Práctica 4. Exploración de grafos

Juan Manuel Grondona Nuño
juanmanuel.grondonanu@alum.uca.es
Teléfono: 656485032
NIF: 49193526E

21 de enero de 2023

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

   El algoritmo no es más que una traducción a c++ del algoritmo de la diapositiva 39 de teoría. Este es el algoritmo A* el cual se encarga de llegar al "destino" asociando cada nodo que recorre con el nodo desde el cual ha llegado. Esto, sumado a la heurística y a la distancia euclídea entre el nodo actual y el destino, nos da el camino más corto, el cual se reconstruye posteriormente.

   El algoritmo de la heurística se compone de dos partes, la primera que disminuye el valor del nodo según el número de torretas que se encuentran en su radio de acción, y la segunda, que se obtiene de la diferencia entre, la distancia entre el nodo y la torreta más lejana, y él media de la distancia de las torretas.

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);
}

bool pertenece (std::list<AStarNode*> lista, AStarNode* nodo){
    for(auto i = lista.begin(); i != lista.end(); ++i){
        if((*i) == nodo){
            return true;
        }
    }
    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;

    for(int i = 0 ; i < cellsHeight ; ++i) {
        for(int j = 0 ; j < cellsWidth ; ++j) {
            Vector3 cellPosition = cellCenterToPosition(i, j, cellWidth, cellHeight);
            float cost = 0;
            float media = 0;
            float max = 0;
            for (auto it = obstacles.begin(); it != obstacles.end(); ++it) {
                if (_sdistance((*it)->position, cellPosition) < (*it)->radio) {
                    cost -=10 * _sdistance((*it)->position, (*obstacles.begin())->position);
                }
                media += _sdistance((*it)->position, cellPosition);
                if (_sdistance((*it)->position, cellPosition) > max) {
                    max = _sdistance((*it)->position, cellPosition);
```

```
                }
            }
            cost += max - (media / obstacles.size());

            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) {

    AStarNode* current = originNode;
    std::list<AStarNode*> abiertos, cerrados;
    bool encontrado = false;

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

    abiertos.push_back(current);

    while(!encontrado && !abiertos.empty()){
        current = abiertos.front();
        abiertos.pop_front();
        cerrados.push_back(current);

        if(current == targetNode){
            encontrado = true;
        }else{
            for(auto i = current->adjacents.begin(); i != current->adjacents.end(); ++i){
                int g = current->G + _sdistance((*i)->position, (*i)->position);
                int h = additionalCost[(int)((*i)->position.y / cellsHeight)][(int)((*i)->
                    position.x / cellsHeight)];
                int f = g + h;

                if((pertenece(cerrados, (*i)) || pertenece(abiertos, (*i))) && (g < (*i)->G))
                    {
                    if(pertenece(cerrados, (*i))){
                        cerrados.remove((*i));
                    }
                    if(pertenece(abiertos, (*i))){
                        abiertos.remove((*i));
                    }
                }
                if (!pertenece(cerrados, (*i)) && !pertenece(abiertos, (*i))){
                    (*i)->G = g;
                    (*i)->H = h;
                    (*i)->F = f;
                    (*i)->parent = current;
                    abiertos.push_back((*i));
                }
            }
            abiertos.sort([](AStarNode* a, AStarNode* b){return a->F < b->F;});

        }

    }

    AStarNode* auxNode = targetNode;
    int cont = 0;
    while(auxNode->parent != nullptr){
        path.push_front(auxNode->position);
        auxNode = auxNode->parent;
    }

}
```

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.