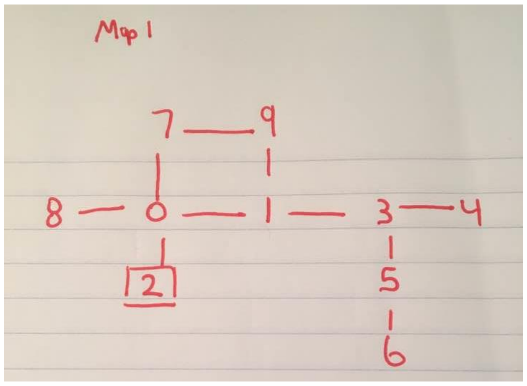
Joshua Chung

ECE 4574 – Assignment 1

Overview:

Map implementation – I use a graph model of an adjacency matrix so that I can keep the weights of all the connecting nodes and whether there may be an obstacle. The obstacle can be changed simply by changing the obstacle type. The map does not heavily depend on the obstacles type so changing it will not affect the overall map. Also, the map isn't "type" specific (i.e. grid, space, free-space). The map is only dependent on the existing nodes. Therefore, however the map is set up all that needs to be ensured is the nodes for where the rover can travel to.

TEST DOCUMENTATION



GRAPH test (i.e. GRAPH_*)

1. **Check_initialize()**
   For this I make sure I initialize my edges and nodes within the constructor. Each edge and node

Weight (used for obstacle detection)	Default: 5
Visited (Boolean for pathfinding)	Default: false
Heuristic Value (they are all set to max nodes)	Default: N*N where N = # of nodes
Movement Cost (the total cost to move node)	Default: 1000

Test: PASS CASE
Explanation: A pass case ensures that I can create my map with default values and then modify them as needed. Also, setting the default movement cost to 1000 ensures a pass case for calculating new lowest movement cost. I run QVERIFY to ensure all of my node default values equal to the default values. I know this is correct since I use a return_method to get the current nodes values. Based on the returning value I know whether or not the returned value equals.

2. **Check_weight()**
   To allow modifiability of the obstacles I included an *"add_weight"* function which uses and input to assign a weight to any specified node-edge. For this test I test whether or not I can correctly set my weight values. This would be used for when there is a new obstacle that needs to get registered into a node-edge.

   Test: PASS CASE
   Explanation: A pass case would be that I can input any weight and the map will regenerate itself with the new weights and have an effect on the new path to take. To test this I first test a simple node-edge by first setting the weight to 2 and then check whether or not I correctly assigned the node-edge the correct weight by using a return_method of a specific node-edge. I know this is correct since the default weight value is 5 and since I changed it to 2 the Test passed. Afterwards, I chose a random weight value and then proceeded to add weight values to a random amount of node-edges. Then, I check to see if I correctly set the weight by simply using the return_method at the generated node-edges.
   Extra: Since I implemented a graph using an adjacency matrix I also had to check that there were the same weights going both ways from a node since my "map" was an undirected map (i.e. The weight from A-B should be the same as B-A).

5	5	5
5	5	5
5	5	5

Table 1: Original Initialization Matrix of weights

Setting node weight B-C to 10 will result in : therefore B-C == C-B

5	5	5
5	5	10
5	10	5

Table 2: Side goes A-C (top to bottom), Top goes A-C (left to right)

3. **Check_map_resize()**
   To allow flexibility for my map I created a resize() method such that if the map had to expand an extra node all I would have to do is pass in the amount of new nodes such that the map can generate another node for my matrix. For the test I ensure that when I add a new node count the nodes and edges both equal the new node count – since I am using an adjacency matrix.

   Test Result: PASS CASE
   Explanation: Since I am running an adjacency matrix every node has the possibility of n edges (where n = #nodes and the node itself counts as an edge). Therefore, for a map with 2 nodes I would need a 2x2 matrix to satisfy the nodes-edges. The resize() method does so by simply creating a new matrix with the new given size.

5	5
5	5

Table 3: initial Map without new node count

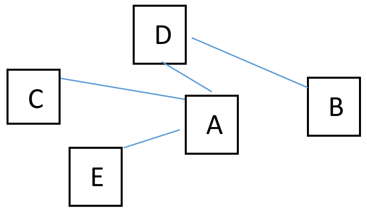
Setting a new node count of 3 will result in 3 nodes and 3 edges now and the resulting adjacency matrix will equal.

5	5	5
5	5	5
5	5	5

Table 4: A new column for the edges and row for the node

Test Result: FAIL CASE
Explanation: This test will fail if the memory space does run out of memory to hold the matrix and all of its internal data. This, can also have the possibility of failing the other cases for checking weight since it does not keep the original matrix's values – however, this is an easy fix by just resizing the matrix vector.

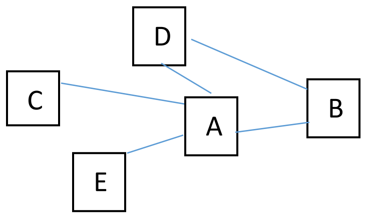
4. **Check_neighbors()**
The purpose for this test is to check that I can simply just get the neighbors of any node without having to change or reach into the actual class. The return_neighbor() method was developed for the purpose of the pathfinding. I wanted to make sure that the pathfinding didn't have to reach into the actual "map" but can simply just get the returned neighbors and their corresponding data to calculate the movement cost for next nodes. For the test I put in any random existing node and use a return() method to extract the neighbor for that node.



For this graph (left), if I were to use the return_neighbor() method for A it would return a vector {C,D,E}. However, if E happened to be an obstacle then the vector returned would become {C,D}. I make sure I omit all obstacles from any calculation vectors.

Test Result: PASS CASE
Explanation: I tested this by creating an existing vector of known neighbors and comparing them to the actual neighbor vector that would be returned from the return_method().

5. **Check_lowest_node()**
The purpose for this test was to test the functionality of my pathfinding() after the map had been regenerated with new movement costs. The function will take in a parameter and then return the single lowest movement cost node to move to based on current node. I decided to design it this way because I assumed that the map will be constantly updated with new obstacles or nodes. Therefore, if the map does change then I will recalculate and simply go to the next best node. To test this I created a start and end node for a path. Then I would check at each node what the best path would be to move to.



For this graph (left), if the start was C and the destination was B. Then, if I were to input C as the current node then I would get returned A since that is the only possible node to move to. However, if I input A then I would return B since it is my destination and also since it would have the lowest movement cost rather than A-D-B.

Test Result: PASS CASE
Explanation: To test this functionality I simply just input different node locations after setting a start and destination and then checked if my returned node was the closest node to the destination. A pass case would also be that I do not return an obstacle as a neighboring node.

<u>STAR test (i.e. STAR_*)</u>

1. **Check_openList()**
   The purpose for this test was to ensure that I populate the open-list for my A* correctly for all neighboring nodes. The function simply returns the vector consisting of all neighboring nodes for the node. When the node advances to another node, the existing node in the open-list disappears and then the open-list gets repopulated with the new neighboring nodes.

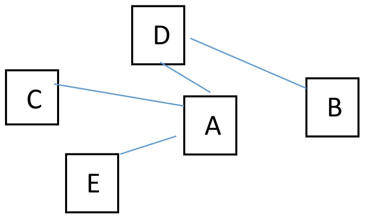
   Test Result: PASS CASE
   Explanation: A pass case would ensure that I can correctly search through my Matrix in order to find all connecting edges to the current node without including an obstacle node. To test, first I set up a start node and use this start node and first get the vector of current open-list nodes. Then, I would check to see if the open-list did indeed contain the neighboring nodes. Then, I would move node locations and repeat the test but this time to see if the current node was removed from the open-list and the new neighboring nodes were included.

2. **Check_no_obstacle()**
   The purpose of this test is the same as the first test (above) but the only difference is that this test was specific for obstacles. I wanted to make sure that the open-list never contained an obstacle.

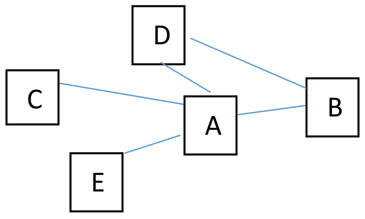
   Test Result: PASS CASE
   Explanation: Whatever the obstacle may be, if the obstacle exists it will never be a part of any vector list. The movement cost of the obstacle will also be at the max number (1000) such that it will never be included within a list.

3. **Check_Path() + Check_new_obstacle_path()**
   This is probably my major test. The purpose of this test is to ensure that I can correctly navigate to a node for the correct path from any node. The new_obstacle_path() test's purpose is to essentially

test the same thing, but this time if the map changed with an inclusion of an obstacle the path would get updated correspondingly.



For this graph (left), if the start was C and the destination was B. The optimal path returned would be C-A-D-B. For the right graph, the optimal path would be C-A-B.



Test Result: PASS CASE
Explanation: Essentially, I am testing the whole A* class as a whole. For this test to pass I had to ensure that my algorithm was indeed correct where my openlist and closedlist contain correct nodes and do not contain any obstacles. Also, for this to be correct the correct low-cost algorithm for node movement had to be made such that I would allow to move in any direction. Furthermore, the test should pass the case that I can move from any one location to any destination at any given point. To test this I create a map where I know the calculated fastest path. Then, I compare the generated path with my known path to see if they equal whenever I change the path.
Test Result: FAIL CASE
Explanation: There is a chance that continually running this may end up with incorrect movement cost values since I do not essentially “reset” these movement cost values if an obstacle gets included – I would just recalculate.