Pathfinding
Pinak Jalan
U0965201

Abstract:

This document is a reflection of the implementation of pathfinding algorithms using weighted graphs. Random graphs have been generated and have had A* and Dijkstra's algorithm applied on them to find the most optimal path with the least amount of time. Then, using Tile Based division scheme, we quantize and localize the graph for finding a way that is then used by our boid to arrive dynamically at a point defined dynamically by us on said graph by a mouse click.

First Steps:

There are 3 graphs used in this pathfinding assignment. I have obtained one of them with the help of Luis - the grid graph. The graph of Rome, Italy was gotten from the website suggested in class, which represents intersections and streets. The other custom graph was a result of random numbers and help from Luis by using his house as a reference. The Grid graph is a 16x9 graph. All of these graphs are made in a csv format and parsed using c++'s own ifstream class. The source represents the first node, the sink the second and the cost accrued is the third(-1 on a sink is an obstacle).

I have used Dijkstra's algorithm, A star as well as other dynamic movement algorithms to achieve pathfinding in a weighted graph from node a to b. The results will be described below.

Dijkstras Algorithm and A*

Everything was stored in STL's vector class, because of all the helper methods(size(), push and pop) available. Since I was measuring time in ms(milliseconds), the first graph was done in 0 milliseconds(as per image 2) because it contained about 20-30 nodes for both dijkstras algorithm and A*, while the second one took about ~285 ms for Dijkstras on the big graph of Rome, where as for A*, surprisingly, it took MORE time. It took about ~360 ms for the constant heuristic A* pathfinding simulation, while it took even more time (~420 ms) for random heuristic values on the same graph. After researching

and further thinking, it makes sense because heuristics for calculation imply more memory usage which would mean more processing time. And if there is a random number generator being called everytime for a heuristic that would increase computation time even more. A Star, however, on the larger graph managed to find a shorter path between the shorter vertices with a heuristic as compared to Dijkstra's(explanation for that will be below).

Heuristics

There were 3 different types of heuristics i used for different weighted graphs. They include constant vectors of ints, random vectors of ints, and a vector of manhattan distances. Funnily enough, the random heuristic visited more nodes as compared to the constant heuristic and sometimes give fewer nodes for the shortest path. This was inconsistent though, because sometimes it would overestimate and even give longer paths. Constant heuristic was a lot more consistent, but on average because of the addition of another parameter in calculating cost, namely the heuristic, the nodes on average that were visited to calculate the shortest path were more, even though the overall shortest path generated was better than that of Dijkstras.

I also read about the concept of "admissibility", and I learned that a given heuristic is admissible if and only i the heuristics never overestimate cost, and they are consistent if and only if they satisfy the triangle inequality. I learned that both the manhattan and euclidian distances are also good, and that the manhattan distance outperfoms euclidian, in an ideal world where both of them are admissible. Also, even though it didnt occur to me earlier, the constant heuristic distance is just an unnecessary memory overhead added as a heuristic calculation, because if f(x) = g(x) + c where c is cost and c is always the same, we can cancel it out, and in that regard isnt A* basically Dijkstra's algorithm?

Putting it all together

For the last part, I used lines and circles inbuilt in open frameworks to draw a grid based on grid values from a graph like was suggested in class(the cost to every single connection is 1). I used random spots and experimentation to create obstacles(aforementioned circles) but experimented enough to the point that there were many different paths the boid could take to find the shortest path it could take. Unfortunately even though the path for the boid to take on a mouse click was being

generated, I was having a small bug and a quirk where the boid crashes will trying to go there(see image 2 for the path being generated on mouse click, it is the one with the manhattan distance and the 6th spot from the left on the first row was clicked). I made a 16 x 9 graph, because this was the easiest and each box could occupy pixels on the screen(16*120 = 1920 and 9*120 = 1080) which gave us a big enough graph to work with with many different possibilities. I used the manhattan distance as a form of heuristic in this specific instance because from my experience it is the most commonly used form of heuristic and it is pretty easy to calculate given the player and a target point. I created a Grid class with the help of Luis which helped me get the exact point for a click and i used dynamic arrives and orientation centering("Look where you are going") to help me get there.

There were actually a lot of problems I was having with this implementation aside of the game breaking bug where the path was printed but when the boid tried to move the game crashed. I had to mess a decent amount with the values for arrive, because sometimes my boid would just stop(I took the paths i got and paper debugged the answers). I figured it was because it was getting stopped by an obstacle because of an error mismatch in the graph vs what was actually drawn and I figured out my dynamic arrive was getting too short of a linear radius. Once that was fixed there were issues where the boid would just stop when it arrived near a circle(obstacle). After debugging I also figured out that getting a better linear steering output(upping the input values) would actually fix the issue. While working on this initially, I had a lot of friends who were using euclidean distance as opposed to my manhattan distance approach. Some of them were getting weird behaviour where their boids started to move in weird diagonal paths instead of using straight line paths to get to the end node. I never had this problem and for the longest time could not figure out what the problem was until recently when i figured out the euclidian distance prioritizes nodes in its diagonals rather than straight ones. Sometimes this might even affect the shortest path.
The one other thing that i noticed was that sometimes i would get more nodes visited for a particular traversal as compared to another with the same source and goal. This was really confusing for me until I started debugging, when i found out that sometimes there are certain edges that have a combined same total cost(with a constant and unchanging heuristic). As a very small example, getting from node 1---> 18 can be 1 - 6 - 8 - 13 - 11 - 7- 19 as well as 1- 9 - 11 - 15 - 8 - 19. Since the total cost is the same, it doesnt matter which path we take; obviously in my opinion for memory management fewer nodes the better, but in this case we are good.

Appendix:

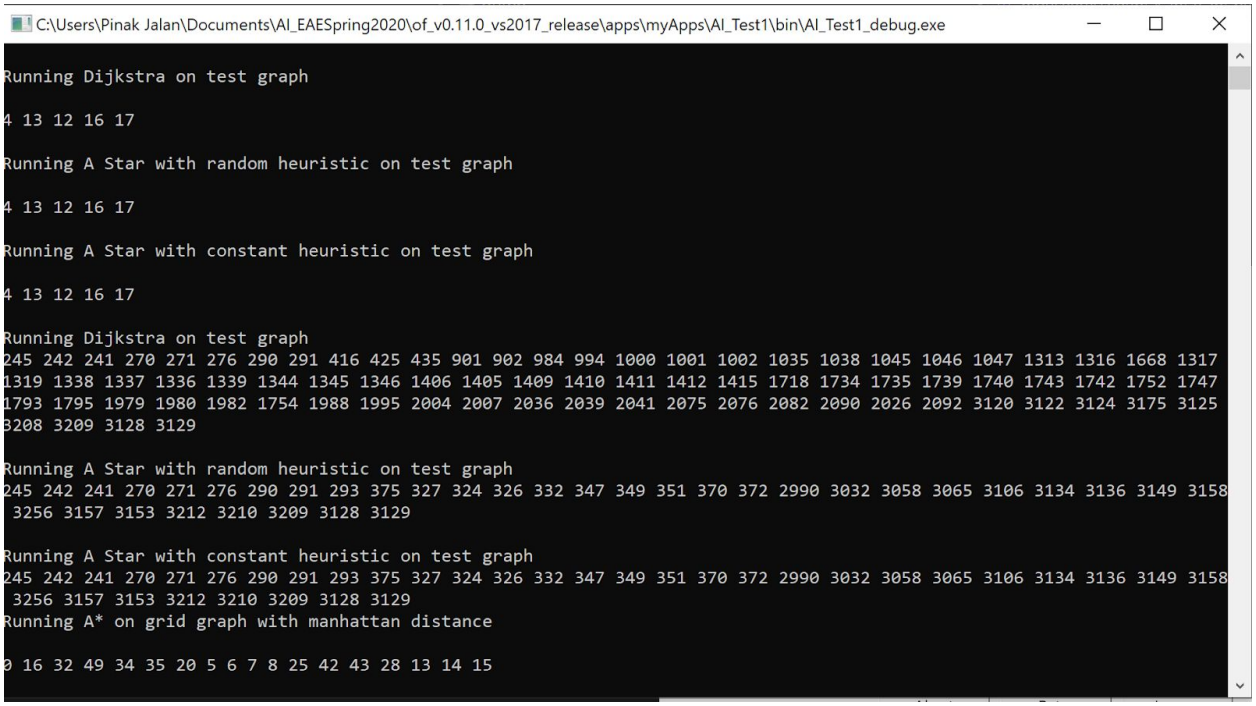

Image 3

Scroll down for more images.

```
4 13 12 16 17

Running A Star with random heuristic on test graph

4 13 12 16 17

Running A Star with constant heuristic on test graph

4 13 12 16 17

Running Dijkstra on test graph
245 242 241 270 271 276 290 291 416 425 435 901 902 984 994 1000 1001 1002 1035 1038 1045 1046 1047 1313 1316 1668 1317 1319 1338 1337 1336 1339 1344 1345 1346 1406 1405 1409 1410 1411 1412 1415 1718 17
34 1735 1739 1740 1743 1742 1752 1747 1793 1795 1979 1980 1982 1754 1988 1995 2004 2007 2036 2039 2041 2075 2076 2082 2090 2026 2092 3120 3122 3124 3175 3125 3208 3209 3128 3129

Running A Star with random heuristic on test graph
245 242 241 270 271 276 290 291 416 425 435 901 902 984 992 993 999 1089 1097 1109 1130 1134 1135 1136 1160 1170 1173 1175 1825 1845 1847 2312 2318 1861 2324 2326 2327 2323 2329 2328 2352 2361 2362 2363
 2364 2436 2442 2443 2463 2485 2469 2468 2470 3124 3175 3125 3182 3211 3209 3128 3129

Running A Star with constant heuristic on test graph
245 242 241 270 271 276 290 291 293 375 327 324 326 332 347 349 351 370 372 2990 3032 3058 3065 3106 3134 3136 3149 3158 3256 3157 3153 3212 3210 3209 3128 3129

Running A Star with constant heuristic on test graph
245 242 241 270 271 276 290 291 416 425 435 901 902 984 992 993 999 1089 1097 1109 1130 1134 1135 1136 1160 1170 1173 1175 1825 1845 1847 2312 2318 1861 2324 2326 2327 2323 2329 2328 2352 2361 2362 2363
 2364 2436 2442 2443 2463 2485 2469 2468 2470 3124 3175 3125 3182 3211 3209 3128 3129
```

Image 2

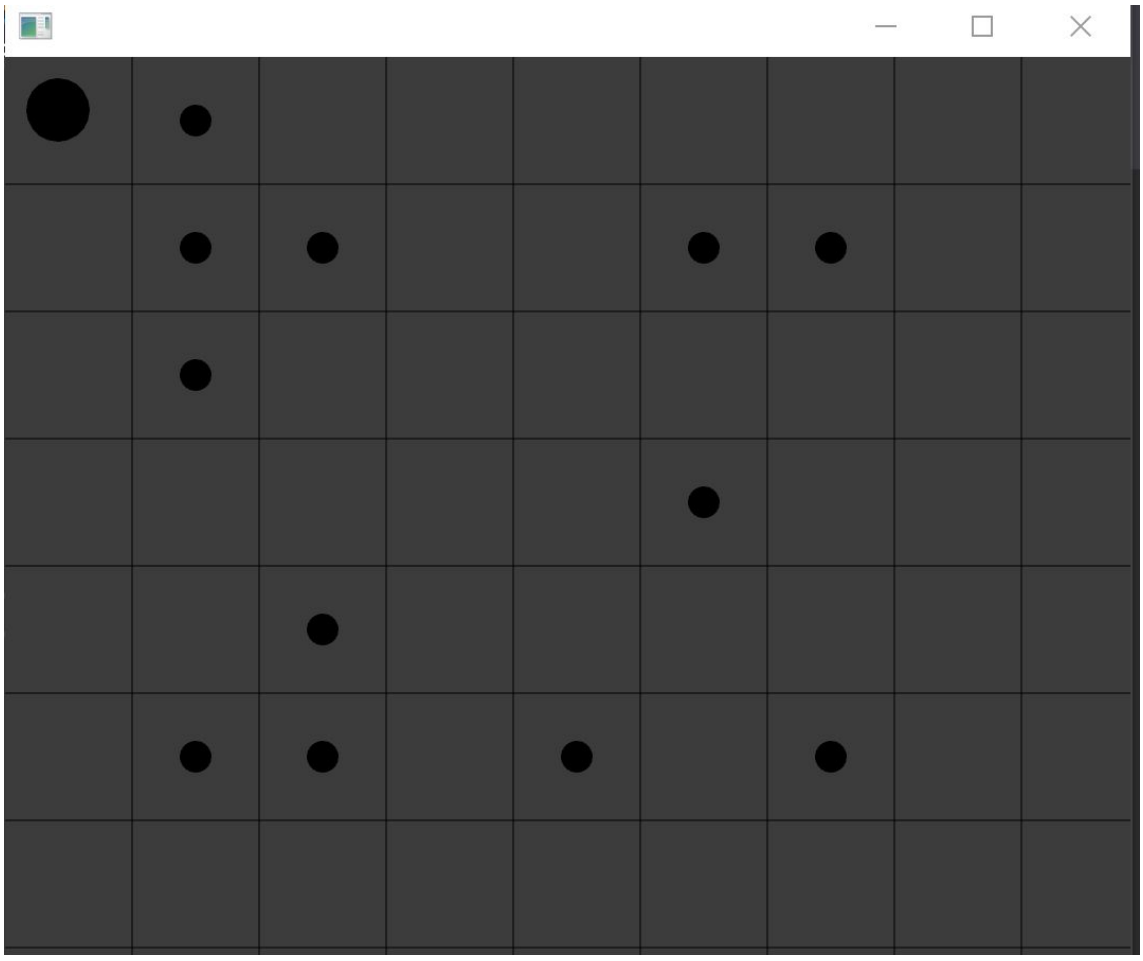

Image 1: Obstacle graph