Práctica 4. Exploración de grafos

Iván Alba Gómez
ivan.albagomez@alum.uca.es
Teléfono: XXXXXXXXX
NIF: 49302616T

27 de diciembre de 2021

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

   Para el cálculo de caminos mínimos hemos usado el algoritmo A*, el cual es un camino heurístico, ya que una de sus principales características es que hace uso de una función de evaluación heurística mediante la cual etiqueta los diferentes nodos de la red y servirá para determinar la probabilidad de dichosnodos de pertenecer al camino óptimo. Mi implementación consta de las siguientes estructuras:

   - cur: Puntero a AStarNode que apunta al nodo que estamos procesando.
   - G[i]: Distancia del camino más corto entre el origen y el nodo i.
   - H[i]: Distancia estimada del cmino más corto entre el nodo i y el destino.
   - F[i]: G[i] + H[i]
   - closed: Vector de nodos procesados.
   - opened: Vector de nodos pendientes.
   - additionalCost: Matriz dinámica de tipo float que almacena el coste adicional de cada celda.

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

```
// CALCULO DEL COSTE ADICIONAL:

    List<Defense*>::iterator itDefense = defenses.begin(); itDefense++;
    List<Object*>::iterator itObstacle = obstacles.begin();

    for(int i = 0; i < cellWidth; i++) {
        for(int j = 0; j < cellHeight; j++) {
            Vector3 cellPosition = cellCenterToPosition(i, j, cellWidth, cellHeight);
            float cost = 1000;
            while(itDefense != defenses.end()) {
                if(_distance(cellPosition, (*itDefense)->position) < 10) {
                    cost *= 2;
                } else {
                    cost /= 2;
                }
                itDefense++;
            }
            while(itObstacle != obstacles.end()) {
                if(_distance(cellPosition, (*itObstacle)->position) < 10) {
                    cost *= 2;
                } else {
                    cost /= 2;
```

```
                }
                itObstacle++;
            }
            additionalCost[i][j] = cost;
        }
    }

// ALGORITMO A*:

    std::vector<AStarNode*> opened, closed;
    std::make_heap(opened.begin(), opened.end());
    AStarNode* cur = originNode;
    cur->G = 0;
        cur->H = additionalCost[(int)(cur->position.y / cellsHeight)][(int)(cur->position
        cur->parent = NULL;
        cur->F = cur->G + cur->H;
    opened.push_back(cur);
    std::push_heap(opened.begin(), opened.end(), comp);
    bool found = false;
    while(!false && !opened.empty()) {
        std::pop_heap(opened.begin(), opened.end(), comp);
        cur = opened.back();
        opened.pop_back();
        closed.push_back(cur);
        if(cur == targetNode) {
            found = true;
        } else {
            for(std::list<AStarNode*>::iterator j = cur->adjacents.begin(); j != cur->adj
                if(std::find(closed.begin(), closed.end(), *j) == closed.end()) {
                    if(std::find(opened.begin(), opened.end(), *j) == opened.end()) {
                        (*j)->parent = cur;
                        (*j)->G = cur->G + _distance(cur->position, (*j)->position);
                        (*j)->H = additionalCost[(int)((*j)->position.y / cellsHeight)][(
                        (*j)->F = (*j)->G + (*j)->H;
                        opened.push_back(*j);
                        std::push_heap(opened.begin(), opened.end(), comp);
                    } else {
                        float d = _distance(cur->position, (*j)->position);
                        if((*j)->G > cur->G + d) {
                            (*j)->parent = cur;
                                                    (*j)->G = cur->G + d;
                                                    (*j)->F = (*j)->G + (*j)->H;
                            std::make_heap(opened.begin(), opened.end());
                        }
                    }
                }
            }
        }
    }

// RECUPERACION DEL CAMINO:

    cur = targetNode;
    path.push_front(targetNode->position);
    while(cur->parent != originNode && cur->parent != nullptr) {
        cur = cur->parent;
```

```
        path.push_front(cur->position);
    }

// FUNCIONES ADICIONALES:

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

    bool comp(AStarNode* n1, AStarNode* n2) {
        return n1->F > n2->F;
    }
```

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.