P2 | Los extraños mundos de BelKan

Alumno: **Miguel Ángel Fernández Gutiérrez**

Nivel 1. Agente deliberativo básico

Algunas consideraciones de optimización

En el primer nivel de la práctica se pide la implementación de un agente deliberativo, mediante la búsqueda de un plan para llegar a un objetivo según los algoritmos de búsqueda en anchura y en costo uniforme. Las primeras implementaciones de estos algoritmos están en las funciones `pathFinding_Anchura` y `pathFinding_CostoUniforme`; sin embargo, se tratan de implementaciones extremadamente ineficientes en memoria, especialmente por la estructura de los nodos, `nodo`:

`jugador.cpp`

```
245 struct nodo{
246         estado st;
247         list<Action> secuencia;
248 };
```

Obsérvese que, cada vez que creamos un hijo, copiamos toda la `secuencia` anterior y añadimos una nueva `Action`. Estas copias hacen que el uso de memoria se incremente exponencialmente conforme aumenta el tamaño del problema.

Esto ha sido solucionado en los métodos `pathFinding_Anchura2` y `pathFinding_CostoUniforme2`, que son los usados al llamar a `think` con los niveles 2 y 3, respectivamente. Esto produce una considerable mejora en los tiempos de ejecución respecto a los algoritmos no optimizados: usando el algoritmo no optimizado en mapas grandes, el ordenador no escapaz de calcular las rutas, mientras que el algoritmo optimizado calcula las rutas en menos de un segundo, como podemos ver a continuación:

Mapa	Anchura	Anchura Optimizado	Costo Uniforme	Costo Uniforme Optimizado
`mapa30`	0.138 s	0.004 s	0.089 s	0.015 s
`mapa50`	1.292 s	0.018 s	2.179 s	0.155 s
`mapa100`	NC	0.106 s	NC	0.477 s
`mapa100c`	NC	0.071s	NC	0.380 s

Donde indicamos con NC los que no ha logrado calcular en menos de 3 minutos.

La optimización consiste en no almacenar `secuencia`, sino usar un `unordered_map`, que llamamos `vector_de`, que nos servirá para saber quién es el padre de un cierto estado (junto con información adicional). Comentaremos brevemente las principales características de cada implementación.

Búsqueda en anchura

- Usamos una función auxiliar, `siguientes`, que, a partir de un cierto estado, devuelve un vector de pares {`estado`, `Action`}, correspondiente a todos los estados a los que se puede llegar desde el estado a expandir, junto a la acción para llegar a éste.
- Usamos una cola, `frontera`, para de este modo obtener el camino con menos acciones.

Búsqueda por costo uniforme

- En lugar de usar `estado`, puesto que hemos de tener en cuenta si hemos llegado a un cierto estado con o sin bikini y/o zapatillas, usamos `estadoBZ`, que almacena además esta información.
- La función auxiliar es `siguientes_costo`, que devuelve a partir de un cierto estado un vector de tuplas {`estado`, `Action`, `costo`}, al igual que en `siguientes`, pero donde además `costo` es el coste de realizar cada acción.
- En lugar de usar una cola para `frontera`, usamos una cola con prioridad, en el que insertamos cada estado con el coste de llegar desde el origen hasta éste.

Otras consideraciones

Para la implementación de los algoritmos anteriormente descritos ha sido necesaria la sobrecarga de ciertos operadores para los structs, así como la definición de funciones hash para éstos (pues es necesario para `unordered_map`). En el código se encuentran documentados todos los comentarios pertinentes, así como ciertos detalles de la implementación que no se explicarán en este documento.

Nivel 2. Agente reactivo/deliberativo

El agente reactivo/deliberativo entregado ha sido fruto de diversas modificaciones para mejorar su desempeño. A continuación detallaré los diversos aspectos que aparecen en el agente de una forma similar a la que las he desarrollado.

Algoritmo principal

El algoritmo principal usado es **A*** (usando los principios de optimización de memoria usados para los de búsqueda por anchura y costo uniforme), tomando como heurística la *distancia Manhattan*:[1]

$$d_{\text{Manhattan}}((\texttt{st1.f}, \texttt{st1.c}), (\texttt{st2.f}, \texttt{st2.c})) = |\texttt{st1.f} - \texttt{st2.f}| + |\texttt{st1.c} - \texttt{st2.c}|$$

La estructura básica es la siguiente:

1. Calcular un camino hasta el destino con A*.
2. Seguir el camino hasta el destino.
3. Si nos encontramos ante un obstáculo o actualizamos el mapa: volver a 1
4. Llegar al destino.

Donde en el paso 1 consideramos las casillas desconocidas como atravesables (en cada movimiento iremos actualizando `mapaResultado` con la información de los sensores del agente, véase el método `actualizarMapa`). De este modo, incitamos al agente a explorar el mapa. Sin embargo, pronto observamos diversas posibles mejoras, que iremos especificando a continuación.

Buscando en el código la palabra clave `MEJORA X`, con X siendo el número de la mejora, podrá leer los comentarios y la documentación detallada al respecto.

Mejora 1. Esquivando aldeanos

Primeramente vemos que nos encontramos con muchos aldeanos por nuestro camino. Para poder arreglar esto, recalcularemos el camino en caso de que tengamos un aldeano en frente y la siguiente acción sea `actFORWARD`. Basta incluir la posición del aldeano como obstáculo, y A* calculará adecuadamente el camino a seguir.

Mejora 2. Recargas

Pronto vemos que nuestro agente se queda sin batería, por lo que sería ideal recargarla. Sin embargo, saber cuándo y cuánto es complicado, principalmente por los siguientes factores:

- Si recargamos excesivamente, estamos perdiendo tiempo.
- Si queda poco tiempo, ir a recargar será contraproducente.
- Si tenemos más batería restante, deberíamos recargar menos.

[1] Hemos abreviado `fila` con `f` y `columna` con `c`. `st1` y `st2` son de tipo `estado`.

Por ello he adoptado un criterio que tiene como factores la cercanía a la recarga, la cercanía al objetivo y el tiempo restante. Para poder efectuar el concepto de "cercanía" hacemos de nuevo uso de la heurística. De este modo, haremos una recarga de acuerdo a los siguientes principios:

- Si la batería es inferior a `b_i`:
  - **Recargar inmediatamente** si se encuentra cerca de una recarga, en el sentido de que

    $$h(\text{actual}, \text{destino}) > k_D \cdot h(\text{actual}, \text{recarga})$$

    donde h es la heurística (distancia Manhattan) y k_D es una cierta constante (`K_DESTINO`).
- Si la batería está entre `b_i` y `b_s`, y nos queda un tiempo superior a `TIEMPO_RECARGA_INMEDIATA`:
  - **Recargar** si por el camino al objetivo el agente se encuentra "cerca" de una recarga, en el sentido de que

    $$h(\text{actual}, \text{recarga}) \leq k_C \cdot \text{tamaño mapa}$$

    donde k_C es una cierta constante (`K_CERCANIA`).

Donde las constantes `b_i` y `b_s` son cotas calculadas en función del tamaño del mapa. Además, calculamos la recarga en función de la batería restante, para que no recargue excesivamente. De este modo las recargas son mucho más eficientes, pues evitamos desviarnos para recargar innecesariamente, o recargar de forma excesiva.

Mejora 3. Capturando el bikini y las zapatillas

Uno de los principales problemas llegados a este punto es que nuestro agente gastará mucha batería en caso de que alguno de sus objetivos se encuentre en medio del bosque o de un océano. Esto podremos arreglarlo sencillamente priorizando capturar el bikini y las zapatillas: cada vez que nuestro agente vea el bikini o las zapatillas, irá a por ellas.

Mejora 4. Evitando ciclos en las recargas

En algunas ocasiones vemos cómo el agente efectúa una recarga y toma un camino hacia el objetivo que le hace volver a recargar. Para arreglar esto, añadiremos el atributo `ultimo_recarga`, que nos indica si la última acción ha sido recargar. Podremos recargar, por tanto, si no hemos recargado antes.

Mejora 5. Ir a por el objetivo si está de camino a la recarga

Observamos que en algunos de los caminos hacia la recarga el objetivo se encuentra cerca, siendo poco eficiente ir a recargar y volver a él. Por tanto, usaremos la heurística –al igual que hemos hecho anteriormente– para saber si dicho objetivo se encuentra "cerca".

Resultados

Después de ejecutar el reto con las especificaciones indicadas 5 veces, obtenemos las siguientes medias:

Mapa	Objetivos	Tiempo restante	Batería restante
`mapa30`	117	1	542.4
`mapa50`	94.8	1	431.2
`mapa75`	70.2	14.6	98.4
`mapa100`	40.4	132.4	86.2
`islas`	26.2	612.4	21.6
`medieval`	35.6	4.6	45.4

Conclusiones

En esta práctica hemos visto cómo para poder construir un agente reactivo/deliberativo, hemos de tener en cuenta una gran cantidad de factores, siendo necesarias múltiples modificaciones en el diseño (algunas de ellas, modificaciones que no dan mejores resultados y que por tanto no han sido usadas). Además, observamos que hemos recurrido a diversos índices y constantes, que podríamos afinar mediante la realización de sucesivas iteraciones del juego.