Pablo Rodríguez Ávila B2

Búsqueda en anchura

Para implementar la búsqueda en anchura he diseñado una función en la clase ComportaminetoJugador guiandome de la búsqueda en profundidad que se nos ha suministrado. La principal diferencia de ambos es que el contenedor de la lista de nodos pasa a ser de tipo cola en lugar de pila. Esto es totalmente necesario ya que para recorrer el árbol por niveles necesitamos un contenedor de tipo LIFO.

Búsqueda por coste

En este tipo de búsqueda necesitamos calcular el coste de un recorrido. Teniendo en cuenta lo que dice el guión he diseñado una funcion costeEstado que devuelve el coste de un movimiento en en un determinada casilla.

Se define un nuevo nodo para no mezclarlo con los otros algoritmos. Este nodo se denomina nodoCoste. Este nodoCoste a su vez esta formado un plan (lista de Action) y un estadoCoste. EstadoCoste es otro struct formado por fila, columna, orientacion, gasto y accesorios.

Para calcular el coste de un nodo necesitamos saber si se han recogido el bikini o las zapatillas es por eso que se añade una variable de tipo int en estadoCoste. Se utiliza solo una variable para representar todo en lugar de dos ya que esto aumenta mucho la eficiencia.

El valor 0 corresponde a ningún accesorio, 1 tiene zapatillas, 2 tiene bikini y 3 tiene ambos accesorios.

La energía calculada se suma a la energía previamente calculada por lo que definimos una variable gasto en el struct estadoCoste.

En este algoritmo definimos una cola con proridad denominada abiertos. En cada iteración escogemos el nodo con menor coste y lo eliminamos de esta cola. Si este nodo es solución paramos porque ya tenemos el plan. En caso contrario añadimos a abiertos los descendientes de este nodo. El algoritmo sigue hasta que encontremos una solución o ya no haya mas nodos que analizar.

Pablo Rodríguez Ávila B2

Búsqueda sin información

Para esta búsqueda he implementado una función auxiliar llamada pintaMapa en la que a través de los sensores del jugador podemos rellenar la matriz resultadoMapa.
Un factor a tener en cuenta a la hora de pintar en mapaResultado es que NO podemos escribir fuera de la matriz. Es por este motivo que incorporo otra función auxiliar llamada casillaValida que comprueba que existe esa casilla en la matriz.

También he tenido que añadir código en el método think para poder llamar correctamente a la búsqueda sin información.

Este nivel ha sido implementado con prueba y error constantemente. Primero lo realize con el algoritmo A*. Este algoritmo de tipo voraz. Al ser un algoritmo de tipo voraz se necesita una heurística que nos calcule el valor objetivo de un nodo. Este valor objetivo corresponde a una distancia junto a un coste.

NO OBSTANTE este no ha sido el algoritmo que utilizo. Tras muchas pruebas este algoritmo no era tan eficiente como el algoritmo de coste uniforme. Es por eso que acabe utilizando este. Ambos algoritmos están incluidos en el código.

El algoritmo A* calculaba un plan en cada movimiento y se lo podía permitir porque tenia una gran eficiencia en tiempo de procesador. Asi que con la búsqueda de coste calculo un plan cada 15 movimientos.

Esta búsqueda de coste tiene algunas pequeñas modificaciones de manera que no es la misma que el nivel 3. Los cambios realizados han sido para ganar eficiencia en el procesador.

Ambos algoritmos usan colas de prioridad por lo que ha sido necesario definir los operador < sobre los struct nodo de cada algoritmo.

Aunque en la práctica pone que el jugador se pueda chocar con los aldeanos he añadido una condición en el atributo if que evita estas colisiones utilizando el sensor de superficie. En caso de chocarse con el aldeano cambia el actForward por un actIDLE de manera que no colisiona con él y no gasta bateria.

Pablo Rodríguez Ávila B2

Está implementado que si nuestros sensores detectan un bikini o unas zapatillas cerca del jugador. El jugador establece como objetivo recoger estas zapatillas o este bikini.

A su vez cuando la bateria del jugador llega a un nivel muy bajo, el jugador se dirige a la casilla de Recarga para poder recargar su bateria y seguir consiguiendo objetivos. Aqui entran números factores en juego como el tiempo restante que hay en el momento, el nivel de batería actual, si el jugador tiene zapatillas , etc.

Si el jugador esta ejecutando un plan se espera a que lo termine para que vaya a recargar. Esto sucede cuando tenga menos de 550 bateria. Si el jugador no ha terminado el plan actual y le queda menos de 300 bateria, lo abandona y se dirige inmediatamente a recargar porque corremos el riesgo de que se quede sin bateria si sigue con el plan actual.

Para detectar la bateria, las zapatillas y el bikini se establecen tres variables de estado en jugador.hpp que indican si hemos tenemos bikini/zapatillas y si hemos encontrado bikini/zapatillas/recarga.

En el caso de que nuestro jugador vaya a avanzar a una casilla de tipo agua sin bikini o de tipo bosque sin zapatillas, recalculamos el plan pues estos movimientos conllevan un gran gasto de bateria.

También recalculamos el plan si vamos a avanzar a una casilla de tipo muro o precipicio.

Está implementado que cada 10 unidades de tiempo, si hemos descubierto alguna zona del mapa y no hemos recalculado el plan, volvemos a calcular el plan.

Los resultados obtenidos finalmente son todos Buenos excepto en islas donde el algoritmo logra unos 21 resultados. Estas ejecuciones se han realizado en las mismas circunstancias que hay indicadas en ValoraciónNivel2.pdf