EAE6610 Assignment 2 Report

Jonah Brooks
Entertainment Arts & Engineering (EAE)
University of Utah
Jonah.Brooks@utah.edu

ABSTRACT
This is the report for assignment 2 of EAE6610, AI for Games. In this report I detail how I approached the assignment of creating and using pathfinding algorithms for simple AI in an openFrameworks C++ environment and the explorations I did to better understand the material. I discuss how I generated and stored directed weighted graphs, how I implemented Dijkstra's and A* algorithms, and how I put it together with the movement algorithms I learned during the last assignment.

Keywords
AI for Games, EAE6610, Pathfinding Algorithms, A*, Dijkstra's

INTRODUCTION
This assignment focused on creating and using directed weighted graphs, pathfinding algorithms, and heuristics. In this assignment I was tasked with making, storing, and using graphs, as well as implementing Dijkstra's and A* algorithms. I was also tasked with tying this together with the movement algorithms I implemented in the last assignment.

GRAPHS
Details on how I generated and stored the directed weighted graphs can be found below.

Representation
For representing the graphs, I chose to make a Graph class that stores an adjacency list for each node. Initially I tried to use a simple 2D array or vector with the sentinel value of -1 to represent no connection, but the memory usage was extremely wasteful, and adjacency was not easy to query. Ultimately, I chose to use a vector of vectors of int pairs to represent adjacency, with the first entry representing the destination node and the second representing the weight of the connection.

Small Graph
I hand made one small graph consisting of 20 nodes that represents downtown Salt Lake City. This graph was primarily made for testing, as I can get a better idea of how the pathfinding works if I know what it represents and can trace it through by hand.

Large Graph
I also implemented a larger graph of 2000 nodes to stress test the algorithms on a larger scale. This graph consisted of the first 2000 nodes found in the New York City graph found at http://www.diag.uniroma1.it//~challenge9/download.shtml. Initially I tried to use the full graph with 200000+ nodes, which was the main impetus behind revising my graph data structure to not store -1 for each missing edge, as doing so would have taken over 260 GB

of RAM, according to my estimates. However, it still takes too long to build a graph of that size, so even with the optimized data structure I chose to truncate the graph to its first 2000 nodes.

ALGORITHMS

To use those graphs, I implemented a search function with adjustable heuristic selection to accomplish A* and Dijkstra's Algorithms.

Heuristics

Dijkstra's obviously uses a constant heuristic of 0 as its estimated cost to the goal. For A*, I implemented two simple heuristics for testing. One makes a constant guess of 1 and the other uses the minimum distance between any two nodes as its guess. I chose those two methods due to ensuring that they will always be admissible and consistent, even though they will both grossly underestimate the actual distance to the goal.

Performance

I ran sequences of random walkthroughs of each graph with each heuristic type and observed the following. I wasn't sure how to get memory usage, but the time and nodes visited ended up similar across all the heuristics. This is likely due to the differences between my selected heuristics not being large enough.

Salt Lake City (small) Graph

For this graph, I found the following results:

```
SLC graph with A* GuessMinimumEdgeWeightHeuristic:
        Walkthroughs: 1000
        Total Nodes Visited: 10508
        Average Nodes Visited: 10.508
        Total Time Taken: 0.0924935
        Average Time Taken: 9.24935e-05

SLC graph with A* Guess1Heuristic:
        Walkthroughs: 1000
        Total Nodes Visited: 10207
        Average Nodes Visited: 10.207
        Total Time Taken: 0.0858592
        Average Time Taken: 8.58592e-05

SLC graph with Dijkstra's:
        Walkthroughs: 1000
        Total Nodes Visited: 10350
        Average Nodes Visited: 10.35
        Total Time Taken: 0.0834697
        Average Time Taken: 8.34697e-05
```

Figure 1: The statistics for randomly walking through the SLC graph 1000 times for each algorithm.

For this graph, the minimum distance is 1, so the first two heuristics should be the same and slightly different than Dijkstra's Algorithm. 1000 samples is a fairly small size, but it looks like for this graph the three heuristic methods are very similar.

New York City (large) Graph

For the larger NYC graph, I found the following results:

```
NYC graph with A* GuessMinimumEdgeWeightHeuristic:
        Walkthroughs: 1000
        Total Nodes Visited: 876228
        Average Nodes Visited: 876.228
        Total Time Taken: 28.7605
        Average Time Taken: 0.0287605

NYC graph with A* Guess1Heuristic:
        Walkthroughs: 1000
        Total Nodes Visited: 871800
        Average Nodes Visited: 871.8
        Total Time Taken: 28.6308
        Average Time Taken: 0.0286308

NYC graph with Dijkstra's:
        Walkthroughs: 1000
        Total Nodes Visited: 887554
        Average Nodes Visited: 887.554
        Total Time Taken: 29.1017
        Average Time Taken: 0.0291017
```

Figure 2: The statistics for randomly walking through the NYC graph 1000 times for each algorithm.

In this graph, the minimum distance between two edges is 24, so slightly different than 1 or 0, but probably not significantly better of a heuristic. The performance was similar across all three, with Dijkstra's being slightly slower and with more nodes visited.

PUTTING IT ALL TOGETHER

The final task of this assignment was to tie in the pathfinding to the movement algorithms from last assignment to create an obstacle avoiding, pathfinding agent.

Grid and Indoor Environment

I chose to use a grid-based graph for the level. I first created a complete grid with edges connecting every node to its adjacent nodes. From there, I wrote a function that will draw a wall of the desired dimensions at a given location and automatically remove the edges it cuts off from the graph. This lets me easily place walls wherever I desire, and the pathfinding will adapt to it automatically. I did slow down the movement of the Boid considerably to keep it from overshooting each point along the path. I would like to have found a better solution for that, but this certainly works as desired, even if it is a bit slow.

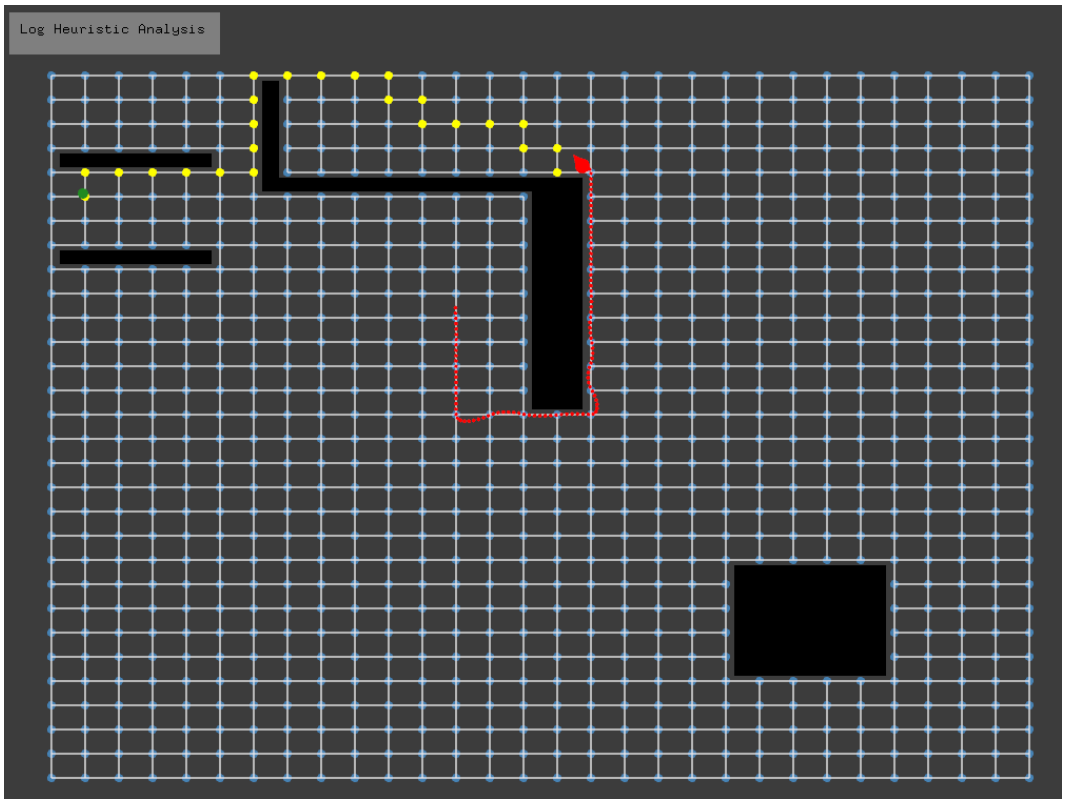

Figure 3: Image of the indoor environment and A* pathfinding. The yellow dots represent the path found by A* to reach the mouse click represented by the green dot.

CONCLUSION

In all, I had a lot of fun with this assignment. It all came together so well in the end, and it was cool to see an agent pathfinding around an environment I made. I'm especially proud of the wall creation code that automatically modifies the graph to adjust the pathfinding. I could have added any number of walls, but I figured what I had was enough to show how it all works.