Documentación Práctica 2 I.A.

Juan Sánchez Rodríguez

Nivel 1:

En este nivel simplemente debemos de buscar un camino dependiendo del pathfinding elegido hasta el destino en el caso de no haber ya un plan. Teniendo en cuenta que ya tenemos el mapa la generación de un plan es simple, este plan consiste en una serie de acciones almacenadas que posteriormente son realizadas una por una hasta llegar al objetivo. En esta práctica hemos creado dos pathfindig:

- **pathFinding_Anchura:** Hace una búsqueda en anchura, es decir, nos busca el camino más corto. Para ello usa esencialmente un set de estados (generados), un nodo (current) y una queue (cola). Básicamente va generando distintos caminos creando 3 descendientes, uno en el que gira a la derecha, otro en el que gira a la izquierda y otro en el que avance, esto lo hará en bucle y lo hará por niveles, con eso quiero decir que primero recorrerá los el nivel de los 3 primeros descendientes, después el nivel de sus 9 hijos totales y así sucesivamente hasta llegar al destino. Recalcar que para cada nodo que llevamos vamos almacenando como llegar hasta el en su secuencia.

- **pathFinding_CostoUniforme:** Hace una búsqueda de costo uniforme, es decir, busca el camino con menor coste. Para ello usa un set de estados (generados), un multiset con los nodos (lista) y un nodo (current). Como el anterior genera distintos caminos creando 3 descendientes, uno en el que gira a la derecha, otro en el que gira a la izquierda y otro en el que avance, esto lo hará en bucle pero recorrerá los nodos en función de sus costes totales hasta llegar a ellos (a menor coste antes llegará hasta el), así hasta llegar a su destino. Como antes también cabe destacar que para cada nodo que llevamos vamos almacenando como llegar hasta él en su secuencia. Además como resulta evidente este pathfinding hace uso de otras funciones y structs:
    - **struct ComparaCostes:** en este struct tenemos “bool operator()”, esto tiene como objetivo decirnos si un coste es mayor que otro.
    - **int ComportamientoJugador::CalculaCostes(unsigned char casilla, bool bikini, bool zapatillas, int coste):** esta función sirve para sumarle a un nodo el coste actual en función del tipo de casilla

que sea a su coste anterior. Todo esto teniendo en cuenta si ya se han obtenido alguno de los objetos zapatillas, bikini o ambos.

Nivel 2:

En este nivel el agente no conocerá el mapa por lo tanto trazar una ruta será mucho más complicado. Seguimos la estructura anterior donde los movimientos se hacen uno por uno, es decir, para cada movimiento se deberá pasar por el think:

- **Línea 39:** Lo primero que haremos es establecer el tiempo máximo (tiempoMax) que tendrá para procesar un pathfinding en función del tamaño ya que tenemos un límite de 300, esta variable también nos servirá para distinguir si estamos en un mapa de 100 o en uno inferior.
- **Línea 45:** Después en el caso de que nuestra batería supero los 600 puntos y nos queden menos de 250 de tiempo realizamos un pathfinding en anchura, ya que llegado este punto nos dan igual los recursos y lo usaremos como "sprint final" para obtener algunos objetivos más.
- **Línea 58:** Por otro lado si nuestra batería es inferior a 850 y todavía nos quedan 500 de tiempo (y hayamos encontrado alguna casilla de recarga y no estemos ya recargando), borramos el plan actual y generamos uno hasta la casilla de recarga más cercana por coste uniforme.
- **Línea 86:** Si hemos encontrado alguna casilla con bikini y no tenemos bikini borramos el plan actual y trazamos una ruta hacia el mismo por coste uniforme.
- **Línea 98:** Si hemos encontrado alguna casilla con zapatillas y no tenemos zapatillas borramos el plan actual y trazamos una ruta hacia las mismas por coste uniforme.
- **Línea 111:** Si no hay plan busca un plan.
- **Línea 127:** En el caso de que colisione contra un muro borra el plan y crea uno nuevo sabiendo la existencia de ese muro. En el caso de que el mapa no sea de 100 cuando choca contra un aldeano recalcula el plan pero antes se mantiene quieto, esto solo lo hace en los mapas inferiores a 100 ya que si esto se hiciera en estos mapas podría agotar el "Tiempo Consumido".
- **Línea 143:** Evita que el agente caiga por un precipicio, es decir, si tiene delante un precipicio y su siguiente acción es avanzar, elimina su plan y se para.
- **Línea 156:** Intenta evitar que el agente vaya por el agua sin bikini o con bikini y poca batería a no ser que no tenga otra opción de menor coste, para ello elimina el plan actual a no ser que no tenga otra opción mejor.

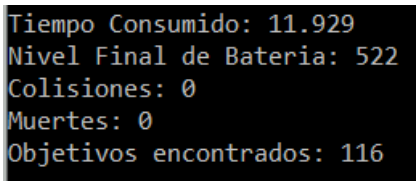
- **Línea 169:** Intenta evitar que el agente vaya por el bosque sin zapatillas, para ello elimina el plan actual.
- **Línea 191:** Comprueba si nos situamos en una casilla con algún objeto, en el caso afirmativo ponemos en true unos bool (bikini o zapatillas), para advertirnos de la posesión de los mismos.
- **Línea 201:** Comprueba si nos situamos en una casilla de recarga en caso afirmativo si nuestra batería es inferior a 2400 y todavía nos quedan más de 1000 de tiempo recargamos hasta que esta condición deje de cumplirse, por otro lado si nuestra batería es inferior a 1200 pero todavía nos quedan más de 500 segundos también continuamos recargando hasta que esta condición deje de cumplirse.

Acabamos de ver el funcionamiento del think, pero dentro de él no paramos de usar una función muy importante a la que yo he llamado **mapeo**, *void ComportamientoJugador::mapeo(estado st, vector<unsigned char> sensor),* esta función es vital, ya que no tenemos nada del mapa descubierto y se encarga de ir actualizándolo conforme el agente va haciendo alguna acción. Además guarda en un set las casillas de recarga que va encontrando para recordarle al agente donde están por si necesita consultarlo rápidamente. También guarda si acaba de encontrar una casilla de zapatillas o bikini para que el agente vaya a por los objetos nada más descubrirlos.

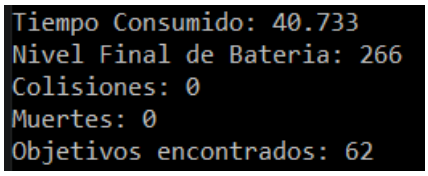
Si nos fijamos en el código vamos que el pathfinding esencial que usa no es "pathFinding_CostoUniforme" sino que es "pathFinding_CostoUniforme_AdaptadoNivel4", este pathfinding hace en esencia lo mismo que el anterior con una ligera diferencia, aquí tiene en cuenta si ya tenemos las zapatillas, el bikini o ambos (en el anterior nivel no hacía falta, ya que nada más empezar no iba a tenerlos). Por otra parte usa también la variable "elTiempo" donde da un tiempo máximo (este dependerá del tamaño del mapa) para procesar una ruta, en el caso de llegar al límite hará la ruta que tenga. La última diferencia es que en el caso de ser un mapa de 100 usará otra función "CalculaCostesMapaGrande" donde las casillas desconocidas (?) las valorará con un tamaño 2, con el objetivo de que si ya conoce un camino seguro hasta su destino que sea de bajo coste lo use en vez de gastar batería en descubrir una nueva ruta.

A modo de ejemplo añado unas capturas con los resultados del código en los mapas de los diferentes tamaños, situado todo inicialmente como el pdf “ValoracionEnObjetivosNivel2.pdf” estipula, para verificar su correcto comportamiento:

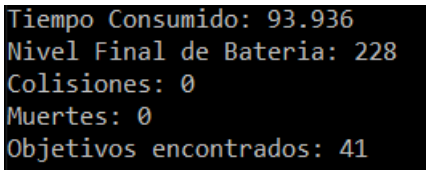
- mapa30.map:



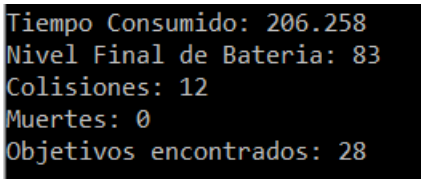
- mapa50.map:



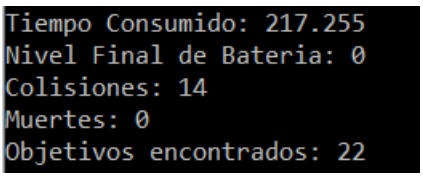
- mapa75.map:



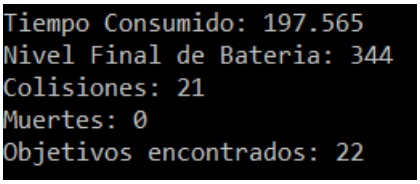
- mapa100.map:



- islas.map:



- medieval.map:



Dificultades encontradas

La mayor dificultad por excelencia ha sido modificar el think a la hora de hacer el mapa “islas.map”, ya que llegado ese momento todos los mapas me daban una puntuación muy alta, inclusive el “medieval.map” con 26 puntos, excepto este. Esto se debía a que el tiempo consumido de procesamiento se agotaba cuando llevaba unos 5 destinos, ya que la idea original era que si iba por el agua o por el bosque sin bikini o zapatillas respectivamente recalculara otro plan, esto hacia que al avanzar por el agua recalculara continuamente. Solucionando este problema he conseguido pasar de esos 5 destinos a 22, pero con el sacrificio de mapa medieval donde ha pasado de 26 a 22 (pongo de ejemplo este porque es el más evidente, otros mapas se han visto afectados como el de 30).