Práctica 2 Inteligencia Artificial

Agente deliberativo

David Gil Bautista C1

Índice

1. Descripción del problema

En esta práctica se nos pedía modificar la anterior (Los mundos de Belkan), la cual constaba de un agente reactivo ubicado en un mundo desconocido el cual debía explorar evitando la muerte.

Para esta práctica había que modificar el comportamiento de dicho agente para que actuara como uno deliberativo. Además, se incluye un nuevo objeto llamado “regalo" y un nuevo personaje “rey” al cual debemos entregarle regalos para completar misiones y ganar más puntos.

En definitiva para este problema debemos recorrer el mundo, hallar regalos y entregarlos a los reyes.

2. Solución

Para solucionar este problema he optado por usar un algoritmo de búsqueda para hacer que el agente encuentre una serie de movimientos los cuales lo lleven a una posición concreta.

He optado por usar el algoritmo “A estrella”, ya que es un algoritmo capaz de encontrar una solución óptima y no consume tanto tiempo como una búsqueda en profundidad.

He seleccionado un contenedor de tipo pila (stack de la stl) para almacenar el plan que el agente debe seguir, ya que, al usar el “A estrella” el primer movimiento que seleccionemos será el último que nuestro agente deba ejecutar, al usar una pila este plan queda ordenado.

Lo primero será describir la estructura de datos que hemos usado como estado.

```cpp
#include "../Comportamientos_Jugador/jugador.hpp"
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <set>

using namespace std;

class nodo{
    public:

        int fil;
        int col;
        int orientacion;
        int f;
        int g;
        int h;
        nodo *padre;

        int getF(){return fil;}

        int getC(){return col;}

        int getO(){return orientacion;}

        nodo(){
            fil = col = orientacion = f = g = h = 0;

            padre = NULL;
        }

        nodo(int f, int c, int o, int gf, nodo *p){
            fil = f;
            col = c;
            orientacion = o;
            g = gf;

            padre = p;
        }

        nodo operator=(const nodo  &copia) const{

            nodo cop = nodo(copia.fil, copia.col, copia.orientacion, copia.g, copia.padre);
            cop.f = copia.f;
            cop.h = copia.h;

            return cop;
        }
```

Nuestro estado tendrá el valor de la fila, columna en la que estamos, la posición hacia la que miramos (orientación), tres enteros que darán valor a la función heurística que indica lo bueno que es un nodo y un elemento nodo que hará referencia al padre que ha extendido al nodo en cuestión.

```
        }

        bool operator==(const nodo &otro) const{
            if (this->fil == otro.fil and this->col==otro.col and this->orientacion==otro.orientacion
                and this->f==otro.f and this->g==otro.g and this->h==otro.h and this->padre==otro.padre){
                return true;
            }else
                return false;

        }

        bool operator!=(const nodo  &otro) const{
            return !(*this==otro);
        }

        bool operator< (const nodo &otro) const{

            if (*this==otro)
                return false;
            else if (this->h<otro.h)
                return true;
            else if(this->f<otro.f)
                return true;
            else if(this->g<otro.g)
                return true;
            else
                return false;
        }

        void heuristica(nodo otro){

            if (this->fil - otro.fil <0)
                h += otro.fil - this->fil;
            else
                h += this->fil - otro.fil;

            if (this->col - otro.col <0)
                h += otro.col - this->col;
            else
                h += this->col- otro.col;
        }

        int funcion(){
            this->f= this->g + this->h;

            return this->f;
        }


};
```

Contamos con las funciones básicas (constructor, constructor por defecto, operadores) y con la heurística y la función heurística.

Para la heurística me he basado en la distancia Manhattan para determinar la distancia estimada desde la posición actual hasta una objetivo.

La función heurística sumará el valor del resultado obtenido con el método anterior más un valor ‘g’ que determina el costo desde el nodo origen hasta el actual.
Algoritmo A estrella:

```
bool aStar( nodo &origen, pair<int,int> meta, stack<Action> &plan, vector<vector<unsigned char>> mapaResultado){

    nodo destino= nodo(meta.first, meta.second, 0, -1, NULL);

    origen.heuristica(destino);

    set<nodo> abiertos;
    set<nodo> cerrados;

    //abiertos.insert(destino);  //**********************************************************
    abiertos.insert(origen);

    nodo actual = origen;

    bool primera = true;

    while(!abiertos.empty() and actual!=destino){

        if(primera){
            primera = false;
            cerrados.insert(actual);
            abiertos.erase(abiertos.begin());
        }else{
            cerrados.insert(actual);
            actual = *abiertos.begin();
            abiertos.erase(abiertos.begin());
        }

        nodo n1;
        nodo n2;
        nodo n3;


        unsigned char front;
        unsigned char left;
        unsigned char right;

        switch(origen.getO()){
            case 0://Norte
                n1 = nodo(actual.getF()-1, actual.getC(), actual.getO(), actual.g+1, &actual);
                n2 = nodo(actual.getF(), actual.getC(), 1, actual.g+1, &actual);
                n3 = nodo(actual.getF(), actual.getC(), 3, actual.g+1, &actual);

                front = mapaResultado[actual.getF()-1][actual.getC()];
                left = mapaResultado[actual.getF()][actual.getC()-1];
                right= mapaResultado[actual.getF()][actual.getC()+1];

                if (front == 'T' or front == 'S' or front == 'K'){
                    n1.heuristica(destino);
                    abiertos.insert(n1);
                }
```

A nuestro algoritmo le daremos un nodo desde el que empezar la búsqueda, una posición a la que deseamos ir, el plan que devolveremos y el mapa actual para ayudarnos a elegir el mejor camino.

Lo primero será crear la lista de nodos abiertos y cerrados, usaremos un set ya que a la hora de elegir los nodos abiertos gracias al set escogeremos el que mejor heurística tenga (función implementada en nuestra clase nodo).

Tras crear el nodo destino con las coordenadas que hemos obtenido por referencia lo introducimos junto al nodo origen en la lista de cerrados. Una vez hecho esto entramos en un bucle cuyo fin será que la lista de nodos abiertos quede vacía o que el nodo actual sea nuestro objetivo.

Dentro del bucle usamos un nodo auxiliar con el que podamos ir avanzando en nuestra búsqueda y del cual crearemos 3 hijos, uno en una posición por delante, otro girando a la izquierda y otro a la derecha.

Para crear los nodos hijos dependeremos de la orientación del nodo actual y dependiendo de su valor crearemos los hijos (switch). Cuando creamos los hijos debemos cuestionarnos si es buena idea introducirlos en nuestra lista de nodos abiertos porque puede que un camino sea imposible y lo calculemos de todas formas. Introduciremos todo nodo que esté situado sobre una posición válida y si lo está calcularemos su heurística y lo introduciremos en la lista de nodos abiertos. Al finalizar el switch el bucle vuelve a comenzar y se cierra el nodo usado.

```
                n3 = nodo(actual.getF(), actual.getC(), 2, actual.g+1, &actual);

                front = mapaResultado[actual.getF()][actual.getC()-1];
                left = mapaResultado[actual.getF()+1][actual.getC()];
                right= mapaResultado[actual.getF()-1][actual.getC()];

                if (front == 'T' or front == 'S' or front == 'K'){
                    n1.heuristica(destino);
                    abiertos.insert(n1);
                }

                if (right == 'T' or right == 'S' or right == 'K'){
                    n2.heuristica(destino);
                    abiertos.insert(n2);
                }

                if (left == 'T' or left == 'S' or left == 'K'){
                    n3.heuristica(destino);
                    abiertos.insert(n3);
                }

                break;
            }//Fin Switch

        }


        while(actual != origen){

            actual = *cerrados.begin();

            if (actual.getF() != actual.padre->fil or actual.getC() != actual.padre->col){
                plan.push(actFORWARD);
            }else if(actual.getO() > actual.padre->getO()){
                plan.push(actTURN_R);
            }else if(actual.getO() < actual.padre->getO()){
                plan.push(actTURN_L);
            }

            cerrados.erase(actual);
        }

        return true;

}
```

Para finalizar, recorreremos la lista de cerrados y iremos actualizando el plan según los cambios que hayamos tenido entre los nodos. Hecho esto el agente comenzará a recorrer el plan hasta que llegue a su destino.

Modificaciones en el main:

```
// Encontrar reyes
if (noRey and bien_situado){

    unsigned char rey;

    for (int i = 0; i < MAX_SIZE; ++i){
        for (int j = 0; j < MAX_SIZE; ++j){
            rey = mapaResultado[i][j];

            if (rey == 'r')
            {
                reyes[reyes.size()].first=i;
                reyes[reyes.size()].second=j;

                noRey = false;
            }
        }
    }


}
```

Para facilitar la ejecución y creación de planes guardaremos la posición de los reyes.

```
if(!noRey and bien_situado and tengo_regalo and !estoy_ejecutando_plan){
    nodo origen;
    origen.fil = fil;
    origen.col = col;
    origen.orientacion = brujula;

    int distancia[reyes.size()];

    for (int i = 0; i < reyes.size(); i++) {

        distancia[i] = 0;

        if ((reyes[i].first - fil) < 0) {
            distancia[i] += fil-reyes[i].first;
        }else
            distancia[i] += reyes[i].first-fil;

        if ((reyes[i].second - col) < 0) {
            distancia[i] += col-reyes[i].second;
        }else
            distancia[i] += reyes[i].second-col;

    }

    int *menor = &(distancia[0]);
    int index=0;

    for(int j=1; j<reyes.size(); ++j){
        if(*(menor) > distancia[j]){
            menor=&(distancia[j]);
            index = j;
        }
    }

    pair<int,int> destino;
    destino.first = reyes[index].first;
    destino.second = reyes[index].second;

    estoy_ejecutando_plan = aStar(origen, destino, plan, mapaResultado);
```

Para hacer el agente un poco más inteligente siempre que vaya a ejecutar un plan consultará la información que tiene para escoger el plan que más beneficio proporcione, es decir, el que esté más cerca.

Para los regalos se realiza lo mismo pero una vez escogido el regalo que más cerca esté se decidirá si es rentable o no ir, ya que puede que al ir hacia un regalo hayas perdido más vidas de las que te proporcione al entregárselo al rey.

3. Conclusiones

Al igual que en la práctica anterior el código no está depurado completamente por lo que en ocasiones puede fallar. En sí, la estructura del código y los algoritmos usados son buenos y todo funcionando correctamente proporciona una solución bastante óptima y eficiente aunque se podría mejorar.

Teniendo en cuenta que la primera práctica fue un fracaso absoluto en esta he conseguido mejorar muchos errores que ya tenía además de implementar un método de búsqueda que funciona y no consume muchos recursos.