Memoria Práctica 2 - Mario López González

Nivel 1:

Este nivel lo he implementado con el algoritmo de búsqueda en anchura. El método trata de buscar el camino con el menor número de pasos. Para realizar la búsqueda hago uso de una lista de nodos explorados declarada como set<estado, ComparaEstados> y una cola de nodos que tienen que ser revisados denominada Abiertos.

Inicialmente, introduzco en Abiertos el nodo actual. A continuación, itero hasta que haya explorado todos los nodos pendientes de revisar y haya llegado al destino. Dentro de esta iteración saco de abiertos el primer nodo introducido y, para optimizar el tiempo, no compruebo si ya se encuentra en Abiertos. Antes de introducirlo en Cerrados, compruebo que no esté explorado, para evitar que los nodos que haya duplicados en Abiertos no se vuelvan a tener en cuenta para explorar. Si no lo está, expando el nodo para generar los descendientes correspondientes a girar a la derecha, a la izquierda y continuar de frente. Para cada uno compruebo si ya ha sido explorado, si no es así lo introduzco en Abiertos para explorarlo más tarde y añado a la secuencia de acciones la acción correspondiente a cada descendiente. Si quedan nodos en abiertos, actualizo el valor del nodo actual al primer nodo introducido en Abiertos.

Nivel 2:

Este nivel lo he implementado con el algoritmo de búsqueda de costo uniforme. El método trata de buscar el camino con el mínimo coste de batería. Para realizar la búsqueda hago uso de una lista de nodos explorados, como en el nivel 1, y una cola con prioridad de vectores de nodos ordenada en función del coste de llegar a cada nodo.

Como tener o no las zapatillas o el bikini modifican el consumo de batería, he declarado dos variables booleanas en el struct estado para indicar si tiene zapatillas o bikini. Como consecuencia de esta declaración, he tenido que modificar el struct ComparaEstados para que tenga en cuenta los dos objetos.

```
struct ComparaEstados{
    bool operator()(const estado &a, const estado &n) const{
        if ((a.fila > n.fila) or (a.fila == n.fila and a.columna > n.columna) or
          (a.fila == n.fila and a.columna == n.columna and a.orientacion > n.orientacion) or
          (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion and
 a.zapatillas > n.zapatillas) or
          (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion and
 a.zapatillas == n.zapatillas and a.bikini > n.bikini))
            return true;
        else
            return false;
    }
};
```

Para actualizar el valor de las zapatillas y el bikini, he implementado un método que comprueba si está en una casilla en la que puede adquirir alguno de los dos objetos y modificar el valor de los atributos zapatillas y bikini del estado, teniendo en cuenta que no puede tener ambos al mismo tiempo.

```
void ComportamientoJugador::actualizarEstado(estado &st){
    //Pongo al estado las zapatillas
    if(mapaResultado[st.fila][st.columna] == 'D'){
        st.bikini = false;
        st.zapatillas = true;
    }
    //Pongo al estado el bikini
    else if(mapaResultado[st.fila][st.columna] == 'K'){
            st.zapatillas = false;
            st.bikini = true;
    }
}
```

Para la cola con prioridad he tenido que implementar un struct ComparaConsumo que compara dos nodos conforme al valor de su variable costeAcumulado declarada en el struct estado. Esta variable contiene el consumo de batería para llegar al nodo actual desde la raíz, calculada como suma del costeAcumulado del nodo padre mas el coste del estado actual en función de la acción a realizar, el tipo de casilla, y, si es necesario, tener en cuenta si dispone de bikini o zapatillas.

```
struct ComparaConsumo{
    bool operator() (const nodo &n1, const nodo &n2){
        return n1.st.costeAcumulado > n2.st.costeAcumulado;
    }
};
```

La estructura de la exploración es igual a la del nivel 1, pero con dos cambios. Para cada nodo descendiente calculo el consumo de batería para llegar hasta él y actualizo la variable costeAcumulado. Además, si el nodo descendiente es continuar de frente, actualizo, si es necesario, el valor de las variables zapatillas y bikini del nuevo nodo.

```
void ComportamientoJugador::calcularConsumo(const estado &padre, estado &hijo, Action accion){
    int costeCasilla = calcularCosteCasilla(hijo, mapaResultado, accion);
    hijo.costeAcumulado = costeCasilla + padre.costeAcumulado;
}
```

Nivel 4:

Este nivel lo he implementado con el algoritmo de búsqueda A Estrella.

En primer lugar, añado dos nuevas variables al struct estado, distanciaDestino que es la componente heurística, h(n), que calculada como la distancia Manhattan desde el nodo actual hasta el nodo destino, por un método llamado actualizarDistancia(). Por otro lado, la variable estimaciónCosteTotal, f(n), que se calcula como la suma de la variable distanciaDestino y la variable costeAcumulado, g(n), declarada para el nivel 2.

```
void ComportamientoJugador::actualizarDistancia(estado &st, const estado &destino){
    st.distanciaDestino = abs(st.fila - destino.fila) + abs(st.columna - destino.columna);
}

void ComportamientoJugador::actualizarEstimacionCoste(estado &st){
    st.estimacionCosteTotal = st.distanciaDestino + st.costeAcumulado;
}
```

El algoritmo de búsqueda es el mismo que en el nivel 2 pero con ligeras modificaciones la cola con prioridad se ordena en función de la variable estimacionCosteTotal con el struct ComparaEstimacion.

```
struct ComparaEstimacion{
    bool operator() (const nodo &n1, const nodo &n2){
        return n1.st.estimacionCosteTotal > n2.st.estimacionCosteTotal;
    }
};
```

En este caso, si un nodo descendiente, se encuentra en Cerrados, compruebo si el nuevo descendiente es mejor que el que estaba ya en Cerrados, si es mejor, lo intercambio.

```
if (Cerrados.find(hijoTurnR.st) == Cerrados.end() ){
    Abiertos.push(hijoTurnR);
}
else{
    set<estado,ComparaEstados>::iterator it = Cerrados.find(hijoTurnR.st);
    if((*it).estimacionCosteTotal > hijoTurnR.st.estimacionCosteTotal){
        Cerrados.erase(it);
        Cerrados.insert(hijoTurnR.st);
    }
}
```

Ha sido necesario declarar nuevas variables en jugador:

- int pasosDados: contabiliza el número de pasos que da el jugador.
- list <estado> objetivosPendientes: almacena los objetivos pendientes de visitar.
- list <estado> bateriasEncontradas: almacena las baterías que visualiza a medida que explora el mapa.
- int instantesConsumidos: contabiliza los instantes de tiempo consumidos por el jugador.

He implementado una función que pinta el mapa a medida que avanza para tener en cuenta el terreno que visualiza y así poder buscar la mejor ruta hacia el objetivo. Esta función utiliza el atributo terreno de la variable sensores y dependiendo de la orientación pinta la parte del mapa que estará viendo el jugador. En esta función, si encuentra una casilla de batería la introduce en la lista bateriasEncontradas de jugador si no estaba ya introducida. Para saber si estaba introducida o no he implementado una función “buscar” que comprueba si un estado está dentro de una lista de estados.

El jugador recalcula una nueva ruta hacia el objetivo cada dos pasos o cuando se encuentre un obstáculo en el camino. Para poder realizar dicha tarea he hecho uso del método HayObstaculoDelante() pasándole como parámetro el estado actual o cuando haya dado dos pasos. El objetivo de recalcular una nueva ruta es buscar un nuevo camino que consuma menos batería que el anterior ya que a medida que avanza se va pintando el mapa y por lo tanto, puede ver qué casilla es la mejor.

```
if(pasosDados % 2 == 0 || HayObstaculoDelante(actual)){
    hayPlan = pathFinding(sensores.nivel, actual, objetivosPendientes, plan);
}
```

Para escoger un objetivo, ordeno objetivosPendientes de menor a mayor distancia desde la posición actual. Para realizar la ordenación he implementado un método donde, para cada objetivo, calculo la distancia a la posición actual y, en una variable auxiliar de tipo list<estado> llamada auxList, introduzco el estado con menor distancia Manhattan con push_back() y lo borro de objetivosPendientes. Cuando objetivosPendientes está vacío se vuelca el contenido de la variable auxList en objetivosPendientes, de este modo objetivosPendientes queda ordenado.

```
void ComportamientoJugador::ordenarLista(list<estado> &lista){

    list<estado> auxList;
    estado menor, aux;
    list<estado>::iterator it, auxIt;

    while(!lista.empty()){

        menor = lista.front();
        actualizarDistancia(menor, actual);
        it = lista.begin();
        auxIt = it;
        ++it;

        for(it; it != lista.end(); ++it){
            aux = *it;
            actualizarDistancia(aux, actual);
            if(aux.distanciaDestino < menor.distanciaDestino){
                menor = aux;
                auxIt = it;
            }
        }

        lista.erase(auxIt);
        auxList.push_back(*auxIt);
    }

    lista = auxList;
}
```

Con el fin de optimizar el número de objetivos alcanzados he realizado modificaciones en el método think las cuáles sólo son accesibles cuando el nivel del juego es el 4. Las describiré a continuación.

Cuando el jugador llega a un objetivo, se elimina de objetivosPendientes con pop_front() puesto que al estar ordenados el primer objetivo de la lista es el que acabo de encontrar. Una vez borro el objetivo, se ordena la lista para que se dirija al objetivo más cercano. En el momento en que la lista de objetivosPendientes queda vacía, es decir, se han encontrado

todos los objetivos, vuelco el contenido de la variable objetivos en objetivosPendientes y los ordeno, esta acción se lleva a cabo en pathFinding().

```
case 4: cout << "Algoritmo de busqueda usado en el reto\n";
    if(objetivosPendientes.empty()){
        objetivosPendientes = objetivos;
        ordenarLista(objetivosPendientes);
    }
    un_objetivo = objetivosPendientes.front();
    return pathFinding_AEstrella(origen, un_objetivo, plan);
    break;
```

```
//Si está en un objetivo
if(actual.fila == objetivosPendientes.front().fila && actual.columna == objetivosPendientes.front().columna){
    //Elimino el primer objetivo puesto que es en el que estoy
    objetivosPendientes.pop_front();
    //Vuelvo a ordenar para ir a los objetivos más cercanos a la posicion actual
    ordenarLista(objetivosPendientes);
}
```

Puesto que en este nivel los destinos son infinitos, es necesario recargar la batería y para ello he declarado una lista de estados en los atributos de jugador para almacenar las casillas de batería que encuentro por el camino. Tras varias pruebas, determiné que sería necesario recargar cuando tuviera menos de 1000 de batería. Por lo tanto, si ha encontrado baterías y necesita recargar, introduce en objetivosPenditenes el estado de la batería más cercana para que, cuando esté cerca o haya recorrido todos los objetivos se disponga a recargar.

En el caso de que se posicione en una casilla de batería y hubiera consumido menos de 2500 instantes de tiempo recargará la batería hasta 2980, pero cuando le queden 200 instantes de tiempo recargará solo hasta 250 de batería puesto que habrá recorrido gran parte del mapa y escogerá caminos con un coste de batería mínimo, esto es así para priorizar los objetivos y que no se demore en recargar la batería completamente.

```
if(((sensores.bateria < 2980 && instantesConsumidos < 2500) || (sensores.bateria < 250 && instantesConsumidos >= 2800))
&& mapaResultado[actual.fila][actual.columna] == 'X'){
    hayPlan = true;
    return actIDLE;
}

if(necesitaRecargar(sensores) && !bateriasEncontradas.empty()){
    if(instantesConsumidos < 2500 || (instantesConsumidos >= 2500 && sensores.bateria <= 500)){
        ordenarLista(bateriasEncontradas);
        if(!buscar(bateriasEncontradas.front(), objetivosPendientes))
            objetivosPendientes.push_back(bateriasEncontradas.front());
    }
}
```

Para tener en cuenta los instantes de tiempo consumido, he declarado una variable de tipo int en los atributos de jugador que se incrementa al principio del método think.

Por último, para no colisionar, espero a que los aldeanos se aparten. Para realizar esta tarea, compruebo si la acción es avanzar y con el atributo de superficie de sensores compruebo si hay un aldeano justo delante, en ese caso devuelvo que la acción sea actIDLE.

```
if(plan.front() == actFORWARD && sensores.superficie[2] == 'a')
    return actIDLE;
```