Práctica 2

Inteligencia Artificial

Nivel 1:

Para resolver las diferentes búsquedas que nos pide el nivel 1 , se nos da el método de la búsqueda por profundidad.

El método de la búsqueda por anchura , es igual que el método de la búsqueda en profundidad , pero cambiando las pilas por colas.

Por lo tanto , agregamos lo siguiente en nuestro archivo "jugador.cpp":

En la parte superior donde los include :

```
#include <queue>
```

Cambiamos la declaración de la pila por esta :

```
queue<nodo> cola;
```

Además , cambiamos todas las variables que se llamaban pila , por cola.

También , necesitaremos cambiar el método "top()" por "front()" ya que las pilas utilizan top y las colas front.

Podemos observar el funcionamiento de este método , generando hijos al girar a la derecha , izquierda y al avanzar (donde comprueba primero si hay un obstáculo) y donde al final pasa al siguiente nodo de la cola mientras el destino no se haya alcanzado (todo dentro del while).

(No entraba el código de la búsqueda en profundidad para no superar el limite de 5 paginas , así que he dejado un enlace al código).

https://github.com/vaderrama/IA/blob/master/Practica%202/jugador.cpp

La búsqueda por coste uniforme :

Para resolver este método hemos partido desde el algoritmo de búsqueda por anchura , pero cambiando la cola por una cola por prioridad declarada de la siguiente manera :

```
priority_queue<nodo,vector<nodo>,ComparaCostes> cola;
```

También tendremos que añadir una variable “coste” a nuestro Struct de nodo :

```
struct nodo{
    estado st;
    list<Action> secuencia;
    int coste;
```

Ademas , para resolver esta búsqueda , he tenido que implementar los siguientes métodos :

```
struct ComparaCostes{
    bool operator() (const nodo &a, const nodo &n) const{

        if (a.coste > n.coste )
            return true;
        else
            return false;
        }
```

El cual se le pasa a la declaración de la cola por prioridad para que vaya ordenando los nodos que vamos introduciendo en la cola según su implementación (coste menor primero).

```
int calcularCosteCasilla(unsigned char casilla){
    int coste = 0;

    if (casilla == 'A')
            coste = 10;
    if (casilla == 'B')
            coste = 5;
    if (casilla == 'T')
            coste = 2;
    if (casilla == 'S')
            coste = 1;

return coste;

}

int ComportamientoJugador::DevolverCoste(estado &hijo){
    int fil=hijo.fila, col=hijo.columna;
    int coste=1;


    coste = calcularCosteCasilla(mapaResultado[fil][col]);

    return coste;
}
```

También he tenido que implementar estos dos métodos siguientes. El método devolver coste lo utilizamos para ver en qué posición se encuentra el hijo y poder calcular el coste de la casilla a la que va a avanzar. Así como calcularCosteCasilla únicamente devuelve un entero con el coste de la casilla.

Seguidamente se va a mostrar la modificación realizada en el método Búsqueda por Anchura para el correcto funcionamiento de la Búsqueda por coste uniforme , ya que necesitamos acumular el coste para poder ordenarlo más tarde.

```
// Generar descendiente de girar a la derecha
nodo hijoTurnR = current;

hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion+1)%4;
if (generados.find(hijoTurnR.st) == generados.end()){
    hijoTurnR.coste = hijoTurnR.coste +1;
    hijoTurnR.secuencia.push_back(actTURN_R);
    cola.push(hijoTurnR);

}

// Generar descendiente de girar a la izquierda
nodo hijoTurnL = current;

hijoTurnL.st.orientacion = (hijoTurnL.st.orientacion+3)%4;
if (generados.find(hijoTurnL.st) == generados.end()){
    hijoTurnL.coste = hijoTurnL.coste +1;
    hijoTurnL.secuencia.push_back(actTURN_L);
    cola.push(hijoTurnL);
}

// Generar descendiente de avanzar
nodo hijoForward = current;

if (!HayObstaculoDelante(hijoForward.st)){
    if (generados.find(hijoForward.st) == generados.end()){
        hijoForward.coste =
            hijoForward.coste+DevolverCoste(hijoForward.st);
        hijoForward.secuencia.push_back(actFORWARD);
        cola.push(hijoForward);
    }
}
```

Como podemos observar , en los tres apartados que tenemos (generar hijo girar izquierda , gira derecha y avanzar) , necesitamos acumular el coste , por lo tanto para los giros únicamente sumamos 1 y para avanzar llamamos al método “DevolverCoste()” al cual le pasamos el estado del hijo.