Práctica 4. Exploración de grafos

Adrian Reyes Alba
adrian.reyesalba@alum.uca.es
Teléfono: 647243014
NIF: 49074565V

29 de enero de 2018

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

   He seguido el algoritmo A*. Tenemos dos vectores donde se van a guardar los nodos que esten abiertos(estará aqui hasta que no recorramos) o cerrados(cuando pasemos por él), y inicializamos la celda inicial con los valores de G, H, F y el nodo padre que será nulo e incluyéndolo en la lista de nodos abiertos. Para la solución de la heuristica he usado la distancia euclídea. nodos de la lista de nodos abiertos. El algoritmo sacará el nodo con menor coste y compararía si ese nodo es el nodo destino, si es destino ya tendríamos solución y finalizaría, y encaso contrario miraríamos lista de nodos adyacentes y calcularía sus valores(F, G, H, nodo padre). Estos nodos pasarían a la lista de nodos abiertos. En el caso de que un nodo ya esté en la lista, se calcularía el coste del camino por el actual y se compararía con el que tenía. Si es mayor no hace nada y si es menor se actualiza. Una vez encontremos una solución correcta, o nos quedemos sin poder encontrarla procederemos a recuperar el camino.

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

```
bool comparar(AStarNode* A, AStarNode* B){
    if(A->F < B->F)
return true;
    else
return false;
}

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 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;
            if( (i+j) % 2 == 0 ) {
                cost = cellWidth * 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) {
float cellWidth = mapWidth / cellsWidth;
float cellHeight = mapHeight / cellsHeight;
    std::vector<AStarNode*> abierto, cerrado;
    AStarNode* current = originNode;
    current->H = _sdistance(current->position,targetNode->position);
    current->F = current->G+current->H;
    abierto.push_back(current);
    std::make_heap(abierto.begin(),abierto.end(),comparar);
    bool found = false;

    while(!found && abierto.size() > 0){
        current=abierto.front();
        std::pop_heap(abierto.begin(),abierto.end(),comparar);
        abierto.pop_back();
        cerrado.push_back(current);

        if(current == targetNode)
            found = true;
        else{
            for(List<AStarNode*>::iterator it=current->adjacents.begin(); it != current->adjacents.end
                if(cerrado.end()==std::find(cerrado.begin(),cerrado.end(),(*it))){
                    if(abierto.end()==std::find(abierto.begin(),abierto.end(),(*it))){
                        int cx = (*it)->position.x/cellWidth;
                        int cy = (*it)->position.y/cellHeight;
                        (*it)->parent = current;
                        (*it)->G = current->G + _distance(current->position,(*it)->position) + additio
                        (*it)->H = _sdistance((*it)->position,targetNode->position);
                        (*it)->F = (*it)->G + (*it)->H;
                        abierto.push_back(*it);
                        std::make_heap(abierto.begin(),abierto.end(),comparar);
                    }
                    else{
                        float d = _distance(current->position,(*it)->position);
                        if((*it)->G > current->G + d){
                            (*it)->parent = current;
                            (*it)->G = current->G + d;
                            (*it)->F = (*it)->G + (*it)->H;
                            std::sort_heap(abierto.begin(),abierto.end(),comparar);
                        }
                    }
                }
            }
        }
    }
    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.