Práctica 4. Exploración de grafos

Lucía Atienza Olmo
lucia.atienzaolmo@alum.uca.es
Teléfono: 646767043
NIF: 32089267N

29 de enero de 2022

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

   Para poder aplicar el algoritmo A*, necesitamos una lista de nodos abiertos y otra de nodos cerrados. Para ambas he usado el contenedor de la biblioteca estándar de C++ std::vector. Esto es así por eficiencia y para poder utilizar montículos. Debido a que el montículo de la biblioteca estándar es un montículo de máximos, he necesitado la función min:

   ```
   bool min(const AStarNode* nodo1, const AStarNode* nodo2)
   {
       return nodo2->F < nodo1->F;
   }
   ```

   para poder hacer un montículo de mínimos.

   Comenzamos el algoritmo desde el nodo origen, el cual introducimos en la lista de abiertos. Mientras no hayamos llegado al nodo objetivo y nos queden nodos disponibles en la lista de abiertos:

   - Escogemos el nodo más prometedor sacándolo del vector de abiertos y lo introducimos en cerrados.
   - Si hemos llegado al objetivo, paramos de buscar.
   - En caso de no haber llegado al objetivo, recorremos la lista de adyacencia del nodo actual. Para cada uno de los nodos que se encuentran en la lista de adyacencia, comprobamos primeramente que no se encuentre en la lista de cerrados (en caso de encontrarlo, no hacemos nada). Posteriormente buscamos en la lista de abiertos. Ahora tenemos dos posibilidades, que el nodo se encuentre en la lista de abiertos o que no.
     - El nodo no se encuentra en abiertos: establecemos los valores correspondientes al mismo y lo incluimos en la lista de abiertos.
     - El nodo se encuentra en abiertos: en este caso, actualizaremos el valor del nodo si, pasando por el nodo al cual estamos mirando la lista de adyacencia, el valor del nodo sea mejor que su valor actual.

   Una vez hecho esto, debemos recuperar el camino. El camino solo se debe recuperar en caso de haber llegado al objetivo. Vamos incluyendo en la lista path desde el targetNode hasta el originNode haciendo uso del puntero al padre disponible en cada nodo.

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

   ```
   void DEF_LIB_EXPORTED calculatePath(AStarNode* originNode, AStarNode* targetNode
                       , int cellsWidth, int cellsHeight, float mapWidth, float mapHeight
                       , float** additionalCost, std::list<Vector3> &path)
   {
       bool found = false;
       std::vector<AStarNode*> closed, opened;
       AStarNode* cur = originNode;
       int x, y;
       positionToCell((cur)->position, x, y, cellsWidth, cellsHeight);
       cur->H = estimatedDistance(originNode, targetNode, x, y, additionalCost);
   ```

```
    cur->G = 0;
    cur->F = cur->G + cur->H;
    opened.push_back(cur);
    std::make_heap(opened.begin(), opened.end(), min);
    while(!found && opened.size() > 0) //mientras no se encuentre solucion y queden nodos
        disponibles
    {
        std::pop_heap(opened.begin(), opened.end(), min);
        cur = opened.back();
        opened.pop_back();
        closed.push_back(cur);
        if(iguales(cur, targetNode))
            found = true;
        else
        {
            List<AStarNode*>::iterator j;
            for(j = cur->adjacents.begin(); j != cur->adjacents.end(); ++j)
            {
                if(std::find_if(closed.begin(), closed.end(), isEqualTo(*j)) == std::end(
                    closed)) //no se encuentra en cerrados
                {
                    if(std::find_if(opened.begin(), opened.end(), isEqualTo(*j)) == std::end(
                        opened)) //no se encuentra en abiertos
                    {
                        (*j)->parent = cur;
                        (*j)->G = cur->G + _distance(cur->position, (*j)->position);
                        int row,col;
                        positionToCell((*j)->position, row, col, cellsWidth, cellsHeight);
                        (*j)->H = estimatedDistance(*j, targetNode, row, col, additionalCost)
                            ;
                        (*j)->F = (*j)->G + (*j)->H;
                        opened.push_back(*j);
                        std::push_heap(opened.begin(), opened.end(), min);
                    }
                    else //se ha encontrado en abiertos
                    {
                        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::sort_heap(opened.begin(), opened.end(), min);
                        }
                    }
                }
            }
        }
    }
    if(found) //si hemos encontrado solucion, recuperamos camino
    {
        cur = targetNode;
        path.push_front(targetNode->position);
                while(cur->parent != originNode)
                {
                        cur = cur->parent;
                        path.push_front(cur->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.