Práctica 4. Exploración de grafos

Alejandro Serrano Fernandez
ale.serranofer@alum.uca.es
Teléfono: 640217690
NIF: 20501318S

14 de enero de 2021

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

   El algoritmo utilizado para determinar el camino que los ucos deben recorrer ha sido el A* explicado en las clases de teoría. La heurística a seguir en mi algoritmo para calcular el coste en entre un nodo y el nodo final(centro de extracción) ha sido la distacia euclídea. Destacar que, a ésta le sumamos un coste adicional si el nodo se encuentra cerca de las defensas. Por tanto escogeremos aquellas celdas que supongan un menor recorrido al centro de extracción y que están mas lejos de las defensas.

   En cuanto a las estructuras de datos, he hecho uso de un montículo para obtener los nodos con menor coste de la lista de abiertos, y dos vectores para almacenar los nodos abiertos y cerrados.

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

```
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 celda_x, celda_y;
    float valor,posicion;

    List<Defense*>::iterator currentDefense = defenses.begin();


    for(int i = 0 ; i < cellsHeight ; ++i) {
        for(int j = 0 ; j < cellsWidth ; ++j) {


            Vector3 cellPosition = cellCenterToPosition(i, j, cellWidth, cellHeight);
            float cost = 0;
            auto it = defenses.begin();
            it++;

            //Asignamos un mayor valor a aquellas posiciones que mas cerca esten de las
                defensas
            for(it; it != defenses.end(); it++)
            {
                celda_x = (((*it)->position.x - cellWidth/2) / cellWidth);
                celda_y = (((*it)->position.y - cellHeight/2) / cellHeight);

                posicion = abs(celda_x - i);
                valor -= posicion;
                posicion = abs(celda_y - j);
                valor -= posicion;
                cost = valor * 100;
            }

            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) {


//Lista de abiertos y cerrados
std::vector<AStarNode*> abiertos,cerrado;
AStarNode* current = originNode;
bool encontrado = false;
float d = 0.0f;

targetNode->parent = NULL;
current->G = 0;
current->H = _sdistance(originNode->position, targetNode->position);
current->F = originNode->G + originNode->H + additionalCost[(int)(originNode->position.y /
    cellsHeight)][(int)(originNode->position.x / cellsWidth)];

abiertos.push_back(current);
std::make_heap(abiertos.begin(),abiertos.end(),es_menor);


while(!encontrado && !abiertos.empty())
{
current = abiertos.front();
std::pop_heap(abiertos.begin(),abiertos.end(),es_menor);
abiertos.pop_back();

cerrado.push_back(current);

if(current == targetNode)
        encontrado = true;

else
{
        for(List<AStarNode*>::iterator j=current->adjacents.begin(); j != current->adjacents.
            end(); ++j)
        {
                std::vector<AStarNode*>::iterator it = std::find(cerrado.begin(),cerrado.end
                    (),(*j));

                if(it == cerrado.end())
                {
                        it = std::find(abiertos.begin(),abiertos.end(),(*j));

                        if(it == abiertos.end())
                        {
                                (*j)->parent = current;
                                (*j)->G = current->G + _sdistance(current->position, (*j)->
                                    position);
                                (*j)->H = _sdistance(current->position, (*j)->position);
                                (*j)->F = (*j)->G + (*j)->H + additionalCost[(int)((*j)->
                                    position.y / cellsHeight)][(int)((*j)->position.x /
                                    cellsWidth)];
                                abiertos.push_back((*j));
                                std::push_heap(abiertos.begin(),abiertos.end(),es_menor);

                        }
                        else
                        {
                                d = _sdistance(current->position,(*j)->position);

                                if((*j)->G > (current->G + d))
                                {
                                        (*j)->parent = current;
                                        (*j)->G = current->G + d;
                                        (*j)->F = (*j)->G + (*j)->H + additionalCost[(int)((*
                                            j)->position.y / cellsHeight)][(int)((*j)->
```

```
                                        position.x / cellsWidth)];
                                    std::make_heap(abiertos.begin(),abiertos.end(),
                                        es_menor);

                                }
                            }
                        }
                }
}
}

//Recueracion del camino
current = targetNode;
path.push_front(targetNode->position);

while(current->parent != originNode && targetNode->parent != NULL){
        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.