Assignment 3

For the first task of this assignment, we were required to create two graphs. For the smaller first graph, I found a graph online which consisted of 20 nodes. This graph has enough connecting edges to represent something like a set of shops in a city market. The graph however did not have weights and was not directed so, I decided to randomly assign the weights and made it directed such that a lower ordered node points to the higher ordered node. The weights can represent the distance between the hypothetical shops. This graph was chosen because it can represent many real world situations as the nodes can represent almost anything. Most of the nodes, especially the ones at the center, have many connections adding a bit to the complexity yet it is easy to observe the shortest path. This graph should serve as a great test case. The shortest path for this graph is- 1 -> 2-> 7-> 8-> 14-> 20 and this is the expected answer when running each of the search algorithms. For the second graph, I decided to create a graph such that it is in the form of a grid. This decision was made so as to facilitate the heuristics calculation because it is simpler to visualize the positions when each node lies inside the grid.. I have included the script that I wrote to create a graph with 10,000 nodes. Each of the nodes is connected to 3 other nodes, the one next to it, the one diagonally adjacent to it and the one directly below it. The weights for each of the eachs is decided randomly as a value between 1 and 10. For each of the graphs we try to find a path from the 1st node labeled 1 to the last node labeled 20 or 10,000.
Next task was to implement Dijkstra's algorithm and A* search algorithms. As noted during the class, Dijkstra's is essentially A* but the heuristics is always 0. For both our graphs the A* search algorithm performs better. This is expected because of the heuristic which works to our advantage. For the graph with 20 nodes, Dijkstra's algorithm took about 0.366561ms on an average and the open list was empty at the end. The closed list consisted of all 20 nodes suggesting that the algorithm had to visit all the nodes at least once through the shortest path to reach them. In contrast the A* the search algorithm took about 0.344114ms on an average and the open list comprised 2 nodes while the remaining 18 nodes formed the closed list. This suggests that only 18 nodes were expanded including the goal node which gave us the advantage of 2 nodes not being visited and thus suggesting a performance improvement. A* search algorithm with a greater overhead of heuristic calculation and storage should take up more memory. For both these algorithms I observed that we should only replace the way to get to a node in the open list only when the csf/etc values are strictly less for a new path. Not doing so can cause the search to abandon the lowest cost path. For the larger graph, Dijkstra's algorithm took about 4466.25ms on average and the open list consisted of 58 nodes while the closed list consisted of 9174 nodes. Not all 10,000 nodes were visited but 9174 nodes were expanded to reach the goal. In contrast, A* search algorithm took about 3471.26ms on average and the open list consists of 154

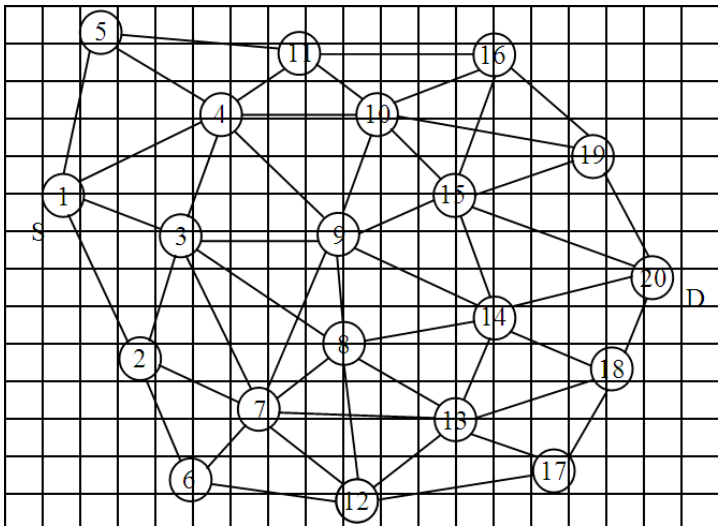
nodes while the closed list consists of 7323 nodes. This is interesting because as the graph becomes larger we have reduced the number of nodes we had to expand but we also have a larger set of candidate nodes in our open list. The fill is large causing more memory to be consumed but we are able to reach the goal much faster. Same as the smaller graph, more memory should be consumed. More nodes in the open list may be due to the breadth-first nature of the search algorithm.

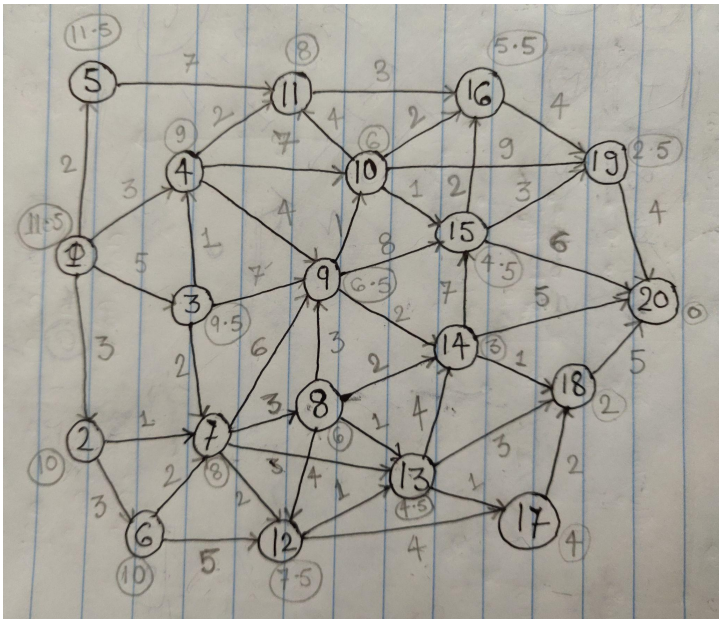
Next we need to discuss the two different heuristics that I have used for the smaller graph. The first is Euclidean Distance metrics. It has proven to be admissible and consistent. This is true for most 2D environments and is also true for the graph. The consistency when checked manually shows that it is consistent at all the nodes. For calculation of the heuristics for the graph, I plotted a grid on top of the graph for visual aid. The second heuristic used was the Manhattan distance. This proved to be inadmissible as it over-estimated the distance several times and is mostly suitable for movement in cardinal directions. When running the algorithm using manhattan distance, the program completed in about 0.285631ms, which is less than when run using euclidean distance. This also yields us a path which is not optimal. The open list contains about 8 elements and the closed list 7 at time of completion. The overestimation results in fewer nodes to be expanded and some of the nodes that form the lowest cost path are not expanded as in the case of node 14. Since the algorithm terminates once we expand the goal, we are not able to explore other options before reaching the goal. If we observe the open and closed lists, we can see that the nodes 14 and 9 are the nodes which were hit by the algorithm and were overestimated by about 2 and 5. If more nodes were hit, more overestimations would be possible.

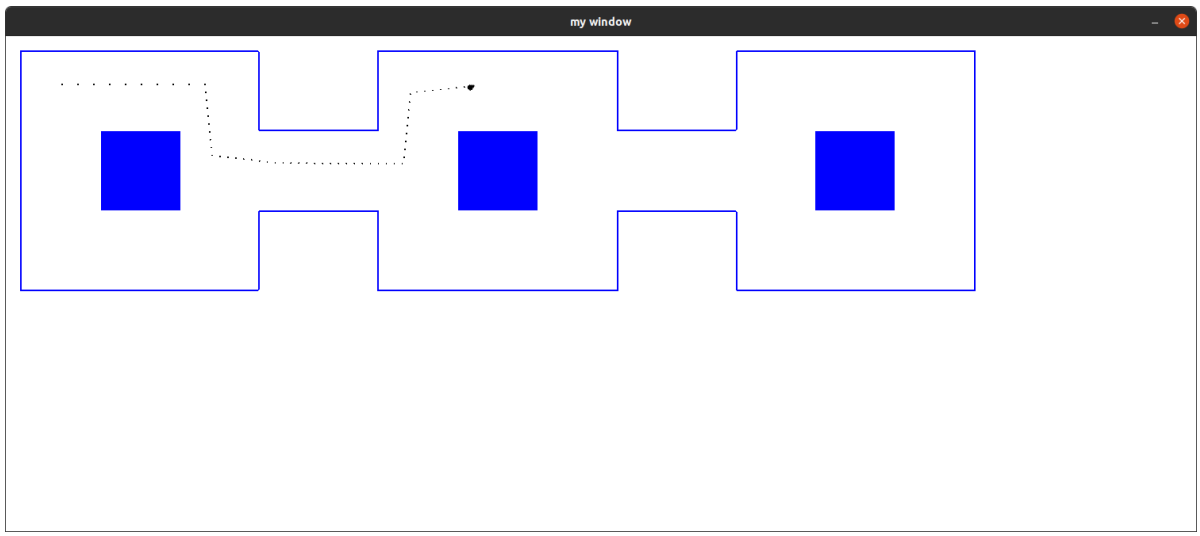
For the next task we needed to combine the pathfinding with path following. The environment that I decided to test this comprises three rooms placed side by side, joined by a narrower corridor. In each of the three rooms an obstacle is placed in the middle to avoid direct movement from corridor to corridor and a need to navigate around the obstacles. In order to apply A* search and form a graph from the created environments, I chose the points of visibility approach. I divided each room into 9 smaller parts around the obstacle and chose the point of visibility as the center of mass for the room. These points will be connected in the graph such that movement will be possible in both directions. The corridors are divided into two equal parts. These points are then added to the graph for navigation through A* search. Whenever we observe the mouse click, we find out the closest nodes to the click and the current sprite position as the starting point and ending point respectively. A new route is calculated for the mouse click and the exact position is then added at the end of the path as the movement from the closest node to the actual target will then be a straight line. The problems not yet rectified include the sometimes incorrect rotation of the sprite in the direction of motion and if we click rapidly between locations the sprite seems to jitter.

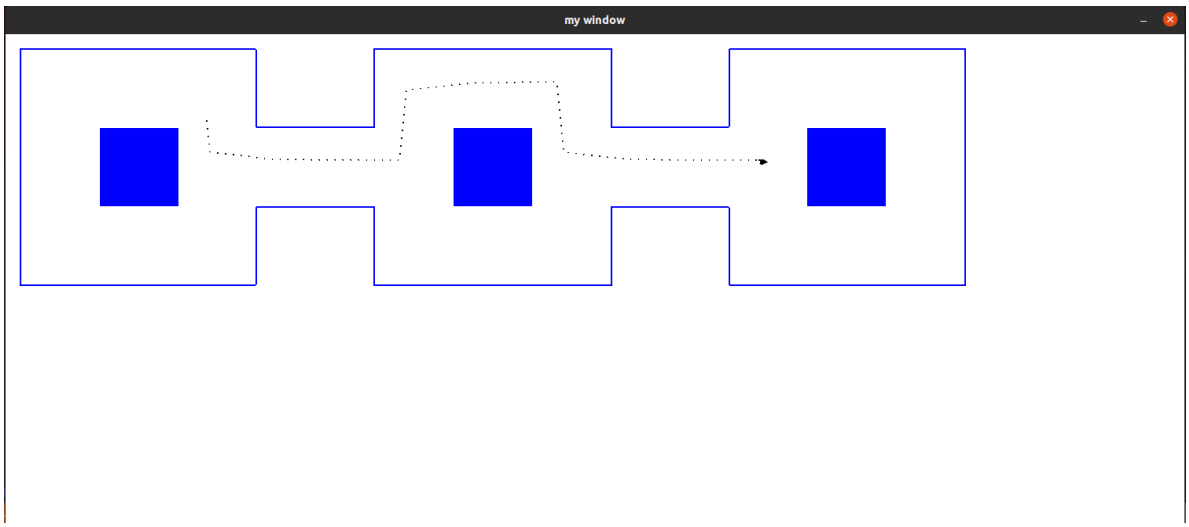
References:
https://www.researchgate.net/figure/A-simple-undirected-graph-with-20-nodes-and-48-edges_fig1_289036860

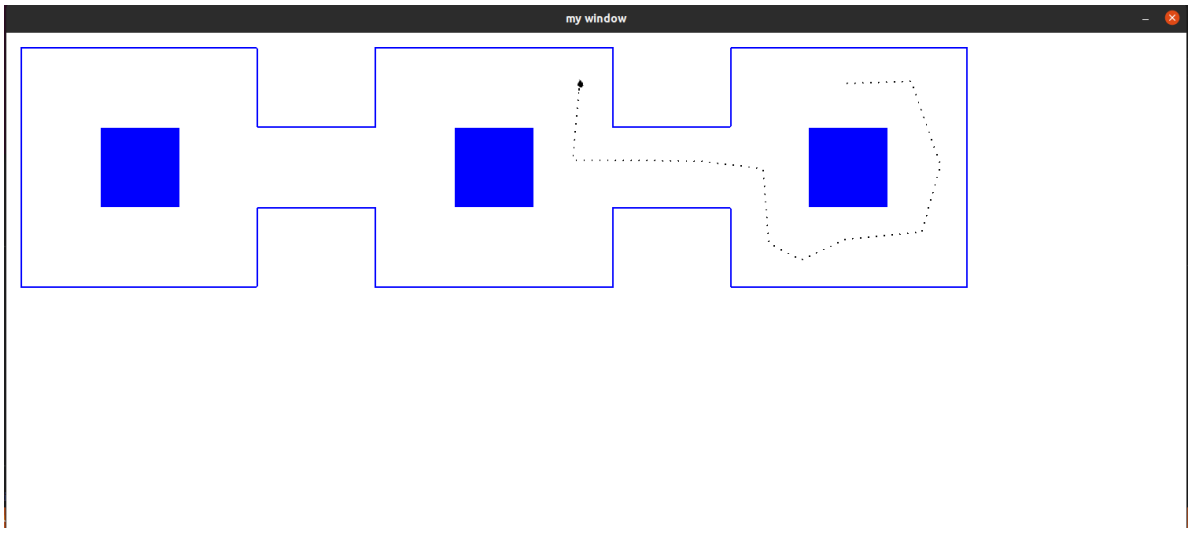
Screenshots and Relevant diagrams:






my window


my window


my window