Práctica 4. Exploración de grafos

Antonio Roldan Andrade
antonio.roldanandrade@alum.uca.es
Teléfono: +34611404497
NIF: 49562495W

13 de enero de 2022

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

   El algoritmo que usaremos sera el algoritmo de busqueda de soluciones A*, el cual nos permite encontrar soluciones mediante comparaciones entre nodos, usando para eso la funcion f(n)= h(n) + c(n), donde:

   h(n)= Representa el valor heuristico dado a ese nodo, en nuestro caso para otorgar dicho valor usaremos la funcion Heuristica$_M anhattan, lacualnospermiteencontrarcomodelejossehallaunnododelnodoobjetivo.$

   c(n)= Representa el coste necesario para transladarnos de un nodo A a un nodo B.

   Para el algoritmo A* tendremos que usar 3 variables (ademas del resto de auxiliares), las cuales seran:

   Actual =¿ El cual nos sirve como iterador, e irá almacenando el nodo que estamos comprobando, partiendo del nodo inicial/origen.

   Abiertos=¿ Iremos guardando en esta lista que nodos se encuentran abiertos y listos para ser explorados, para obtener nuevos nodos hijos, etc.

   Cerrados=¿ Guardaremos en esta lista los nodos ya visitados y que ya han sido explorados.

   Para un correcto funcionamiento iremos ordenando la lista de abiertos, para reducir el tiempo necesario para encontrar el f(n) de menor valor, para esto he usado la clase monticulo proporcionada por C++ de la std. Al usar esto se requiere de una sobrecarga del operador (), ya que sin este el monticulo no podrá realizar una correcta ordenación de los nodos, dando lugar a un mayor tiempo de ejecución y demas problemas.

   Tras la realizacion del algoritmo A* , necesitamos recuperar el camino a seguir para obtener la solución buscada, para esto usamos la funcion recupera$_c aminos, estarecibeelnodoinicialyelnodofinal.$

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

```
std::pair<int,int> get_x_y(const Vector3 pos, float cellWidth, float cellHeight)
{
    //i_out = (int)(pos.y * 1.0f / cellHeight);
    //j_out = (int)(pos.x * 1.0f / cellWidth);

    return std::make_pair((int)(pos.x * 1.0f / cellWidth), (int)(pos.y /cellHeight));

}

//Nos sirve para comprobar si el nodo esta o no en la lista de visitados
bool existe_nodo(AStarNode *nodo, std::vector<AStarNode *> nodos)
{
    bool token = false;

    for(auto i = nodos.begin(); i != nodos.end() && !token; ++i){

        if (nodo == *i)
```

```
            token = true;

    }

    return token;
}

std::list<Vector3> recupera_camino(AStarNode *begin, AStarNode *destino)
{
    AStarNode *actual = destino;
    std::list<Vector3> path;

    //Insertamos la posicion del destino ya que para la obtencion del camino se recorre e
    path.push_back(destino->position);

    //Recorremos en orden inverso
    for(;actual->parent != begin && destino->parent != NULL;actual=actual->parent)
        path.push_front(actual->position);

    return path;
}

int Heuristica_Manhattan(AStarNode *begin, AStarNode *target, float cellWidth, float cell
{
    int o_x, o_y, d_x, d_y, distancia = 0;
    std::pair<int, int> parser;

    //Coordenadas de origen
    parser = get_x_y(begin->position, cellWidth, cellHeight);
    o_x = parser.first; o_y = parser.second;

    //Coordenadas de destino
    parser = get_x_y(target->position, cellWidth, cellHeight);
    d_x = parser.first; d_y = parser.second;

    distancia = abs(o_x - d_x) + abs(o_y - d_y);
    return distancia;
}

struct sort_monticulo_nodo
{
    //Sobrecargamos el operador () para que el monticulo pueda ordenarlos correctamente
    //En funcion de la funcion f(n) = h(n) + coste(n)
    bool operator()(AStarNode* a, AStarNode* b)
    {
        return a->F > b->F;
    }
};



void DEF_LIB_EXPORTED calculatePath(AStarNode *originNode, AStarNode *targetNode, int cel
{
    /* ------------------------------------ VARIABLES ------------------------------------ */
    std::pair<int, int> p;
```

```
int maxIter = 100;

AStarNode *actual = originNode;

std::vector<AStarNode *> abiertos, cerrados;

bool token = false;

std::list<AStarNode *>::iterator it;

float distancia=0;

float cellWidth = mapWidth / cellsWidth;

float cellHeight = mapHeight / cellsHeight;

/* ---------------------------------- ALGORITMO ---------------------------------- */

targetNode->parent = NULL;

actual->G = 0;
actual->H = Heuristica_Manhattan(actual, targetNode, cellWidth, cellHeight);
//Obtener posiciones dentro de la matriz
p = get_x_y(actual->position,cellWidth, cellHeight);
actual->F = actual->G + actual->H + additionalCost[p.first][p.second];

abiertos.push_back(actual);

while (!token && !abiertos.empty())
{
    actual = abiertos.front();

    std::pop_heap(abiertos.begin(), abiertos.end(), sort_monticulo_nodo());

    abiertos.pop_back();
    cerrados.push_back(actual);

    if (actual == targetNode)
    {
        token = true;
    }
    else
    {
        it = actual->adjacents.begin();
        while (it != actual->adjacents.end())
        {
            if (!existe_nodo((*it), cerrados))
            {
                if (!existe_nodo((*it), abiertos))
                {
                    (*it)->parent = actual;
                    (*it)->G = actual->G + _distance(actual->position, (*it)->positio
                    (*it)->H = Heuristica_Manhattan((*it), targetNode, cellWidth, cel

                    p = get_x_y((*it)->position,cellWidth, cellHeight);
                    (*it)->F = (*it)->G + (*it)->H + additionalCost[p.first][p.second
```

```
                        abiertos.push_back((*it));
                        std::push_heap(abiertos.begin(), abiertos.end(), sort_monticulo_n
                    }
                    else
                    {
                        distancia = _distance(actual->position, (*it)->position);
                        if ((*it)->G > actual->G + distancia)
                        {
                            (*it)->parent = actual;
                            (*it)->G += distancia;
                            (*it)->F = (*it)->G + (*it)->H + + additionalCost[p.first][p.
                            p = get_x_y((*it)->position,cellWidth, cellHeight);
                            std::make_heap(abiertos.begin(), abiertos.end(), sort_monticu
                        }
                    }
                }
                it++;
            }
        }
    }
    path = recupera_camino(originNode, targetNode);
}
```

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.