Práctica 4. Exploración de grafos

Ignacio Rodriguez Perez
ignacio.rodriguezperez@alum.uca.es
Teléfono: +34 697 469 718

15 de enero de 2021

1. Comente el funcionamiento del algoritmo y describa las estructuras necesarias para llevar a cabo su implementación.
   Usando el algoritmo A* explicado en clase, vamos buscando un camino para que, dado un estado (nodo) inicial, podamos llegar a un objetivo.

   Todo esto lo llevamos a cabo de la siguiente forma (explicación de dicho algoritmo) :

   Tenemos dos listas de nodos, una para indicar los nodos que ya hemos visitados ("closed" es decir, cerrados), y otra para indicar los que quedan pendientes por visitar.
   Dicho nodo inicial empieza siendo añadido a esta segunda lista, por lo que, en el inicio del primer bucle (while) tomamos el primer nodo de dicha lista, lo añadimos a la lista de nodos cerrados/ya procesados y lo sacamos de la lista de abiertos, ya que lo estamos procesando.
   Ahora, si no pertenece a la lista de cerrados (porque se puede dar el caso de que se llegue a este estado por otro camino) lo procesamos, pudiendose dar dos casos, que esté o no en la lista de abiertos. Si está en la lista de abiertos (else) vemos si con el camino que llegamos, da mejor resultado, y si no está en dicha lista, se actualizan los valores de dicho nodo.

   Una vez se llega a su destino, o nos quedamos nodos que procesar, deshacemos el camino recorrido, actualizando el valor del padre de cada nodo.

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, cellHeight = mapHeight / cellsHeight, aux;
    for(int i = 0 ; i < cellsHeight ; ++i)
        for(int j = 0 ; j < cellsWidth ; ++j)
        {
            for(auto& o: obstacles)
                if(aux = _distance(cellCenterToPosition(i,j,cellWidth,cellHeight),o->position) >= o->radio)
                    additionalCost[i][j]*=(aux*o->radio);
            for(auto d: defenses)
                if(aux = _distance(cellCenterToPosition(i,j,cellWidth,cellHeight),d->position) >= d->radio)
                    additionalCost[i][j]*=(aux*d->radio*d->range*d->range*0.3333f);
        }
}

bool belongs(std::list<AStarNode*> list2find, AStarNode* node)
{ return (std::find(list2find.begin(),list2find.end(), node) == list2find.end()); }

float estimatedDistance(AStarNode* c, AStarNode* t, float** ac, float ch, float cw)
{ return _sdistance(c->position, t->position) + ac [(int)(c->position.y / ch)] [(int)(c->position.x / cw)]; }

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, cellHeight = mapHeight / cellsHeight;

    std::list<AStarNode*> open, closed; AStarNode* cur = originNode;
    cur->G=0; cur->H = estimatedDistance(cur, targetNode, additionalCost, cellHeight, cellWidth);
    cur->F=cur->G+cur->H;
    open.push_back(cur);
    while(cur != targetNode && open.size()>0)
    {   cur = *open.begin(); open.pop_front();
        closed.push_back(cur);
        for (auto j : cur->adjacents )
            if(j != NULL && !belongs(closed, j))
            {   if(!belongs(open, j))
                {
                    j->parent = cur; j->G = cur->G + _sdistance(cur->position, j->position);
                    j->H = estimatedDistance(j, targetNode, additionalCost, cellHeight, cellWidth);
                    j->F = j->G + j->H;
                    open.push_back(j);
                }
                else
                {   float d = _sdistance(cur->position, j->position);
                    if(j->G > cur->G + d)
                    {   open.erase(std::find(open.begin(),open.end(), j));
                        j->parent = cur; j->G = cur->G + d; j->F = j->G + j->H;
                        open.push_back(j);
                    }
                }
            }
    }
    cur = targetNode;
    while ( cur != NULL && cur->parent != originNode)
        path.push_front(cur->position),
        cur = cur->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.