A* Implementation

Introduction

The main task for this assignment is to research and implement the **A* pathfinding algorithm**, testing it using models of real life environments – for example, the third floor of the MHT building at the university.

A* was developed in 1968 in order to combine heuristic (i.e. "guess which direction to go") and formal (i.e. "find the shortest path to the goal that works") approaches to get the most accurate and efficient compromise between the two.

In order to do this, it combines the efficiency of **Dijkstra's algorithm** (in which every path is checked and the shortest path is calculated) with the information from **best-first search** (in which the algorithm moves in the direction it thinks is best) in order to minimize the amount of time it takes, creating a work around for one of Dijkstra's biggest weaknesses.

To do this, it takes the starting position and calculates the cost of moving to any adjacent nodes, taking into account how many **nodes** it's moved already to get there from the starting position and the best approximate distance to the goal (the "**heuristic**"). It can be adjusted to take other things into account too (such as different terrain types, moving obstacles, etc.) but in its simplest implementation it doesn't need to.

Methods

```
template <class T> LinkedList
```

```
Node<T>* first;
int size;
```

```
LinkedList();
~LinkedList();

void push(T value);
void insertAt(int index, T value);
void remove(T value);
void removeAt(int index);
void clear();

int search(T value);
Node<T>* getAddress(T value);
Node<T>* getAddressAt(int index);

void write();
```

First of all for the LinkedList class I required some way of **knowing how many nodes were currently in the list**, and to have **a pointer to the first item in the list** (that's potentially null). The pointer was simple enough (Node<T>* first), but instead of going the suggested route of manually counting the nodes one by one whenever the size is needed I went for a variable (int size) that's updated whenever nodes are removed or added. This saves on processing speed overall, since I figured I'd need to know the size of the list often for validation purposes.

After the standard constructor and destructor (with the destructor going through the nodes one by one and deleting them from memory), I created several methods for inserting nodes. The simplest is the **push** method that adds a node to the end of the list, followed by the **insertAt** method that gives better control over exactly where the node will be added relative to the others. The **remove** method likewise removes any node that has the same value, and the **removeAt** method removes the node from the specified index instead. The **search** method returns the index of any node that matches the input value (-1 if none matched), and the **getAddress** and **getAddressAt** methods return the actually address in memory of the node. The **write** method just attempts to output the contents of the linked list to the console.

```
template <class T> Node
```

```
T data;
Node* previous;
Node* nextNode;
```

```
Node();
```

This is simple enough – it holds a non-specific data type in 'data', and points to the previous and next nodes within the linked list it's a part of (although it has no direct connection to the linked list itself).

For Office Use Only

Map
bool* data; int width, height;

Map();
~Map();
 	
void write();
 	
bool operator()(int x, int y);

The **Map** structure contains all data specific to the maps, which ultimately boils down the size of it (int width, height) and whether each part of the map is walkable or not (which I restricted to two values – false for walkable, true for impassable).

The **write** method simply writes that data to the console for debugging purposes, 0s for false and 1s for true. The **operator()** method however is a bit more complicated – since I knew I'd be having to check specific parts of the map at several points, I overloaded this operator in order to make it easier. You simply pass it the x and y co-ordinates you want to check, and it returns true or false.

MapPos
int X, Y; int G, H, F; int parentX, parentY;

MapPos();
MapPos(int x, int y);
MapPos(int x, int y, int g, int h);
 	
void setParent(MapPos p);
 	
void operator=(const MapPos& b);
bool operator==(const MapPos& b);

In order to implement A* (see below for the specifics), nodes that are checked require the calculation and storage of ***scores***, i.e. the cost of moving to those nodes. At the end, the shortest path is also determined by checking the designated parents of each node. Because of all this extra information, I decided to create the **MapPos** structure, holding all of the relevant information.

The method **setParent** is pretty self-explanatory (you pass it another MapPos object and it stores their co-ordinates – not all of the details, just the co-ordinates, since the other details are subject to change). The **operator=** and **operator==** overloaded functions however were done for two purposes – to ensure that deep copying is done with lines like 'A = B', and to ensure that MapPos objects compared to each other are seen as the same when only their X and Y values match (this is to benefit the score-comparisons used in the algorithm, where the same node can be in the open list and in the adjacent squares list but with differing G values).

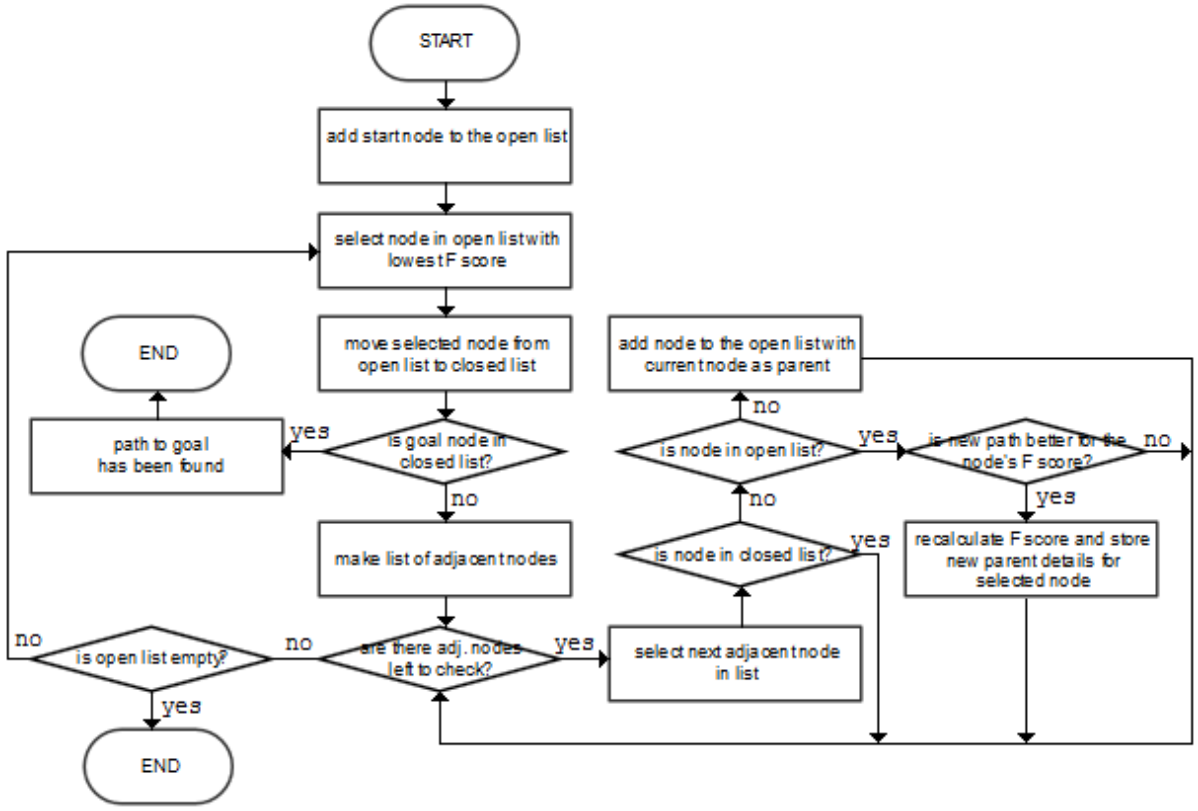
```
Map* LoadMap(const char* filePath);
```

This method simply loads in the data from a text file, first the width and height followed by the actual terrain information (in the form of 1s and 0s). There's nothing else to it, other than that it returns a pointer to the Map in heap memory rather than returning it by value.

```
LinkedList<MapPos>* findPath(Map& map, int startX, int startY,
                                       int goalX, int goalY   );
```

The method being used for calculating the path will take three parameters – a start position, a goal position, and a reference to the Map object being used. It then returns a new pointer to a LinkedList in memory, containing every step (in the form of MapPos objects) of the final calculated path.

The following flow chart details exactly what the method does, which is just a standard implementation of the A* pathfinding algorithm (with no terrain differences taken into account, just passable and impassable nodes):



Flowchart created using Gliffy, a free online tool for diagram creation (Gliffy, Inc., 2014).

Pseudo-Code

The pathfinding algorithm works as follows, and should be implemented the same way (with C++ specific syntax, of course):

```
LinkedList<MapPos> openList, closedList
LinkedList<MapPos> finalList
add start position to the open list

while open list is not empty
      select node in open list with lowest F score
      add selected node to the closed list
      remove selected node from the open list

      if goal square is in the closed list
           break out of while loop

      make list of adjacent nodes to selected node
      foreach adjacent node in list
         if node is not walkable
            remove from list
         else
            if node is in the closed list
               skip to next node
            else if node is not in the open list
               make selected node the parent of adj. node
               add node to the open list
            else if node is already on the open list
               if path from selected node to node is better
                  set parent of node to selected node

if goal square was added to the closed list
      select goal node and add to final list
      while start node is not on the final list
         select parent of selected node
         add selected node to final list
      return final list

else return null pointer
```

For Office Use Only

Results

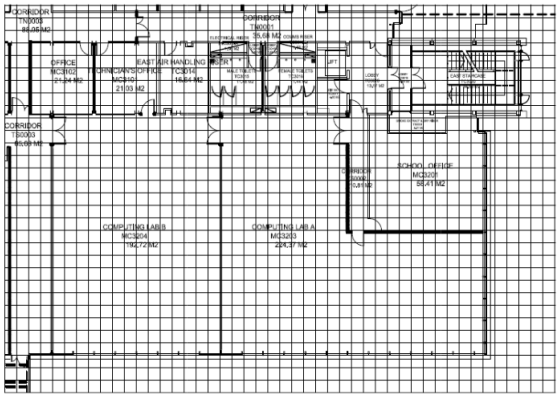


Figure 1 – Provided Map In Brief

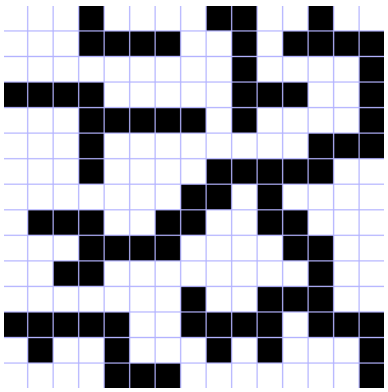


Figure 2 – Original Map

In order to test the algorithm, I made use of a converted version of the map provided (figure 1) as well as a more complex map of my own design (figure 2) to test its limitations.

The map on the left converted into a 42x31 binary representation, and the map I created myself was 15x15 (focusing on more dense complexity rather than the greater scale the provided map gave).

The text files for both of these maps is provided in the source code, named "**testmap1.txt**" and "**testmap2.txt"**, respectively, in which the binary representation required for the program is held along with the width and height of the map (also required by the program).

Test Description	Passed Values	Expected Result	Real Result	Pass
The shortest path is found given no obstacles (straight line)	*testmap1.txt* start = (1 , 1) goal = (10 , 1)	9 moves total (9x right)	9 moves total (9x right)	**Y**
The shortest path is found given no obstacles (y-value difference)	*testmap1.txt* start = (5 , 10) goal = (15 , 16)	16 moves total (10x right, 6x down)	16 moves total (10x right, 6x down)	**Y**
The shortest path is found given obstacles (around simple wall)	*testmap1.txt* start = (15 , 13) goal = (19 , 13)	10 moves total (3x up, 3x down, 4x right)	10 moves total (3x up, 3x down, 4x right)	**Y**
The shortest path is found given obstacles (around 2 simple walls, concave obstacles)	*testmap1.txt* start = (15 , 13) goal = (30 , 13)	21 moves total (3x up, 3x down, 15x right)	21 moves total (3x up, 3x down, 15x right)	**Y**
No path is found, when there is no way to get to goal from the start	*testmap2.txt* start = (1 , 1) goal = (0, 14)	No path is found	No path is found	**Y**
Concave obstacles are traversed out of to get to the goal correctly	*testmap2.txt* start = (9 , 8) goal = (11 , 0)	Path was found without error	Path was found without error, and was the shortest possible path there	**Y**

Extensions

Graphical Interface w/ FreeGLUT

For my extension I decided to create a graphical output window for the pathfinding algorithm that would be able to visually display both the final path and the complete search evolution.

In order to do this I made use of FreeGLUT, as I'm already fairly familiar with how OpenGL works and it's quick and easy to set up. The program reads in the text file, as before, but now displays the data as coloured squares.

White squares mean the area is passable, black squares mean it isn't. The goal position is displayed using a yellow square, the goal position with a slightly darker shade of yellow, and the path itself used a gray. Keeping all these colours distinguishable makes it easier for you to see exactly what's going on.

Left clicking a position will move the start position, and right clicking will move the goal position. When one of these is done, the path is recalculated, and if you hold the CONTROL key while you do so then the evolution of the search is also displayed. You can also edit the map itself from within the window, with the middle mouse button being used to toggle the state of individual co-ordinates (from passable to solid and vice versa), and pressing 'C' will clear the map entirely of solid nodes. Similarly, the 'R' button reloads the map from the text file.

This extension was easy to create, with FreeGLUT being made specifically for situations like this – where heavy graphical capability is not necessary, and a quick setup is more important. In the code all that happens when the program runs is that FreeGLUT initializes itself, links a couple of methods for detecting user input, and then draws the screen and information until the program closes (which happens when you press the ESCAPE key).

The program should be able to run on any computer that has OpenGL installed on it, due to the .dll being all that's necessary to have in the same directory as the executable.

References

Amit Patel (2014) *Introduction to A** [online]
Available from: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
[Accessed 17th March 2014]

Gliffy, Inc. (2014) *Online Diagram Software and Flowchart Software – Gliffy* [online]
Available from: https://www.gliffy.com/
[Accessed 4th April 2014]