Homework 2: Pathfinding and Path Following

Kshitij Patil

Department of Computer Science

North Carolina State University

kspatil2@ncsu.edu

Path Following and Path Finding are one of the core areas in any Game AI. In the previous assignment, we experimented with Movement and various Kinematic and Steering behaviors that form the base for Path Following and eventually Path Finding. In this assignment, we will develop and understand mainly the Path finding algorithms, their applicability on various input graphs and eventually use these algorithms to find the optimal path and display it using Path Following based on steering behaviors implemented in last homework.

FIRST STEPS

The basic procedure for any AI character to display movement follows a certain fixed chain of operations. In order for the character to go from one place to another, it needs to understand its environment. This introduces us to world representation. The character understand the world with the help of a graph. This forms the input for our Path Finding algorithms.

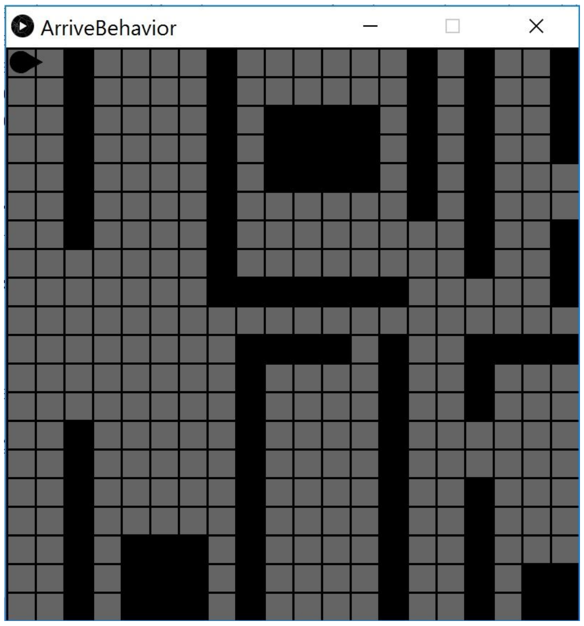


FIG 1: Small Graph

The first step of the homework is about constructing a small and large graph. A graph constitutes of nodes, edges and weights associated with each edge. The instructions does say that the number of nodes for small graph should lie around 20-30, but I have chosen a small graph with about 400 nodes. The main purpose of choosing the number of nodes in small graph is to have a comparative study later with the large graph. The number of nodes in large graph is 10,000 ,so the comparison between these graphs will give a good result even if we choose 400 nodes rather than 50 nodes in small graph. Also, I am using the same graph for the room algorithm so reducing the work for the rest of the subparts.

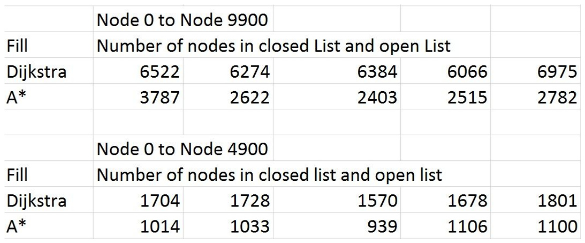
Figure 1 shows the small graph that I have used for testing the Path finding algorithms in further sections. I have designed this graph based on my apartment's top view projection. The black tiles represent the walls and tables around the house. The graph is a basic skeleton of the apartment removing small objects and unnecessary details. The gray tiles are nodes that can be used for movement around the apartment. Each gray tile forms a node and it is connected to maximum of 4 nodes around it if they are gray. The four nodes being top, bottom, left and right.

The large graph consists of 10,000 nodes and randomly selected edges between each vertex. The Large graph is also a tiled graph. As there are no obstacles in this graph, I have used a probabilistic approach to decide whether there is a node between any adjacent nodes and by adjacent I mean only top, bottom , left or right. For each node I create 4 random numbers between 0 and 1 and if the number is greater than 0.2 then there is an edge between the corresponding two nodes. Choosing a probability value of 0.2 gives a very dense graph. Although, there is a possibility that there might not be a path from between some random pair of vertices. That is why, I decided to choose a dense graph to hopefully avoid such a situation.

DIJKSHTRA's ALGORITHM AND A*

This part of the homework actually tests two important Path Following algorithms namely : Dijkshtra's algorithm and A* algorithm to calculate the shortest path between any two nodes.

Firstly, I ran both the algorithms on small graph which I would be doing ahead before my Path Following code too. Not much substantial difference is seen when the algorithms are run on the small graph. A good comparative study can be done between the algorithm for large graph. Both the algorithms are tested on three criteria: Fill, Memory Use and Run time.

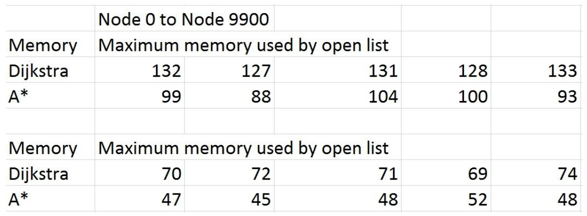


	Node 0 to Node 9900				
Fill	Number of nodes in closed List and open List				
Dijkstra	6522	6274	6384	6066	6975
A*	3787	2622	2403	2515	2782
	Node 0 to Node 4900				
Fill	Number of nodes in closed list and open list				
Dijkstra	1704	1728	1570	1678	1801
A*	1014	1033	939	1106	1100

FIG 2 : Fill Of Dijkshtra and A* for two different path searches.

The Fill of a Path Following Algorithm is the number of nodes visited in searching the shortest path from one node to another node. The procedure taught in the class for calculating the shortest path for both the algorithms uses the open list and closed list of nodes. The sum of the sizes of both the lists gives the fill of the algorithm for that search. Figure 2 shows the fill of algorithms when run on the large graph generated as described in first steps. Both algorithms are run for two searches : from Node 0 to Node 4900 and from Node 0 to Node 9900. The indices 0, 4900 and 9900 do matter since the large graph is a square graph of size 100x100. So, 4900 is like halfway through the entire graph and 9900 is from one end to another and since, it is a tiled graph there are no intermediate paths other than top, bottom , left and right for any node.

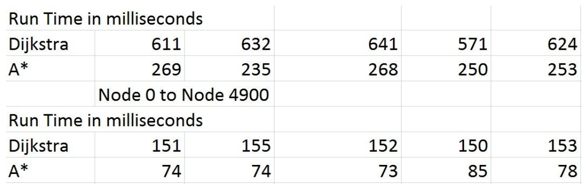
The observations seen in Figure 2 can be explained by understanding the behavior of both the algorithms. Dijkshtra's algorithm behaves like a Breadth First Search algorithm while A* behaves like Depth First Search algorithm. Hence, Dijkshtra's algorithm spreads out more in each direction. A* on the other hand like depth search moves directly towards the target. Because of this approach, not many vertices are searched for while finding the shortest path. In Fig 2, we can see that Dijkshtra's Fill is greater than A* for every search.



	Node 0 to Node 9900				
Memory	Maximum memory used by open list				
Dijkstra	132	127	131	128	133
A*	99	88	104	100	93
Memory	Maximum memory used by open list				
Dijkstra	70	72	71	69	74
A*	47	45	48	52	48

FIG 3: Memory Of Dijkshtra and A* for two different path searches.

The Second criteria on which we compare the algorithms is Memory used. Memory used is the maximum nodes that are searched and stored in the open list. Dijkshtra's algorithm chooses the node with the lowest total cost from the start node irrespective of the target to reach. Hence, at any given point of time, all the vertices connected are into consideration and are in the open list. Being similar to BFS, the number of nodes in open list is higher than as compared to a DFS-like algorithm such as A*.



Run Time in milliseconds					
Dijkstra	611	632	641	571	624
A*	269	235	268	250	253
	Node 0 to Node 4900				
Run Time in milliseconds					
Dijkstra	151	155	152	150	153
A*	74	74	73	85	78

Fig 4:Run time Of Dijkshtra and A* for two different path searches.

The third criteria that we use to compare the algorithms is run time of the algorithm. From Fig 4, we see that A* works twice as fast when searching through half the graph and three times across the entire graph. This can be explained by the use of heuristic in A* algorithms. Heuristic forms a parameter that help estimate the shortest possible path by helping A* decide which path might lead to the shortest path. The choice of heuristic, therefore, has a major impact on how fast the algorithm works. The above implementation of A* uses the manhattan distance as a heuristic which is a very good estimate when using on a tile graph.

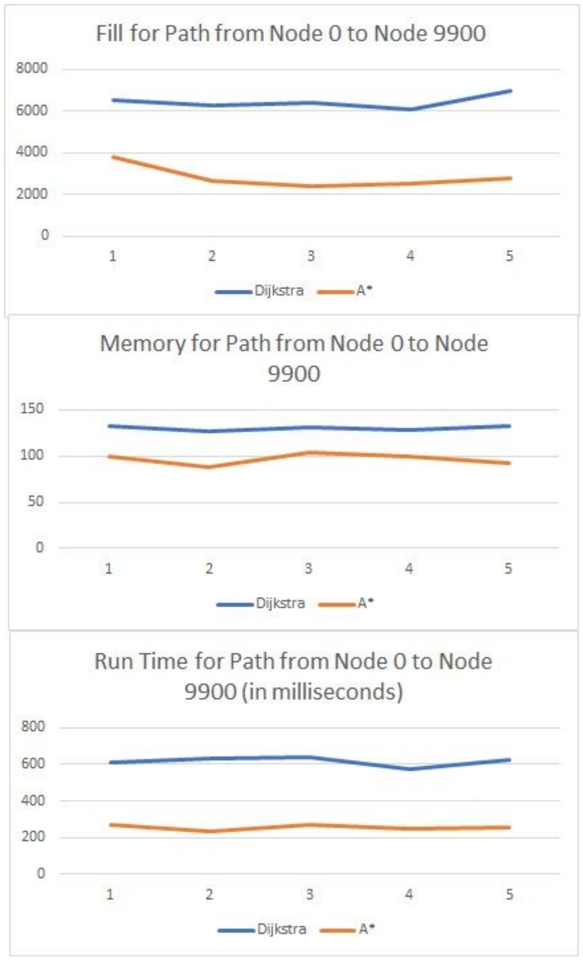


FIG 5: Comparative Graphs for Fill, Memory and Run time for Dijkshtra's and A* algorithms.

Figure 5 shows the graphs for all the three criterias for Dijkshtra's algorithm and A* algorithm. We can see for a large graph or any graph with a large number of nodes A* algorithm is faster, uses lesser memory and visits lesser number of vertices as compared to Dijkshtra's algorithm. That is why, today A* is more popular than Dijkshtra's shortest path approach because of its scalability advantages.

HEURISTICS

There are a number of ways in which we can choose a heuristic while implementing an A* algorithm. In the first part, I used manhattan distance as the heuristic. Manhattan distance, also called as block distance, is the sum of number of vertical and horizontal nodes between source and the target counted on coordinate system on each axis.

Manhattan distance is an admissible heuristic. It is an underestimating heuristic in this graph as the weights associated with each node in my graph is 10 which is greater than 1. Manhattan distance is consistent as it obeys the triangle inequality. If we think about it, manhattan distance gives the actual number of nodes that the character will have to traverse in case of tile graph in the absence of any obstacles. But since our graph does have obstacles decided by the probabilistic edge connections, Manhattan distance is underestimating heuristic.

The next heuristic I have used is the Euclidean distance. Euclidean distance is the straight line distance between any two nodes. Euclidean distance is a consistent heuristic as it does follow the triangle inequality. It is with this distance that we were actually introduced to triangle inequality in the first place. Euclidean distance is an underestimating heuristic in our case mainly because the character only transverses on the tile graph in left, right, top, bottom direction. Hence, it can never take a straight path between any two nodes not adjacent. Thus, the euclidean distance will always give an underestimating value even in the absence of obstacles. Obstacle density later decides the degree of underestimate Euclidean distance provide. Euclidean distance is an admissible heuristic as it is underestimating heuristic.

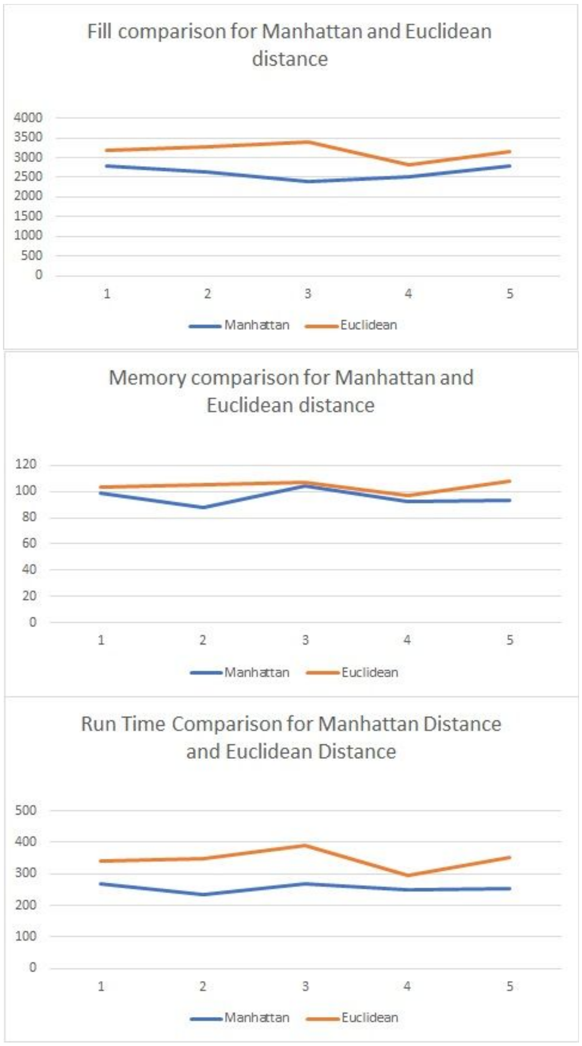
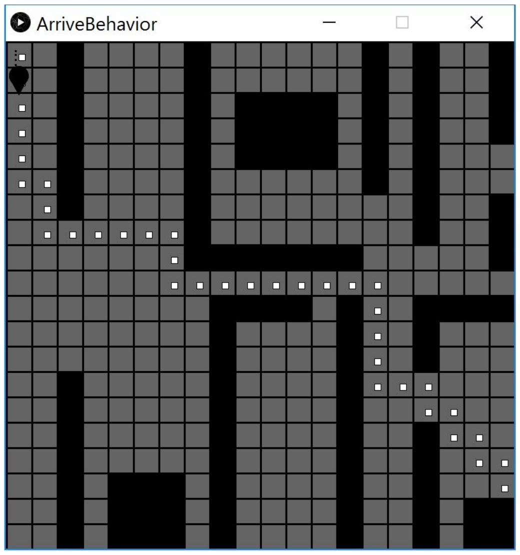


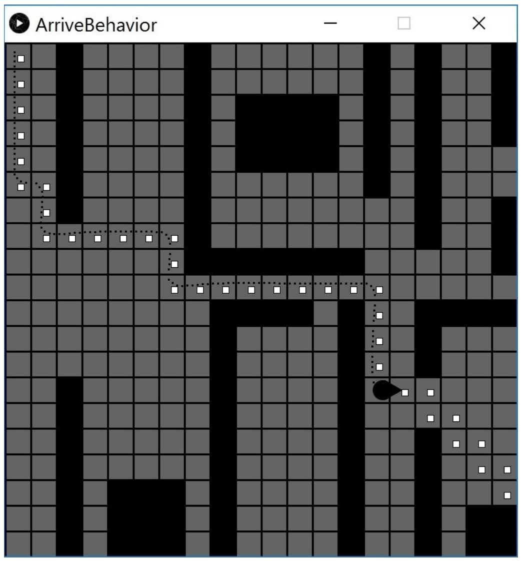
FIG 6: Fill, Memory and Runtime comparison for Manhattan and Euclidean Distance.

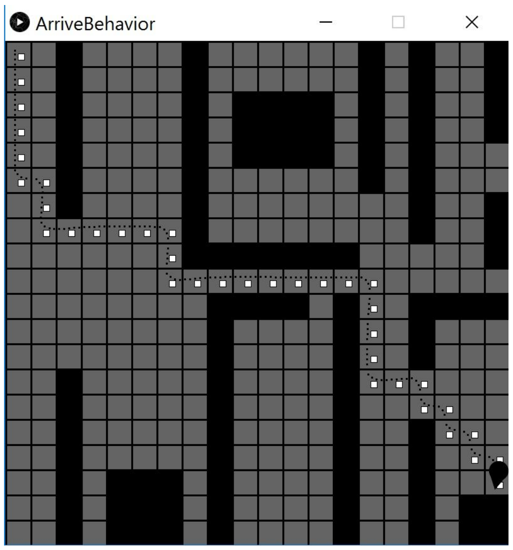
In Figure 6, we can see that there is not much difference in euclidean heuristic and Manhattan distance but in general manhattan distance works better in terms of fill, memory and run time than the euclidean distance. I think this is because of our choice of using tile graphs for the large graph. Manhattan distance which similar to the tile graphs is a much better heuristic than euclidean. Euclidean distance is an underestimating heuristic and the presence of obstacles makes it more underestimating. Hence, I feel in our case, the use of manhattan distance with tile graph gives a better result because of its structural similarities.

PUTTING IT ALL TOGETHER

This part of the assignment is the implementation of Path following on the output generated by Path Finding algorithm.Following are the screenshots of the character following the shortest path on the graph.







The character will not move if we click on a wall, as those nodes are not connected to the nodes that are walkable.

I had to modify the radius of distance, radius of satisfaction considering the size of each node. The character size was also reduced to fit it inside the node. From the breadcrumbs, you can see that the character moves very smoothly on turns. This is because I have given the character a low maximum velocity and high acceleration so as to make a swift turn and avoid drifting due to residual velocity in previous direction of motion.

CONCLUSION

Implemented two Path Finding algorithms, using various metrics in it and performed a comparative study between Dijkshtra's and A* algorithm. The completion of this homework helped me understand the difference in these two algorithms when run on a large graph which is how it is I think at an industrial level. I also performed a comparative study of the manhattan distance and euclidean distance as heuristic for A* algorithm. Finally , implemented Path Following algorithm for the results of path finding algorithm.