Celia Arias Martínez

Práctica 2

Nivel 1

Para el algoritmo de **Busqueda en Anchura** he utilizado una cola, y para el algoritmo de **Coste Uniforme** un set que ordena los nodos según la batería que hayan gastado. He tenido en cuenta las recargas.

Nivel 2

En primer lugar me gustaría decir que, debido a que he utilizado variables globales, para el correcto funcionamiento del programa hay que ejecutarlo cada vez que se quiera iniciar un nuevo juego (no funciona si directamente se le da a nuevo juego).

Para el nivel 2 he utilizado diferentes estrategias, según las condiciones en las que se encontrara el programa.

Como he visto que en algunos mapas se agotaba el tiempo transcurrido he decidido que si en la ultima acción llevada a cabo se sobrepasaba un umbral (60 seg) se llevase a cabo una búsqueda parcial, moviéndonos hacia la posición más cercana al objetivo en un radio de 9 casillas. De esta forma he podido solventar algunos momentos en los que el programa iba más lento ya que me permitía moverme hacia otras casillas que fuesen una buena solución local más rápidamente, y así no gastar los 300 segundos.

En el caso de que el algoritmo esté funcionando rápidamente (*busqueda_completa = true*) he llevado a cabo una búsqueda total hacia el objetivo.

También calculo cada vez la distancia desde la posición actual hacia la recarga más cercana (el primer elemento de *set_baterias*) de forma que podemos actualizar el umbral a partir del cual nos dirigiremos hacia la recarga según nuestra situación actual (batería y posición).

En el caso de que no tenga las *zapatillas* o el *bikini* y sepamos donde están (porque hayamos pasado anteriormente por un sitio que esté cerca de ellas) recalculamos el destino para que vaya primero hacia ellas. De esta forma conseguimos equiparnos con las zapatillas y el bikini rápidamente y gastamos menos batería.

Si pasamos cerca de una recarga y cumplimos las condiciones (siempre son que *bateria < tiempo_simulacion* para que los dos valores estén equilibrados y no se gaste uno antes que otro) también cambiamos el destino y nos dirigimos hacia ella.

Si no tenemos plan llamamos a la función *buscarplan(Sensores sensores)* que lo que hace es:

- Si estamos en una casilla de recarga y cumplimos las condiciones mencionadas anteriormente seguimos en la casilla y recargamos la batería.
- Si sabemos dónde hay una recarga y nos queda poca batería nos dirigimos hacia ella. En este caso añadimos como condición un *umbral de tiempo*, para que si nos queda poco tiempo no lo gaste en ir a recargar la batería, y pueda conseguir más objetivos.
- Como he comentado antes dependiendo de si vamos a hacer una búsqueda completa o no llamamos al *AlgoritmoA* o a *avanceRapido*.

En el caso de que tengamos un plan tenemos en cuenta varias opciones:

- Si estamos en una casilla de recarga y se cumple la condición *tiempo-batería* seguimos allí.
- En caso contrario analizamos la siguiente acción. El caso problemático es cuando avanzamos hacia delante, pues nos podemos encontrar un aldeano, o puede que avancemos hacia una casilla desconocida. Si lo que tenemos es una casilla desconocida (lo vemos con *ampliarHorizonte*) rellenamos *mapaResultado* con la nueva información, borramos el plan que teníamos y calculamos otro. Si tenemos un aldeano esperamos 2 segundos, y si no se ha movido no borramos el plan y en la siguiente acción calculamos otro.

Por último actualizamos la batería y el tiempo de simulación que nos queda, y con *cambiarEstado* nos equipamos las zapatillas y el bikini si lo hemos encontrado.

Procedo a explicar ahora las funciones restantes:

estado buscarBateria(int fila_actual, int col_actual)

Es una función que recalcula la distancia que hay desde la situación actual hasta las recarga encontradas. Devuelve un estado que corresponde con el de la recarga más cercana.

bool ComportamientoJugador::bateriaCerca(Sensores sensores)

Función que mira si en un radio determinado (yo he puesto 5) se encuentra una recarga. La he utilizado para poder recuperar batería cuando estamos cerca de una de ellas, y así no tener que volver cuando estemos lejos. Devuelve *true* en el caso de que la encuentre y *false* en el caso de que no.

void ComportamientoJugador::rellenarMatriz(Sensores sensores)

Esta función la utilizo para varias cosas.

- Por un lado actualizo la variable mapaResultado con la información recibida por los sensores.
- También actualizo el set de casillas conocidas, que utilizamos en el caso de que necesitemos una búsqueda rápida (*avanceRapido*)
- Por último si encuentra un bikini o unas zapatillas guarda el estado en el que las ha encontrado, para así poder buscarlas después.

multiset<posicion, ComparaDistancia> ComportamientoJugador::fronteraCercana(Sensores sensores, const estado & destino)

Esta función también la utilizo para *avanceRapido*. Su función es ordenar las casillas que tenemos en un radio determinado (9 en este caso) según la distancia que tengan hasta el objetivo. De esta forma podemos priorizar las posiciones hacia las que vamos a ver si podemos movernos o no en *avanceRapido*. Devuelve un multiset con las posiciones ordenadas según he comentado.

bool ComportamientoJugador::avanceRapido(Sensores sensores, const estado & destino)

Vemos si es posible ir hasta una posición, ordenada según *fronteraCercana*. Para la posición que lo sea devuelve actualiza el plan que se llevará a cabo en *think*.

void cambiarBateria (nodo & hijo, const char terreno)

cambiarBateria es una función que cambia la batería de un nodo según el terreno sobre el que se encuentre. La batería en este caso se refiere a la batería gastada hasta el momento. Esta función la utilizamos para *Busqueda en coste uniforme* y para el *AlgoritmoA*.

void cambiarAtributos(nodo & hijo, const char terreno)

Esta función actualiza el valor de *bikini* y *zapatillas* del nodo, según el terreno sobre el que se encuentre.

struct ComparaEstrella

Compara el coste de dos nodos según la heurística *F = d + b*, donde d es la distancia hasta el objetivo, y b es la batería que el nodo tiene (en este caso no es la gastada). Sirve para ordenar el multiset de *AlgoritmoA*.

bool ComportamientoJugador::pathFinding_AlgoritmoA(const estado &origen, const estado &destino, list &plan)

Esta función cambia en algunos aspectos respecto a *costoUniforme*.

- Por un lado el multiset utilizado se ordena según *ComparaEstrella*, explicado antes, lo que hace que podamos introducir un nuevo parámetro en la heurística y así desarrollar los nodos que se encuentren más cerca del destino antes, y obtener la solución más rápidamente.
- Tenemos que actualizar el valor del *bikini* y *zapatillas* para que en el caso de que ya las tengamos no le de más valor a los nodos que se encuentren estos objetos.
- Para cada nodo tenemos que calcular la distancia que lo separa del destino.

Por último proporciono los objetivos conseguidos en cada uno de los mapas, de acuerdo con las condiciones establecidas en el guión:

Mapa	Objetivos
Mapa 30	109
Mapa 50	61
Mapa 75	32
Mapa 100	14
Mapa isla	12
Mapa medieval	24