Data Structures and Algorithms Assignment

Implementation and visualisation of A* algorithm

Due date: Wednesday 6th February.

Deliverables

Final project build to be committed to your Data Structures & Algorithms github account before the end of Wednesday 6th February. You will be required to demonstrate your project to me in a subsequent lab.

Note: Plagiarism is a very serious offence and will be treated accordingly.

Attendance

You will be permitted four two-hour lab slots on Thursday morning's class to work on this project. The dates are:

- 6th December
- 13th December
- 17th January
- 31st January

Attendance is mandatory during these periods and you must make regular github commits of your project during this time. You will need to contact me if you are unable to make any of these sessions. If you fail to attend one of more sessions you will lose up to 50% of your final mark. Please see the marking scheme at the end of this document for further information.

Part 1

Implement the A* algorithm described in Chapter 3: Lecture 17 (slides are available on blackboard). This should be added as a new member function to the Graph class as follows:

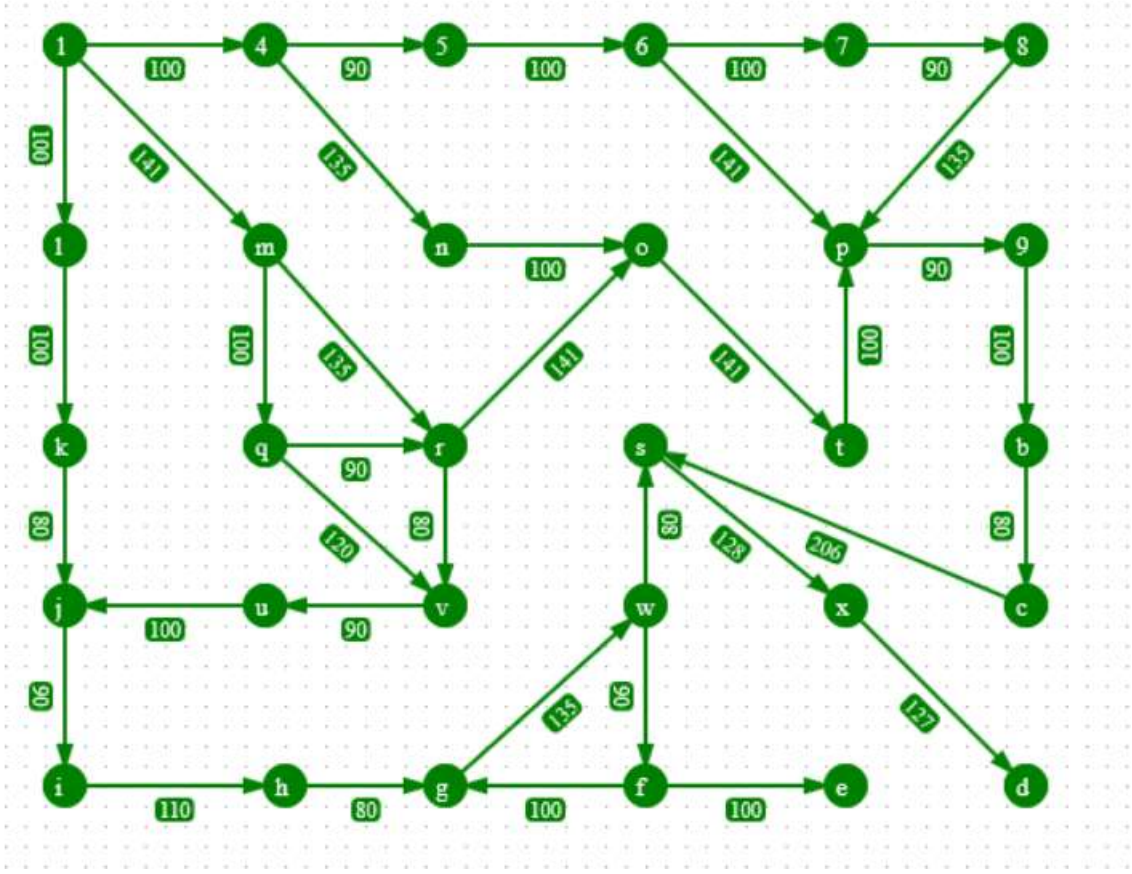
```
void aStar( Node* start, Node* dest, std::function<void(Node *)> f_visit,
            std::vector<Node *>& path );
```

where:

`start` is a Graph node indicating the origin of the search
`dest` is the goal vertex indicating the destination of the search
`std::function<void(Node *)> f_visit` is a function that shows the node currently being expanded.
`path` is a container (a vector) which holds the best path generated by the algorithm's completion

Note that each node now stores three pieces of information: a city name along with values for:
h(*n*) = estimated cost to the goal from *n*
g(*n*) = cost so far to reach *n*

Construct a graph with the topology as illustrated on the next page:



Your implementation should be an <u>undirected</u> version of the above graph, so for example, if there is an arc from ‘w’ to ‘s’, there is also an arc from ‘s’ to ‘w’.
Note also that node ‘1’ is not repeated – the node underneath is the letter ‘l’.

In order to test your algorithm, assume the desired route is from (number) ‘1’ to ‘v’. You need to store an x,y position at each node so you can compute the heuristic (the Euclidean distance) between nodes.
I suggest adding the x,y position as new member variables to class GraphNode. The positions should correlate to the distances shown in the graph above. If we assume the first row of nodes are drawn at y = 0, then, e.g. node ‘4’ might exist at 100,0, while node ‘l’ is at 0,0.

Remember the heuristic should always underestimate slightly.

Test your implementation thoroughly – try different paths through the graph.

Part 1 Marking Scheme (50% total)

Component	Weighting
Initial g(*n*) values set correctly for each node	5
Priority queue sorts nodes where node with lowest evaluation function f(*n*) has highest priority	10
A child (neighbouring) node is only evaluated if its previous pointer is not equal to the node at the top of the priority queue (pq.top) and the child node is still in the priority queue.	5
h(*n*) values for each node are computed correctly.	10
A node's f(*n*) value is only updated if a lower path cost is discovered (previous pointer field is also updated).	5
At end of each iteration, current node is removed from the queue and the node with the lowest f(*n*) value is selected for expansion	5
Algorithm terminates if destination node not found (i.e. priority queue is empty) OR highest priority element is the destination node	5
Input vector holds best path generated by algorithm	5

Part 2

Using the SFML library, draw your graph structure to approximate scale by using circles for nodes. For the above graph, consider the distances as pixel values. This is important as later the A* heuristic will use the Euclidean distance between nodes. You may wish to add additional information to the nodes text file such that each node contains the screen coordinates of where it should be drawn, e.g.:

Node	Screen Coords	
1	100,100	(top left corner of circle's bounding box)
2	200, 100	
..		
8	580, 100	

Inside each circle, show the name of each city, along with the current path cost i.e. g(*n*) (initially, this will be unknown).

Draw edges between nodes to show the relevant connections. Assume the graph is undirected, so a single edge is bi-directional. You may wish to use a similar approach for drawing the edges as for the nodes, i.e. store the screen coordinates in the arcs text file:

From	To	Weight	LineFrom	LineTo	
1	4	100	110, 105	200, 105	(assuming circle diameter = 10)

The weight (path cost) should be illustrated above each edge.

Allow the user select an origin node and a destination node. Use colouring or shading to highlight the selection - it should not be possible to select more than two nodes and the origin and destination must be different.

Once the origin and destination nodes are selected, compute the heuristic h(*n*) value for each node in the graph as the Euclidean distance from the centre of a node's circle to the centre of the destination node's circle. Display the h(*n*) value below the g(*n*) value at each node.

Provide a start button and a reset button. The start button will initiate the search, and show what nodes are (i) *visited* and (ii) *expanded* by A* between the origin and destination (use different colours to show the visited and expanded nodes). As each node is expanded, show the updated g(*n*) values. Once the destination is found, set each node/circle to a specific colour along the optimal search path (if the destination is not reachable, do nothing).

The reset button will reset the graph state so the user can select new origin and destination nodes.

Part 2 Marking Scheme (50% total)

Component	**Weighting**
Nodes visually represented as circles showing city names, g(*n*) values.	5
Edges drawn with weight information shown and Graph drawn to approximate scale	5
User can select origin and destination nodes with visual confirmation	5
h(*n*) values are calculated and shown for each node once the origin and destination nodes are selected.	5
A* runs when start button pressed, nodes that are expanded during the search are coloured and updated g(*n*) values shown.	10
Once destination is reached, nodes along the optimal path are highlighted.	10
Reset button resets graph state to enable selection of new origin/destination nodes.	10

Each component will be graded 0 to 3 where:

- 0 means nothing of value;
- 1 means significant problems with the implementation;
- 2 minor problems with the implementation;
- 3 fully working implementation.

Note carefully: Once your project mark is calculated for part 1 and part 2, it will be weighted by your attendance and participation (i.e. github commits). There is a 50% weighting attached to attendance and participation. For example, if you miss all the working time allocated to this project, you will be awarded 50% of your overall project mark.