguiente artículo científico: Hernández, C., & Meseguer, P. (2005). LRTA*(k). In Proceedings of the 19th International Joint Conference on Artificial Intelligence (pp. 1238-1243).*

https://www.ijcai.org/Proceedings/05/Papers/0764.pdf ¿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*)?

Primero voy a notar que tomando k=1 son el mismo algoritmo.
Diferencias entre LRTA* y LRTA*(k)

-LRTA* actualiza la heurística de un solo estado por iteración mientras que LRTA* (k) actualiza las heurísticas de hasta k estados (pueden ser distintos o repetidos y deben haber sido previamente visitados) por iteración (utiliza estrategia de propagación acotada)

-Propagación, LRTA* no propaga más allá de su estado actual y LRTA* (k) como ya he explicado usa propagación acotada cambiando estados anteriores limitando a k el número de actualizaciones.

-LRTA* (k) usa supports que es básicamente si un estado sea 'a' es support de un estado 'b',este justifica el valor actual de h(b). Esto optimiza la propagación actualizando solo estados cuyo support ha cambiado.

En que consiste el algoritmo:

inicialización:

asigna a cada estado x valor heurístico.
registrar el camino recorrido desde el estado inicial

bucle:

LookHeadUpdateK(función del paper): realiza una búsqueda limitada en profundidad y actualiza heurísticas de hasta k estados en el camino propagando cambios hacia atrás

Se elige el mejor vecino (menor valor coste ir + heurística)

Se ejecuta la acción

convergencia:

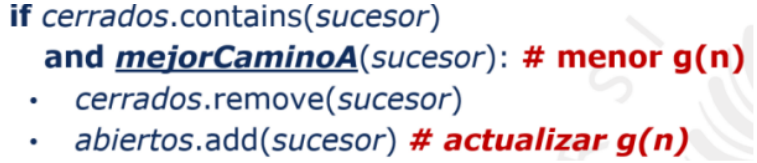
repetir hasta que heurísticas no cambien garantizando caminos óptimos si h es admisible

Ventajas y Desventajas:
-Mantiene las ventajas y desventajas de RTA y LRTA respecto A* y dijkstra (al final es variante modificada de LRTA)
-Mejor calidad en la primera solución encontrada respecto LRTA
-Convergencia más rápida a rutas óptimas que LRTA
-Las soluciones son más estables (consistentes con menos variabilidad)
-El parámetro k permite establecer equilibrio entre tiempo de planificación y calidad de solución
-Tiene mayor costo computacional por cada paso, especialmente con k alto (hay que propagar)
-Necesita mayor uso de memoria por el uso de supports

-Con respecto a otros algoritmos RTA* es mejor en primer intento pero no converge a rutas óptimas,FALCONS converge más lento y requiere más exploración inicial.

En resumen LRTA(K) mejora la calidad de soluciones y convergencia a cambio de un mayor costo computacional y de memoria, siendo este muy útil en problemas con heurísticas poco informadas.

5) ***En el pseudocódigo del algoritmo A* existe el siguiente bloque:***



¿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?

No se entrará al if en los siguientes casos:

-Si h(n) es una heurística consistente ($h(n) \leq c(n, n') + h(n')$ para nodo n y n' sucesor con c(n,n') el costo de moverse de n a n') nunca se ejecutará, pues una vez se saca un nodo de abiertos y se mete a cerrados ya se ha encontrado un camino óptimo hacia ese nodo y por lo tanto no hay mejor camino que él ya metido en cerrados.

-Si el grafo es un árbol sin ciclos ni caminos alternativos (cada nodo tiene un único padre luego no se puede llegar a un nodo de varias maneras).

No es siempre necesario implementar esa parte del algoritmo pues como se ha especificado hay casos donde no es necesario (Por ejemplo heurística consistente).

Respecto uso de ia

He usado principalmente de base el código que hice en IA el curso anterior para Dijkstra y A* pero he usado chatgpt y deepseek principalmente para :

-Problemas con el lenguaje de programación Java. Aquí incluyo declaración de distintas estructuras de datos en java y que los operadores en java se hacen distinto a C + +.

-Para mejorar eficiencia una vez tenía el código según indicaciones que me dieron por ejemplo el apply el switch poner primero el caso más frecuente, en RTA usar bucle para acciones en lugar de a mano (como hice en A* y dijkstra) o por ejemplo me dijeron que sustituyese este control de nodos en abiertos y cerrados:

```
if (!SolutionFound && !frontier.isEmpty()) {
   current_node = frontier.peek();
   while (!frontier.isEmpty() && explored.contains(current_node.estado)) {
      frontier.poll();
      if (!frontier.isEmpty()) {
         current_node = frontier.peek();
      }
   }
}
```

por esta línea porque era más eficiente y hace "lo mismo" realmente

```
if (explored.contains(current_node.estado))
continue
```