Práctica 4. Exploración de grafos

Francisco Jesus Diaz Zajara
francisco.diazza@alum.uca.es
Teléfono: 673540730
NIF: 44062267V

19 de enero de 2018

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

   Como estructuras necesarias utilizo 2 vectores para almacenar los nodos AStarNode* usados por el algoritmo A*, uno llamado 'abiertos' que contiene los que aun están por visitar y ser procesados, y el otro 'cerrados' con los que ya han sido visitados. También utilizamos otro nodo AStarNode* llamado current que representa al nodo actual y por último hacemos uso del montículo de la STL (make_heap) para reordenar el vector abiertos cada vez que se produzca una actualización en los parámetros de un nodo current.

   El funcionamiento del algoritmo sigue la estructura del algoritmo A*: el nodo current es expandido y miramos sus hijos, los cuales se irán guardando en el vector de abiertos para poder visitarlos si no lo estuvieran. Los nodos que estén en este último vector se usarán para comprobar si es más conveniente ir al nodo objetivo mediante el padre del nodo actual(current) o a través del propio current. Una vez un nodo ha sido visitado se guarda en el vector de cerrados para no volver a visitarlo.

   Para calcular que camino es el mejor, utilizamos un coste adicional a cada celda que consiste en calcular la distancia euclídea desde cada una de ellas hasta la celda donde esta situada la defensa principal, es decir, cuanto más lejos esté, mayor valor tendrá. Para ésto hacemos uso de una matriz de float llamada additionalCost.

   Finalmente, una vez hemos obtenido la solución, devolvemos el camino almacenándolo en la lista de Vector3 llamada 'path'.

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

```
using namespace Asedio;

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

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

    float anchoCelda = mapWidth / cellsWidth;
    float altoCelda = mapHeight / cellsHeight;

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

            Vector3 cellPosition = cellCenterToPosition(i, j, anchoCelda, altoCelda);

            //Para el coste sumamos la distancia euclidea existente entre la celda donde
             esta situada la defensa principal y la celda actual.
            std::list<Defense*>::iterator it = defenses.begin();
```

```
            additionalCost[i][j] = _distance((*it)->position, cellPosition);
        }
    }
}

bool comparar(AStarNode* i, AStarNode* j){
return (i->F > j->F);
}

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

    bool encontrado = false;
    float anchoCelda = mapWidth/cellsWidth;
    float altoCelda = mapHeight/cellsHeight;

    //Aplicamos algoritmo A*:

    AStarNode* current = originNode;
    std::vector<AStarNode*> abiertos;
    std::vector<AStarNode*> cerrados;

    current->H = _sdistance(current->position, targetNode->position);   //Distancia estimada
     entre el nodo actual(origen) y el objetivo
    current->F = current->G + current->H;
    abiertos.push_back(current);
    std::make_heap(abiertos.begin(), abiertos.end(), comparar);

    while(encontrado == false && abiertos.size() > 0){
     current = abiertos.front();
     std::pop_heap(abiertos.begin(), abiertos.end(), comparar);
     abiertos.pop_back();
     cerrados.push_back(current);

     if(current == targetNode)
     encontrado = true;
     else{
     std::list<AStarNode*>::iterator it = current->adjacents.begin();
     for(it; it != current->adjacents.end(); it++){
     if(cerrados.end() == std::find(cerrados.begin(), cerrados.end(), (*it))){
     if(abiertos.end() == std::find(abiertos.begin(), abiertos.end(), (*it))){
     int posX = (*it)->position.x/anchoCelda;
     int posY = (*it)->position.y/altoCelda;
     (*it)->parent = current;
     (*it)->G = current->G + _distance(current->position, (*it)->position)
      + additionalCost[posX][posY];
     (*it)->H = _sdistance((*it)->position, targetNode->position);
     (*it)->F = (*it)->G + (*it)->H;
     abiertos.push_back((*it));
     std::make_heap(abiertos.begin(), abiertos.end(), comparar);
     }
     else{
     float distancia = _distance(current->position, (*it)->position);
     if((*it)->G > current->G + distancia){
     (*it)->parent = current;
```

```
        (*it)->G = current->G + distancia;
        (*it)->F = (*it)->G + (*it)->H;
        std::sort_heap(abiertos.begin(), abiertos.end(), comparar);
        }
        }
        }
        }
        }
       }

       //Ahora recuperamos el camino a seguir por el UCO:

       current = targetNode;
       path.push_front(current->position);

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