Práctica 4. Exploración de grafos

Moises Guerrero Lopez
moises.guerrerolopez@alum.uca.es
Teléfono: xxxxxxxx

22 de enero de 2019

1. Comente el funcionamiento del algoritmo y describa las estructuras necesarias para llevar a cabo su implementación.

   Hacemos uso del algoritmo A*. En este algoritmo hacemos uso de una lista de nodos cerrados, donde almacenamos los nodos ya comprobados y un montículo de nodos abiertos, donde se almacenan los nodos a los que podemos acceder y no han sido comprobados. El algoritmo funciona de la siguiente manera: el objetivo es llegar al nodo objetivo, así que mientras el nodo actual no sea el nodo objetivo seguimos generando nodos, primero almacenamos el nodo actual en cerrados, porque ya lo hemos comprobado,luego por cada nodo de la lista de adyacentes del nodo actual comprobamos si están en cerrados y en abiertos, si no están en ambos generamos sus datos según las distancias y lo añadimos al montículo de abiertos, si no esta en cerrados pero si en abiertos comprobamos si hay un camino mejor para llegar a el, y si esta en cerrados pasamos al siguiente sin más. Para decidir el camino hacemos uso de una heurística basada en la distancia del nodo actual al objetivo y del nodo origen al objetivo. Una vez que hemos llegado al nodo objetivo creamos el camino añadiendo los nodos a uno lista para el camino, simplemente empezando por el nodo objetivo, lo añadimos y despues obtenemos su nodo padre y lo añadimos y así sucesivamente hasta que lleguemos a uno nodo nulo. Y de esta forma tenemos el camino más optimo. También debemos tener en cuenta una función de coste adicional para redirigir el camino en función de los obstaculos del mapa y las defensas colocadas, en este caso queremos que se alejen de ellos mientras se dirigen al centro de recursos.

2. Incluya a continuación el código fuente relevante del algoritmo.

```
Vector3 cellCenterToPosition(int i, int j, float cellWidth, float cellHeight){
return Vector3((j * cellWidth) + cellWidth * 0.5f, (i * cellHeight) + cellHeight * 0.5f, 0);
}

void positionToCell(const Vector3 pos, int &i_out, int &j_out, float cellWidth, float
    cellHeight){
i_out = (int)(pos.y * 1.0f/cellHeight); j_out = (int)(pos.x * 1.0f/cellWidth);
}

bool operator <(const AStarNode& a,const AStarNode& b){
return(a.F<b.F);
}

bool operator >(const AStarNode& a,const AStarNode& b){
return b.F<a.F;
}

bool operator ==(const AStarNode& a,const AStarNode& b)//dos nodos distintos no pueden estar
    en la misma posicion
{
if(a.position.x==b.position.x && a.position.y==b.position.y && a.position.z==b.position.z)
    return true;
else return false;
}

bool operator !=(const AStarNode& a,const AStarNode& b)//dos nodos distintos no pueden estar
    en la misma posicion
```

```
{
if(a.position.x!=b.position.x || a.position.y!=b.position.y || a.position.z!=b.position.z)
    return true;
else return false;
}


void DEF_LIB_EXPORTED calculateAdditionalCost(float** additionalCost
, int cellsWidth, int cellsHeight, float mapWidth, float mapHeight
, List<Object*> obstacles, List<Defense*> defenses) {

        float cellWidth = mapWidth / cellsWidth;
        float cellHeight = mapHeight / cellsHeight;
        int fila,col;
        float limite=std::numeric_limits<float>::max();
        for(List<Object*>::iterator it=obstacles.begin();it!=obstacles.end();++it)
        {
                positionToCell((*it)->position,fila,col,cellWidth,cellHeight);
                additionalCost[fila][col]=limite;
        }

        for(List<Defense*>::iterator it=defenses.begin();it!=defenses.end();++it)
        {
                positionToCell((*it)->position,fila,col,cellWidth,cellHeight);
                additionalCost[fila][col]=limite;
        }
        float cost = 0;

        for(int i = 0 ; i < cellsHeight ; ++i) {
                for(int j = 0 ; j < cellsWidth ; ++j) {
                cost=limite;
                if(additionalCost[i][j]!=std::numeric_limits<float>::max())
                {
                        Vector3 cellPosition = cellCenterToPosition(i, j, cellWidth,
                            cellHeight);
                        for(List<Object*>::iterator it=obstacles.begin();it!=obstacles.end()
                            ;++it)
                {
                        cost=cost-_distance((*it)->position,cellPosition);
                }

                for(List<Defense*>::iterator it=defenses.begin();it!=defenses.end();++it)
                {
                        if(it!=defenses.begin())cost=cost-_distance((*it)->position,
                            cellPosition);
                }

                }

                additionalCost[i][j] = cost;
                }
        }
}

void DEF_LIB_EXPORTED calculatePath(AStarNode* originNode, AStarNode* targetNode
, int cellsWidth, int cellsHeight, float mapWidth, float mapHeight
, float** additionalCost, std::list<Vector3> &path) {

        int f,c;
        std::list<AStarNode*> closed;
        AStarNode* current = originNode;
        positionToCell(current->position,f,c,cellsWidth,cellsHeight);
        current->G=0;
        current->H=_distance(current->position,targetNode->position);
        current->parent=nullptr;
        current->F=current->G+current->H;+additionalCost[f][c];
        std::vector<AStarNode*> opened;
        std::make_heap(opened.begin(),opened.end());
        opened.push_back(current); std::push_heap(opened.begin(),opened.end());
        bool found=false;
        float d;
```

```
        while(!found && opened.size()>0)
        {
                current=opened.front();
                std::pop_heap(opened.begin(),opened.end()); opened.pop_back();
                closed.push_back(current);
                if(current==targetNode)found=true;
                else
                {
                for(List<AStarNode*>::iterator it=current->adjacents.begin();it!=current->
                    adjacents.end();++it)
                {
                if(find(closed.begin(),closed.end(),(*it))==closed.end())
                {
                positionToCell((*it)->position,f,c,cellsWidth,cellsHeight);

                if(find(opened.begin(),opened.end(),(*it))==opened.end())
                {

                (*it)->parent=current;
                (*it)->G=current->G+_distance(current->position,(*it)->position);
                (*it)->H=_distance((*it)->position,targetNode->position);
                (*it)->F=(*it)->G+(*it)->H+additionalCost[f][c];
                opened.push_back(*it);std::push_heap(opened.begin(),opened.end());
                }
                else
                {
                d=_distance(current->position,(*it)->position)+additionalCost[f][c];
                if((*it)->G>(current->G+d))
                {
                (*it)->parent=current;
                (*it)->G=current->G+d;
                (*it)->F=(*it)->G+(*it)->H;
                std::sort_heap(opened.begin(),opened.end());
                }
        }
}
}
}
}



                current=targetNode;
                path.push_back(targetNode->position);
                while(current->parent!=nullptr)
                {
                current=current->parent;
                path.push_front(current->position);
                }

}
```

Todo el material incluido en esta memoria y en los ficheros asociados es de mi autoría o ha sido facilitado por los profesores de la asignatura. Haciendo entrega de esta práctica confirmo que he leído la normativa de la asignatura, incluido el punto que respecta al uso de material no original.