Práctica 4. Exploración de grafos

Jesús Rodríguez Heras
jesus.rodriguezheras@alum.uca.es
Teléfono: 628576107
NIF: 32088516C

3 de junio de 2019

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

   El algoritmo que busca el camino para llegar al centro de extracción de minerales sigue el esquema del algoritmo A* y contiene las siguientes estructuras:

   - `std::vector<AStarNode*> opened`**:** Vector de tipo AStarNode que llevará el control de los nodos abiertos.
   - `std::vector<AStarNode*> closed`**:** Vector de tipo AStarNode que llevará el control de los nodos ya visitados por el algoritmo.
   - `AStarNode* current`**:** Objeto de la clase `AStarNode` declarada en `Asedio.h` que almacena el nodo actual que estamos evaluando.

   El algoritmo A* es un algoritmo de búsqueda que puede ser empleado para el cálculo de caminos. Se trata de un algoritmo heurístico, ya que usa una función de evaluación heurística, $f(n)$, mediante la cual etiquetará a los diferentes nodos de la red y servirá para determinar la probabilidad de dichos nodos de pertenecer al camino óptimo.

   Esta función de evaluación está compuesta a su vez por otras dos funciones:

   - $g(n)$**:** Indica la distancia actual desde el nodo origen hasta el nodo a evaluar.
   - $h(n)$**:** Expresa la distancia estimada desde el nodo a evaluar hasta el nodo destino al que se pretende encontrar el camino mínimo.

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);
}

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;

    std::list<Defense*>::iterator it = defenses.begin();

    for(int i = 0 ; i < cellsHeight ; ++i) {
        for(int j = 0 ; j < cellsWidth ; ++j) {
            Vector3 cellPosition = cellCenterToPosition(i, j, cellWidth, cellHeight);
            additionalCost[i][j] = _sdistance(cellPosition, (*it)->position);
        }
    }
}
```

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

    int maxIter = 100;
    AStarNode* current = originNode;
    std::vector<AStarNode*> opened;
    std::vector<AStarNode*> closed;
    bool encontrado = false;
    float cellWidth = mapWidth / cellsWidth;
    float cellHeight = mapHeight / cellsHeight;

    opened.push_back(current);

    while (current != targetNode && !opened.empty() && encontrado == false) {
        current = opened.front();
        opened.erase(opened.begin());
        closed.push_back(current);

        if (current == targetNode) {
            encontrado = true;
        }else{
            int i, j;
            float d;
            std::list<AStarNode*>::iterator iter = current->adjacents.begin();
            while (iter != current->adjacents.end()) {
                if (closed.end() == std::find(closed.begin(), closed.end(), (*iter))) {
                    if (opened.end() == std::find(opened.begin(), opened.end(), (*iter))) {
                        i = (*iter)->position.x / cellWidth;
                        j = (*iter)->position.y / cellHeight;
                        (*iter)->G = current->G + _distance(current->position, (*iter)->
                            position) + additionalCost[i][j];
                        (*iter)->H = _sdistance((*iter)->position, targetNode->position);
                        (*iter)->F = (*iter)->G + (*iter)->H;
                        (*iter)->parent = current;

                        opened.push_back(*iter);
                    }else{
                        d = _distance(current->position, (*iter)->position);
                        if ((*iter)->G > current->G + d) {
                            (*iter)->G = current->G + d;
                            (*iter)->F = (*iter)->G + (*iter)->H;
                            (*iter)->parent = current;
                        }
                    }
                    std::sort(opened.begin(), opened.end());
                }
                ++iter;
            }
        }
    }

    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.