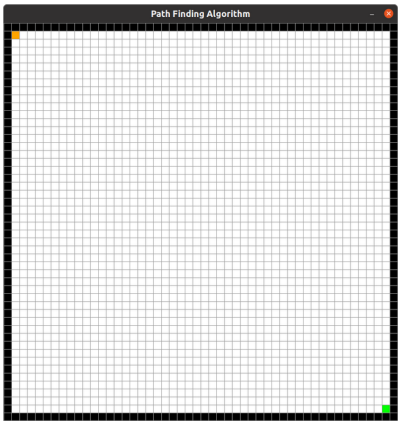
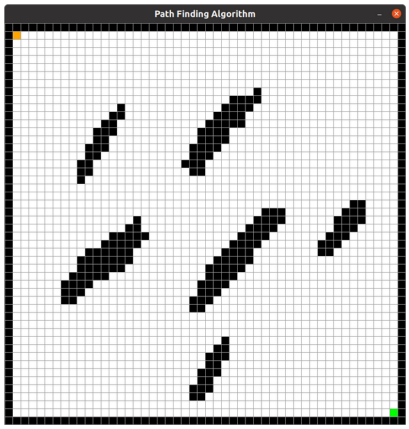
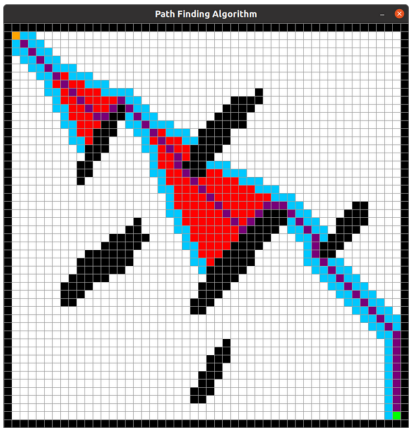


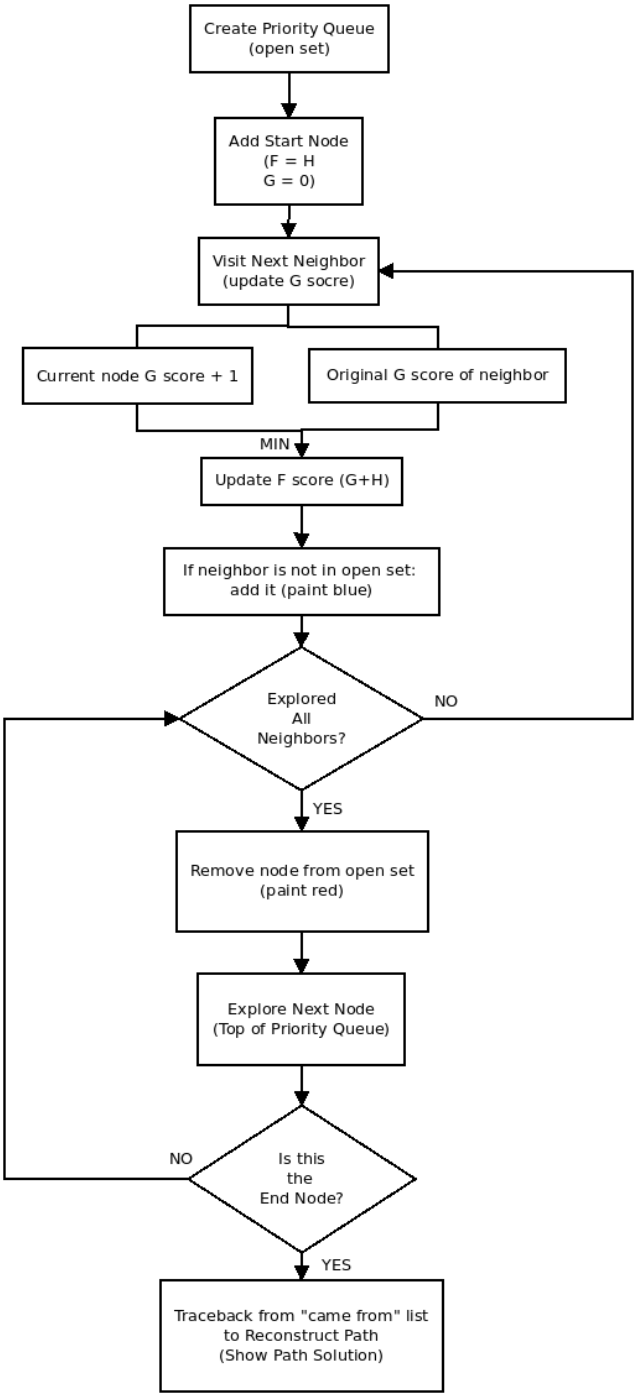





**Sample of Workspace and solution:**

- Figure 1 (on the left) shows the pygame window that pops up when the python script is run. The nodes are represented here as each white square of size gapXgap where gap is the width of columns (or rows) that would fit within the width (or height) of the window. The width and height of this window are assumed to be equal and are defined at the start of the script. The size of the window does not directly translate to the "size" of the grid, the number of nodes define how many boxes are in each grid. Again, the number of nodes in each row is equal to the number of nodes in each column so the grid is square (just as the window is square). Notice that the figure does not actually have a black border, these are just nodes at the edge of the workspace that are assumed to be barriers from the start. Also note the orange start node in the top left corner and the green end node in the bottom right corner.
- Figure 2 (in the middle) shows the same window after it has been updated with random "obstacles". These obstacles assume the same barrier "state" (or black color) as the border nodes and are added to the window by user input. Like many other Python visualization tools, the pygame library includes methods for tracking mouse location and mouse/keyboard events (clicks, keypress, etc) which allows us to directly "paint" obstacles onto the window. And since this visualization tool updates at a specified refresh rate, we can update the map with obstacles in real time. Note that a failsafe is included to prevent user input while the search is running. Furthermore, a clear method was added to allow the user to clear the map from either the obstacles or the obstacles and the completed map/solution.
- Figure 3 (to the right) shows the final result after the search algorithm is called and the while loop has terminated either because no more nodes remain in the open set (no solution) or because an end node is reached

**Flow Diagram of Search Algorithm to Implement:**

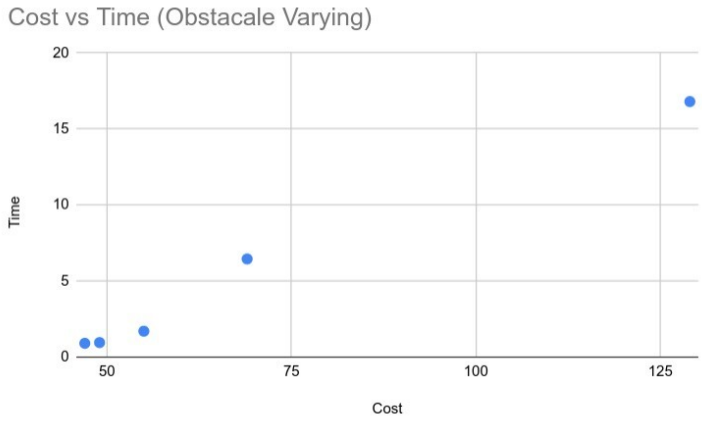


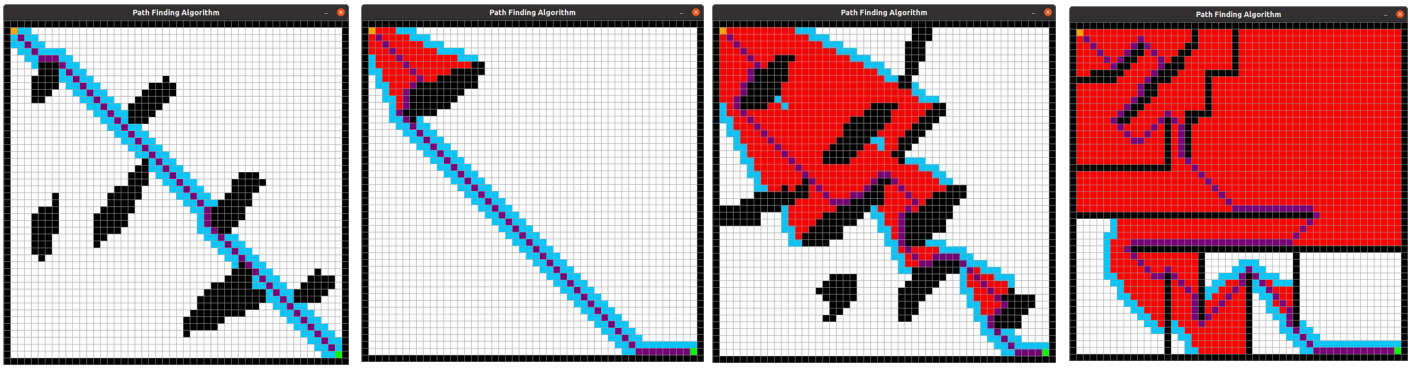

Flow Chart Notes:

- This only defines the search algorithm and not the grid/graph formation. In other words, a separate method is called at the start of the script to update all the neighbors of every available node before the search algorithm is called
- The use of a PriorityQueue allows us to both:
  - store all the nodes that we visited but are yet to explore in ascending order (least first)
  - pull the node that is on the top of the list and remove it from the queue at the same time.

**Graph of Solution Cost vs. Time (Obstacle Configuration Varying):**

Here, the clear method (c key) and the mouse_click method make it very easy to reiterate the search with different obstacle configurations without having to close the pygame window or the main program. Of course, for the different grid sizes (number of nodes) we had to close the window and rerun the program with an updated value for the total number of nodes (rows variable). The points from left to right on the chart correspond to no obstacle, then the first figure on the left and so on:



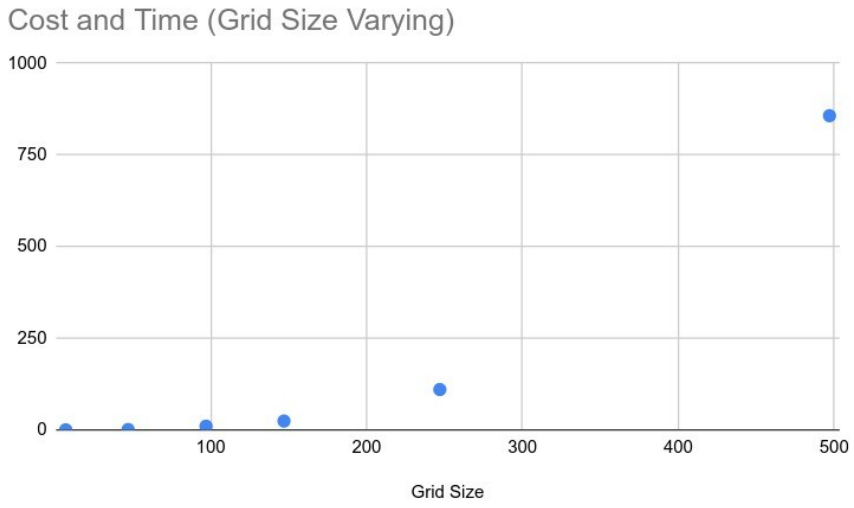




The four figures under the cost vs time chart (from left to right) are Obs1 thru Obs4 as shown on the chart. Note how it does not really matter “how many” barriers are there in the workspace, what matters is how many of those barriers are in the way (diagonal) between the start and the end nodes. Note that the time complexity of the algorithm grows as the cost of the path grows which is expected given that longer paths will take longer to find since this algorithm prioritizes the nodes with the lowest F score and the F score has a heuristic component which does not account for barriers. The more we “block” the shortest possible paths (growing from the diagonal) with barriers, the more the algorithm has to search before it arrives at the final node.

Also note that the grid size was kept to a consistent 50x50 for all iterations above. In other words, we are holding the grid size constant as we vary the obstacle configuration to isolate the effect.

**Graph of Solution Cost vs. Time (Grid Size Varying):**

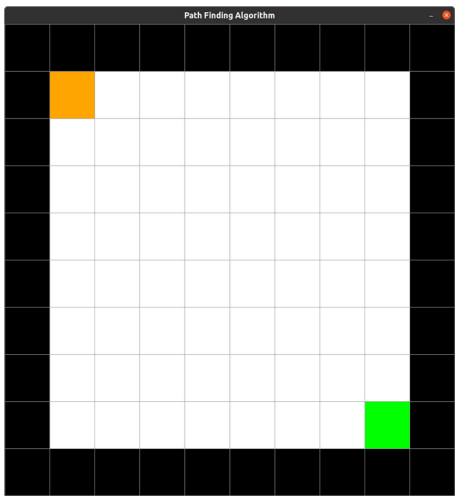
Here we are only varying the grid size with no obstacles to isolate the effect of the number of nodes on the time complexity of the program:

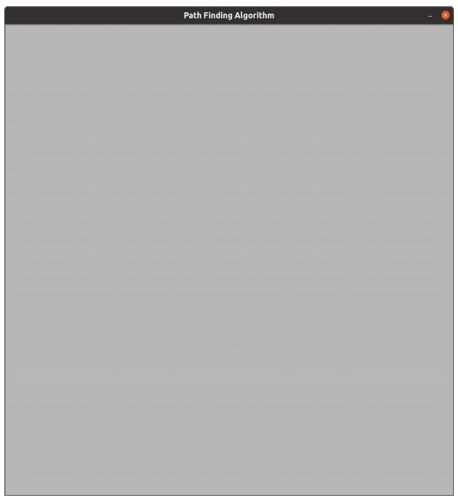



Shown above points from left to right correspond to grid sizes of 10, 50, 100, 150, 250, and 500 respectively

Note that while it is expected for the time complexity of the algorithm to grow as the grid size grows (as there are more nodes to explore and a longer path to get to the end node), the exponential increase in time for higher grid sizes is mainly due to the fact that my python script constantly repaints the map as the current state. This constant refresh rate is useful to give the effect of a simulated search but is extremely computationally expensive. This is why I limited the chart above to grid size of 500, my laptop was crashing for anything above 500x500 grid size.

Below shows the contrast between a 10x10 grid (left) and a 500x500 grid (right):



**Instructions for Running the Script:**

- pip install pygame first through your python terminal
- open RBE550HW1.py and run the code (you may scroll down to line 272 for editing the number of rows (nodes/grid size)
- Use your mouse to draw obstacles on the map
- When you are ready to run the search algorithm, hit the space bar
- When you are done watching the simulation and reviewing the result, press c to clear the workspace and start drawing new obstacles again