Miguel Medina Martínez Grupo B2

PRACTICA 2 IA:
LOS EXTRAÑOS MUNDOS DE BELKAN

NIVEL 1:

1.1 **Búsqueda en profundidad:** Este algoritmo venia implementado por defecto y ha servido como guía en los siguientes algoritmos que se pedían. Se ha implementado con los nodos cerrados contenidos en un set y conteniendo los nodos abiertos en una pila para representar el funcionamiento de la búsqueda de profundidad.

1.2 **Búsqueda en anchura:** Este algoritmo comparte la mayor parte de su implementación con el algoritmo de búsqueda en profundidad pero adaptado al uso de una lista para los nodos abiertos. Permitiendo así una estructura FIFO para expandir los nodos.

1.3 **Búsqueda de costo uniforme:** Para este algoritmo hemos creado un struct nodo personalizado (**nodoCoste**) que contiene dos nuevas variables enteras que explicaremos a continuación.

El primer entero (**coste**) se encarga de controlar el coste de batería que supondría recorrer el camino que llega a ese nodo. Con esto se nos permite ordenar a los caminos que supondrían un menor coste de batería. El segundo entero (**equip**) se encarga de controlar si se han recogido por el camino el bikini o las zapatillas. En un principio implemente esta funcionalidad con dos booleanos independientes pero, al hacer las comprobaciones entre nodos para distinguir si son iguales, era muy ineficiente. De esta manera (usando un solo entero) la velocidad de cálculo se mejoraba mucho. Explicado esto, el funcionamiento del entero es el siguiente: si el valor de **equip** es 0, no se ha recogido nada; si el valor es 1, se han recogido las zapatillas; si el valor es 2, se ha recogido el bikini; y si el valor es 3, se han recogido ambos objetos.

Además hemos creado varias funciones nuevas para controlar los contenedores de los nodos abiertos y cerrados. Los nodos abiertos se han almacenado en una cola de prioridad para la que previamente se ha

implementado una sobrecarga del operador menor que, para así ordenar los nodos de menor a mayor coste. Además se ha expandido la comparación de estados que se nos incluía a una **comparación** de **nodoCoste** que compara el coste y el equipamiento, además de lo previamente implementado.

Para calcular el valor de las nuevas variables hemos creado la función **calcularCoste** que, dependiendo de en qué casilla caes, devuelve el coste que se aplica sobre la batería. Además de esto, cuando se pasa por unas zapatillas o un bikini cambia el valor de **equip** al adecuado. En esta función además he incluido varios ifs. En agua y bosque para reducir el coste si se cuenta con bikini o zapatillas respectivamente y en el caso de casilla desconocida ('?') para intentar que cuando no se tienen ya las dos herramientas se incentive un poco más el descubrimiento por el mapa(esto solo se aplica para el nivel 2)

El algoritmo como tal es como se explica en las diapositivas. Con las funciones que he explicado antes consigo poner en primer lugar los nodos con recorrido más bajo. Además en cada ejecución de los algoritmos he añadido un while que nos proporcionó el profesor para descartar los nodos que ya están en cerrados pero siguen en abiertos. Esto mejora muchísimo la eficiencia del algoritmo.

NIVEL 2:

Para el nivel 2 he usado el algoritmo de costo uniforme con una pequeña modificación en la función de comparar nodos para mejorar su rendimiento (quitándole cosas que comparar).

Tras explicar esto voy a pasar a explicar el **think** del personaje. Empezamos creando un plan para ir al primer objetivo. Después de esto entra en acción la función **rellenarVista**. Esta función originalmente solo servía para ir pintando en la variable **mapaResultado** conforme vamos descubriendo el mundo. Sin embargo, poco a poco le he ido añadiendo funcionalidades para detectar si se ha descubierto algún trozo de mapa, si se descubre una casilla de zapatillas, una de bikini o una casilla de batería poniendo unas variables booleanas que indican que se han encontrado(**zapaE, bikiE, batE**).

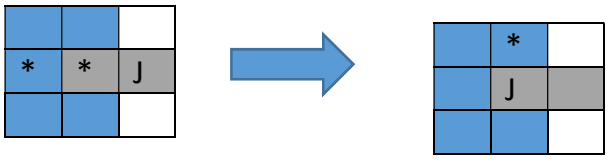
Lo siguiente que ocurre en el código es un if que comprueba si se ha llegado al destino para seleccionar uno nuevo. Este destino además dependerá de si previamente se ha detectado un bikini o una zapatilla (se explicara en detalle más adelante). A continuacion se comprueba si se ha pasado por un bikini o una zapatilla y pone un booleano de estado a true. Después

comprobamos la casilla de delante para ver si tenemos un aldeano delante (lo que conllevará añadir un paso idle al plan hasta que se aparte) y comprobar si tenemos un precipicio o un muro delante para señalar que no hay plan y dar paso a que se recalcule el plan.

Ahora en el código comprobamos si hemos encontrado una casilla de bikini o zapatilla y si no los tenemos ya. Guardamos el destino al que íbamos a ir previamente y fijamos como nuevo destino ésta casilla. Aquí entra en juego los ifs que teníamos en la comprobación de haber alcanzado previamente (de los que hemos hablado antes) ya que comprueban si se había cambiado el objetivo para ir a una casilla especial y se vuelve a fijar el destino original.

Tras hacer esto realizo otra comprobación para recalcular el plan para ir a la casilla especial inmediatamente después de detectarla.

El siguiente método sirve para intentar reducir pasar por casillas ineficientes cuando se planifica el camino a seguir sin tener el mapa totalmente descubierto. Aquí entra en juego la variable que marca si se ha actualizado el mapa (calculada en **rellenarVista)** y hago que se recalcule el plan siempre que vaya a pasar por agua o bosque y no tenga el equipamiento adecuado. Además si se dispone del equipamiento adecuado pero no está muy lejos también se recalcula. Aquí surgía un problema cuando la acción previa no era avanzar, ya que podía darse una situación como esta:



Siendo el asterisco el plan que se calcula, el jugador se encuentra con agua delante pero no ha descubierto mapa (al haber girado no lo descubre porque venía de su derecha) por lo tanto tuve que añadir una variable de estado para controlar esto y recalcular cuando se diese el caso. Este problema también ocurría cuando se ponía en idle para esperar a que un aldeano se moviese.

Ahora pasamos a la administración de la recarga. Compruebo primero que se haya encontrado la batería, que no se esté ya cargando y que el jugador no esté muy cerca del objetivo. Si se dan estas condiciones compruebo dos supuestos: El nivel de la batería no es crítico PERO la batería esta en rango de visión y que la batería este en un nivel demasiado bajo y necesite recargar ya. El primer supuesto solo ocurre cuando queda mucho tiempo y hace una recarga más corta (generalmente). El segundo supuesto lleva a una recarga más larga. Si alguna de estas comprobaciones se cumple, se guarda el destino actual en otra variable de estado auxiliar (como en el caso de encontrar bikini o zapatillas), se

fija como destino la batería, se recalcula el plan y se indica que se está en proceso de carga **cargando=true.**

Para hacer que la batería se cargue realizo una comprobación de si se está cargando y si se ha llegado a la casilla de carga. A continuación en base al tiempo restante se determina hasta que nivel de batería se ha de llegar (añadiendo acciones idle al plan) y cuando se termina la carga se restaura el destino al que se dirigía el jugador originalmente y se señala que la carga ha terminado (**cargando=false**).

Añado unas comprobaciones para controlar si la última acción ha sigo avanzar para usar esta información (variable **ultNoAvanza**) para decidir si se recalcula el plan, como he comentado previamente.

El siguiente paso se encarga de seleccionar el siguiente paso del plan para devolverlo al final de la función.

Por la variable **tiempo** (que se inicializa a 3000 y que controla el tiempo que le queda a nuestro programa para acabar) se decrementa a cada acción que realiza nuestro personaje.