Pathfinding Algorithms

Mit Doshi
University of Utah
Mit.doshi557@gmail.com

Abstract

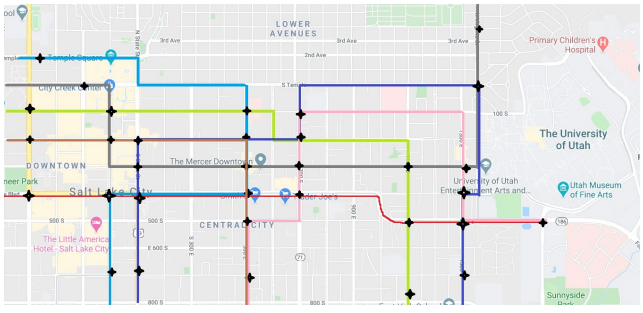
This document is a report of the implementation of various pathfinding algorithms using graphs. We have generated random graphs and then am using A* and Dijkstra algorithms to find the most optimal path with the least amount of time. Then, using the Tile Base division scheme to quantize and localize the graph for finding a way that is then used by our boid to arrive dynamically.

Graphs

Small-Scale Graph

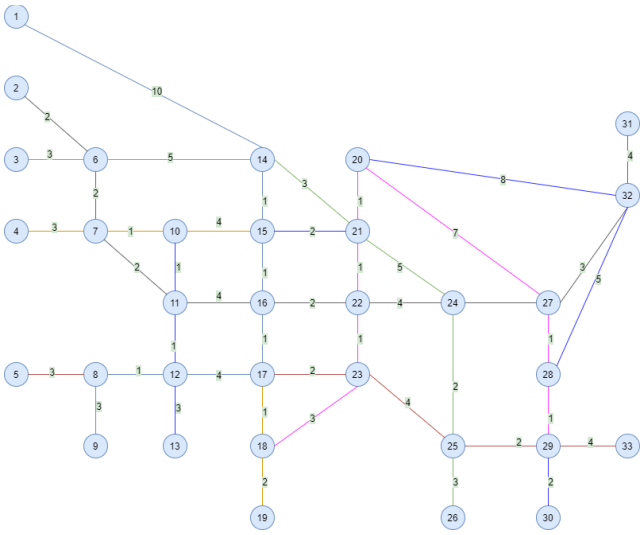
We have different transit routes going across the downtown, and they often intersect with each other, and people usually transfer buses to reach their destinations. So, Downtown Salt Lake City Map is a perfect world space for a small-scale graph. To simplify the algorithm and graph, the nodes of the graph represent any one of the three possibilities:

- Start of the route.
- End of the route.
- The intersection of two routes.



SLC map to create the graph

The black markers in the above figures are the intersection between routes or the start and end of routes. These markers are the nodes in the graph; the cost to travel is the number of blocks from one tag to another tag; the connection between markers would be the edge that connects the two nodes. The graphical representation of the map is shown below with the cost and connecting edges between each node.



Node Representation of Salt Lake City Map

The "*SmallGraph*" class handles this node representation. It takes two inputs to create the graph out of this node representation:

- Number of Nodes inside the graph
- Adjacency Matrix for the Node representation.

There are 33 nodes in the graph and 46 different edges. The exciting thing about this graph is that there are circular nodes that provide multiple paths to a destination from the source. For example, let us take six as the source, and 21 as the destination, following are three paths of all the paths possible:

- 6 – 14 – 21, the cost is 8.
- 6 – 14 – 15 – 21, the cost is 8.
- 6 – 7 – 10 – 15 – 21, the cost is 9.

So, there are multiple routes to take; one is optimal in both nodes to travel and cost, whereas one is optimal in cost but has more nodes to travel before it reaches its destination. So, this graph helps to check the efficiency of the algorithms in multiple ways.

Large Graph

To stress-test our algorithms, we had to perform the algorithms on a graph with more than 1000 nodes. The system allows a user to input the number of nodes for the graph in the range of 1000 to 10000 nodes. The *"LargeGraphCreater"* class creates a random graph with the number of nodes specified with the following consideration:

- Each node has one outgoing edge.
- Each node has one incoming edge.
- Each edge has a random cost attached to it.

The graph created using this has a random number of edges, and each edge has a random cost. The purpose of doing this is to make the graph unpredictable to check the efficiency of the algorithms.

Algorithms

We are supposed to test the working of A* and Dijkstra for computing the shortest path. For the Salt Lake City Map, the difference is not significantly visible. As the cost of each node is too small to have any consequence in the state of algorithms. The image below shows the working of both algorithms, along with memory consumption.

```
Enter Source and Destination in range from 0 to 32
Enter -1 for source and destination to exit
0 31
Time taken for finding path: 0
Number of nodes visited for finding path: 29
A* path using Manhattan Distance is: 0 -> 13 -> 20 -> 19 -> 31 ->
Virtual Memory Usage: 45 MB
Physical memory used: 10 MB

Time taken for finding path: 0
Number of nodes visited for finding path: 29
A* path using constant Distance is: 0 -> 13 -> 20 -> 19 -> 31 ->
Virtual Memory Usage: 45 MB
Physical memory used: 10 MB

Time taken for finding path: 0
Number of nodes visited for finding path: 29
A* path using Over Estimated Hueristic is: 0 -> 13 -> 20 -> 19 -> 31 ->
Virtual Memory Usage: 45 MB
Physical memory used: 10 MB

Time taken for finding path: 0.001
Number of nodes visited for finding path: 29
Dijkstra path is: 0 -> 13 -> 20 -> 19 -> 31 ->
Virtual Memory Usage: 45 MB
Physical memory used: 10 MB
```

Working of A* and Dijkstra (from node 0 to node 31)

Usually, A* is better than Dijkstra, but the Heuristic factor in the A* defines how better it is from Dijkstra. To test this point, I used random values for heuristic and run the algorithms twice; the following are images of my findings.

```
Path Finding on SLC Map Graph....
Enter Source and Destination in range from 0 to 32
Enter -1 for source and destination to exit
5 20
Time taken for finding path: 0
Number of nodes visited for finding path: 8
A* path using Random Hueristic is: 5 -> 13 -> 20 ->
Virtual Memory Usage: 47 MB
Physical memory used: 70 MB

Time taken for finding path: 0.001
Number of nodes visited for finding path: 13
A* path using constant Distance is: 5 -> 13 -> 20 ->
Virtual Memory Usage: 47 MB
Physical memory used: 70 MB

Time taken for finding path: 0.001
Number of nodes visited for finding path: 13
A* path using Over Estimated Hueristic is: 5 -> 13 -> 20 ->
Virtual Memory Usage: 47 MB
Physical memory used: 70 MB

Time taken for finding path: 0
Number of nodes visited for finding path: 13
Dijkstra path is: 5 -> 13 -> 20 ->
Virtual Memory Usage: 47 MB
Physical memory used: 70 MB
```

A* performing better than Dijkstra

```
5 20
Time taken for finding path: 0
Number of nodes visited for finding path: 15
A* path using Random Hueristic is: 5 -> 13 -> 20 ->
```

A* performing worse than Dijkstra (only showing A* because the output of Dijkstra is same as the image above it)

For Large graphs, the results were completely different from the small graph. This test showed how great A* is in comparison to Dijkstra. A graph with 6000 nodes and a max of 30 edges connecting each node created a jumbled graph of multiple paths between two nodes. In terms of complexity, this graph was on a different scale than the small graph used before. Here are the images of the test runs.

```
Enter number of Nodes to create Graph, minimum 1000 max 10000 : 6000
Path Finding on Large auto generated Graph....
Enter Source and Destination in range from 0 to 5999
Enter -1 for source and destination to exit
56 1240
Time taken for finding path: 1.033
Number of nodes visited for finding path: 531
A* path using Manhattan Distance is: 56 -> 2238 -> 2636 -> 3832 -> 2608 -> 1240 ->

Time taken for finding path: 1.426
Number of nodes visited for finding path: 705
A* path using Euclidean distance as Hueristic is: 56 -> 2238 -> 2636 -> 3832 -> 2608 -> 1240 ->

Time taken for finding path: 11.716
Number of nodes visited for finding path: 3632
Dijkstra path is: 56 -> 2238 -> 2636 -> 3832 -> 2608 -> 1240 ->
```

A* and Dijkstra on a large graph

It is clear from the above image that A* took 1.033 and 1.426 CPU milliseconds for Manhattan and Euclidean Heuristic, respectively. On the other hand, Dijkstra took 11.716 CPU milliseconds, which is more than ten times the Manhattan and almost 10-time Euclidean Heuristic. Similarly, Manhattan Heuristic searched 531 nodes, whereas

Dijkstra searched 3632 nodes, which is more than half the nodes in the graphs. So, while Dijkstra searched half the graph for the path, A* using Heuristics get the same path by only searching less than 10% of total nodes in the graph. Here are a few more images to support the claim.

```
Path Finding on Large auto generated Graph....
Enter Source and Destination in range from 0 to 5999
Enter -1 for source and destination to exit
0 5999
Time taken for finding path: 0.294
Number of nodes visited for finding path: 62
A* path using Manhattan Distance is: 0 -> 378 -> 3187 -> 5999 ->

Time taken for finding path: 0.46
Number of nodes visited for finding path: 116
A* path using Euclidean distance as Hueristic is: 0 -> 378 -> 3187 -> 5999 ->

Time taken for finding path: 10.602
Number of nodes visited for finding path: 3355
Dijkstra path is: 0 -> 378 -> 3187 -> 5999 ->
```

A*: 1% node search, Dijkstra: 50% node search

```
Enter number of Nodes to create Graph, minimum 1000 max 10000 : 10000
Path Finding on Large auto generated Graph....
Enter Source and Destination in range from 0 to 9999
Enter -1 for source and destination to exit
65 8564
Time taken for finding path: 8.934
Number of nodes visited for finding path: 1173
A* path using Manhattan Distance is: 65 -> 115 -> 1560 -> 2188 -> 4964 -> 6425 -> 8564 ->

Time taken for finding path: 13.273
Number of nodes visited for finding path: 1737
A* path using Euclidean distance as Hueristic is: 65 -> 115 -> 1560 -> 2188 -> 4964 -> 6425 -> 8564 ->

Time taken for finding path: 44.801
Number of nodes visited for finding path: 8682
Dijkstra path is: 65 -> 115 -> 1560 -> 2188 -> 4964 -> 6425 -> 8564 ->
```

A*: 11% node search, Dijkstra: 86% node search

Heuristics

Heuristics is the most crucial factor that affects and differentiates A* from Dijkstra. A* is guaranteed to be more efficient if the heuristics satisfy two conditions:

- It is Admissible: The heuristics never overestimate the cost.
- It is consistent: The heuristics should satisfy the triangle inequality.

For analyzing the heuristics, I used my large graph over the small graph, as the large graph provides a better analysis than the small one. Also, for the small graph, even a handwritten heuristic would work better than calculating the heuristic values. For my analysis, I worked on three different heuristics.

- Constant Value as a Heuristic.
- Distance Value as a Heuristic.
- Random Value as a Heuristic.

Constant Value as a Heuristic

Constant value as a Heuristic is just using a similar constant value of h(x) for all nodes in the graph. Using constant value as heuristic was only marginally better than Dijkstra in terms of CPU cycles, and similar to Dijkstra in terms of the number of nodes visited. Which makes sense as f(x) = g(x) + c, where h(x) = constant(c). if c = 0 than f(x) = g(x), which is the formula Dijkstra. Thus the value of c has no impact on the actual running of the algorithm as every node has the same constant value added to them, which is a distribution of cost across all nodes, which technically only affects the cost.

Distance Value as Heuristics

I used Manhattan distance and Euclidean distance as heuristics to analyze how distance affects the heuristic value. The analysis showed that the Manhattan distance always outperforms the Euclidean distance, **provided that both distances are admissible**. At first, I thought it would be because Manhattan Distance was used to generate the cost of the edge. And on changing the cost to Euclidean distance, Euclidean distance was efficient. But the heuristic generated by Manhattan distance was not admissible. So, Manhattan would always be non-efficient. This test proved that for an algorithm to be efficient, the heuristic value should be high and close to the distance between source and goal. For example, if D is the total cost between two nodes and h1(x) and h2(x) are two heuristic functions than if $D \geq h1(x) > h2(x)$ than h1(x) would always be more efficient than h2(x), the same is there for Manhattan distance(MD) and Euclidean Distance(ED), MD > ED and so MD outperforms ED.

Random Value as Heuristic

Using random value as a heuristic is self-exploratory, which is to use random numbers from a uniformed int distribution. But to have more control over the numbers and analyze the efficiency correctly, the random numbers were generated using a closed range. Following ranges have been tested for testing the efficiency for algorithms:

- 0 – 100.
- 0 – 1000[1].
- 0 – Number of Nodes specified by the user.
- 0 – Max stored value (UINT64MAX for 64 bits and UINT32MAX for 32 bits)[2].
- Manhattan Distance + range #2[1].
- Manhattan Distance + range #4[2].

Using these ranges allows checking how A* works when the heuristic values transit from an admissible value of heuristic to a non-admissible value. The finding is compared with Manhattan Distance as the lower bound and Dijkstra as upper bound.

```
Enter number of Nodes to create Graph, minimum 1000 max 10000 : 6000
Path Finding on Large auto generated Graph....
Enter Source and Destination in range from 0 to 5999
Enter -1 for source and destination to exit
0 5999
Time taken for finding path: 1.079
Number of nodes visited for finding path: 265
Total Cost: 1599
A* path using Manhattan Distance is: 0 -> 32 -> 279 -> 613 -> 2531 -> 3352 -> 5999 ->

Time taken for finding path: 13.592
Number of nodes visited for finding path: 4543
Total Cost: 1599
A* path using random Distance using 0 to 100 range is : 0 -> 32 -> 279 -> 613 -> 2531 -> 3352 -> 5999 ->

Time taken for finding path: 10.68
Number of nodes visited for finding path: 3373
Total Cost: 1892
A* path using random Distance using 0 to 1000 range is : 0 -> 1614 -> 3586 -> 3322 -> 5155 -> 5999 ->

Time taken for finding path: 4.115
Number of nodes visited for finding path: 1156
Total Cost: 2294
A* path using random Distance using 0 to number of nodes range is : 0 -> 2643 -> 1379 -> 4874 -> 5999 ->

Time taken for finding path: 2.546
Number of nodes visited for finding path: 715
Total Cost: 3646
A* path using random Distance using 0 to max range is : 0 -> 3133 -> 4646 -> 2141 -> 1376 -> 5999 ->
Time taken for finding path: 14.131
Number of nodes visited for finding path: 4752
Total Cost: 1599
A* path using Random Manhattan Distance using 0 to 1000 range is: 0 -> 32 -> 279 -> 613 -> 2531 -> 3352 -> 5999 ->

Time taken for finding path: 13.935
Number of nodes visited for finding path: 4713
Total Cost: 1599
A* path using Random Manhattan Distance using 0 to max range is: 0 -> 32 -> 279 -> 613 -> 2531 -> 3352 -> 5999 ->

Time taken for finding path: 13.612
Number of nodes visited for finding path: 4675
Total Cost: 1599
Dijkstra path is: 0 -> 32 -> 279 -> 613 -> 2531 -> 3352 -> 5999 ->
```

A* on the ranges of heuristics defined above

From the analysis, it became clear that the first two ranges give the shortest path as they are admissible. But performance-wise, they are not much better than Dijkstra. Also, the second one always outperforms the first as the average length of the first range would always be less than the average length of the second range, similar to how Manhattan outperforms Euclidean. The next two ranges performed well in terms of CPU utilization, but the paths were not optimal, the reason being the heuristics were not admissible and were overestimating. At the same time, the last two ranges performed worse than Dijkstra in terms of CPU utilization but gave the optimal path. This analysis resulted in the conclusion that if the heuristic overestimates, then the resulting path can be non-optimal or can be CPU heavy or both.

Combining Pathfinding With Movement

For the final part, We have to use the pathfinding algorithm created above in world space with obstacles and allow a player to move around the obstacles. I decided to create a dynamic world space with random obstacles scattered around the world. This method has more edge cases than having a static maze, thus allowing me to have a more robust dynamic path-following algorithm. Also, we have analyzed the efficiency of the algorithm in previous sections; So, I am using Manhattan distance for finding the path.

For the division scheme to quantize my world space to graph space and localized my graph space to world space, I am using the Tile-Based division scheme. I have a *"TileGraph"* class that handles the division scheme. It takes in the following 5 parameters to generate Tiles in world space to run the pathfinding algorithm:

- Width of Window.
- Height of Window.
- Width of Tile.
- Height of Tile.
- Number of Obstacles.

This class creates a graph from the above information with the following considerations:

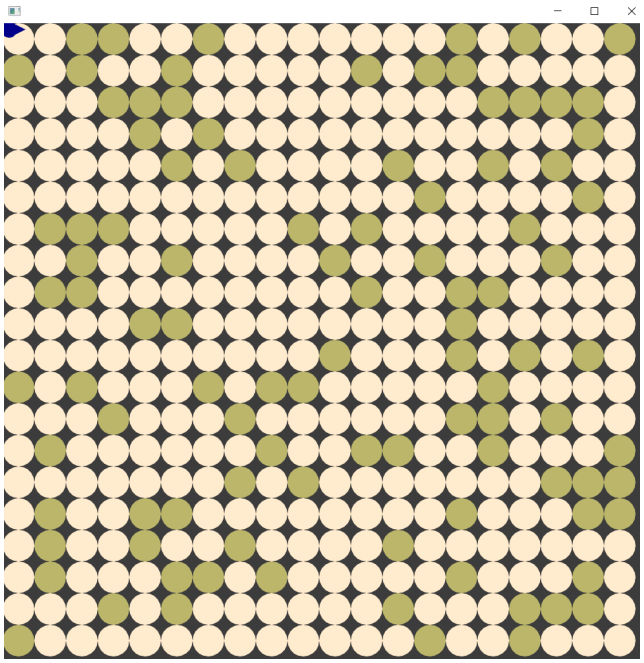
- Each adjacent tile, excluding diagonals, creates a path.
- The cost of the path to another tile is 1, and 0 if the tile is an obstacle.

Clicking anywhere on the map tries to generate a path from the player's position to a clicked position using the A* algorithm. The *"DynamicPathFollow"* class uses this path to handles the movement of the player from his location to the targeted location. This movement delegates to the *"DynamicArrive"* movement algorithm from the previous assignment. A few problems are using this representation are as follows:

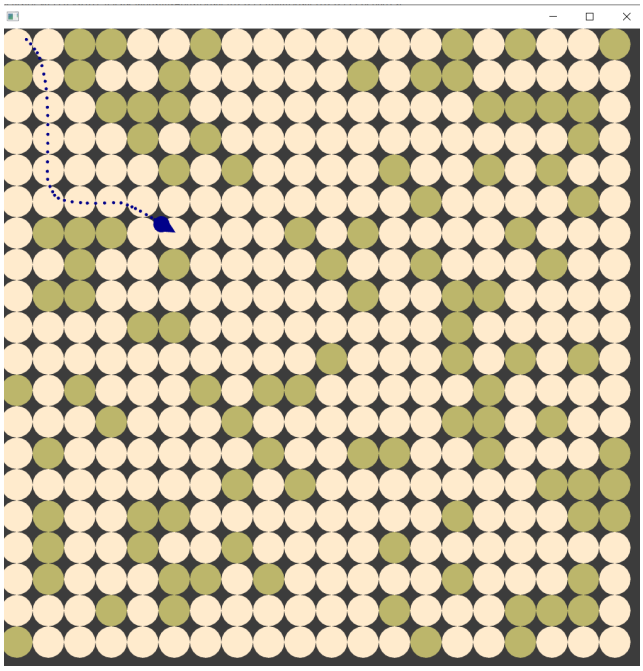
- Clicking on obstacles does not return any path.
- The player may be blocked by the obstacles when the game starts.

Changing the algorithm to return a path closest to the clicked obstacle solves the first problem. Changing the algorithm, however, is a bit tedious and not required for the assignment, so I am skipping that. For the second problem, while creating the obstacles, we can check if the obstacle is created on player location than don't create that node.

In the graphical representation, I am using a *"blanched almond"* color for the node that can be walked by the player and *"Dark Khakhi"* color for obstacles.

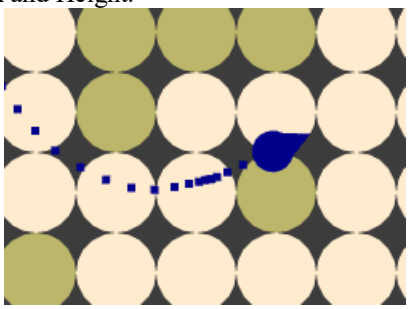


Tile Graph Representation



Player finding the path and moving toward the destination

The DynamicPathFollow algorithm also uses a soothing radius that is similar to the slow and target radius of the DynamicArrive algorithm. Still, instead of slowing down the player, the soothing radius gives the next target to arrive in the path towards the goal, making the player move like a curve rather than a sharp turn when we arrive at the target. Also, if the soothing radius is less than the target radius, then the player can never be able to calculate the next target in the node and can only move after finding a new path. The soothing radius also depends on the Tile Height and the Tile Width; if the radius more then the player might curve around an obstacle. So for the best result, the soothing radius needs a value that is near and less than the Tile Width and Height.



Player going over an obstacle as the soothing radius is more than tile width and height