CSC 484 - Path Finding and Path Following Analysis

Jason Nguyen

I created a graph structure which consists of mostly two core components. An unordered map contains a pair of integers and a Vertex struct, which assigns its id to information of each Vertex. A second unordered map named adjList represents an adjacency list. The key is the id of the vector, with the value containing a pair of integers, representing the id of the adjacent vertex, and a float representing the weight it costs to reach the adjacent vertex. The graph also uses an abstract function variable called heuristicFunc, which stores a static function to perform different heuristic functions based on the user's choice. A red square represents the vertex the entity is trying to reach. A green square represents the goal vertex. The entity will only be using Arrive and Align for better visual clarity. Vertices are clickable, which sets the targeted vertex as the goal vertex the entity will attempt to reach.

Two graphs were created for the first stage of development. By hand, I created a list of 48 different positions representing the location of the mainland states of the U.S.A. I then created an additional list representing the edges of the vertices. These edges represent the state's borders. After creating all 48 vertices and respected borders, I created vertex 0 which is just a random vertex that only connects to vertex 1, which represents Washington state. The choice was simple as it fitted the requirements of 20-50 vertices. Using 49 different vertices and a total of 102 different edges, some vertices had over 8 adjacent vertices. Each position is placed at the capitals of each state instead of the center of the state. This does make some vertices closer to each other, as the capitals are relatively closer to each other. The image created by the vertices and edges do resemble the United States. But with some states not bordering each other, some gaps are formed.

The second graph is a randomly generated graph. The algorithm is simple, creating a number of vertices that fit within a 1000x1000 window. For homework submission, I created 500 vertices and 5000 vertices, but can go up. The main drawback of increasing the number of vertices would be time it took to render each frame. The entity would visually be stuttering across the screen. Along with the number of vertices, it becomes difficult to see all of the connected vertices as the whole screen is cluttered in vertices. Edges are also randomly generated. A random float between 3-4 is created which represents the number of edges compared to the number of vertices. If the float is 3.5, there will be a total of 3.5 times more edges than vertices. Edges are randomly assigned, but each vertex will only have a maximum of 9 edges.

The project is able to use 2 different search algorithms, that being Dijkstra and A* algorithms. The Dijkstra algorithm explores all possible paths in order to find its goal. This means that performance is slow with larger graphs with more vertices and edges. If the goal of the algorithm was to explore all possible paths, then the Dijkstra algorithm would be a great fit.

A* algorithm takes a different approach, which takes the cost of actually traveling to the node and a heuristic cost. A* reduces the number of nodes needed to explore which becomes more efficient in searching for the goal.

3 different heuristics were created for the project. The Manhattan heuristic measures the shortest path in a grid based system, focusing only on vertical and horizontal movement. This system is best designed for grid based systems, but as our randomly generated graph is more open, this algorithm is not best suited. Because of Manhattan heuristic determining distance based on a grid, the distance it estimates for our States graph becomes inadmissible. Many of the distances become over estimations due to following a more diagonal border. The left side of the graph does have more horizontal and vertical edges, but when pathing towards the right side of the graph, the Manhattan heuristic tends to overestimate the cost of traveling to the node.

Euclidean heuristic measures the distance between two vertices in a straight line. This heuristic is more suitable in our situation in a continuous space where diagonal movement is allowed. This is the most accurate algorithm of the three, but becomes more computationally expensive due to the square root function. With the straight line distance, the Euclidean heuristic is an admissible heuristic, as it does not overestimate the distance.

The last heuristic is the squared euclidean heuristic. The squared euclidean heuristic is the same as the euclidean heuristic, only it does not use the square root function. This does mean the exact distance of the two vertices are not recorded, but the rankings relative to the other nodes is still consistent. Without using the square root function, computation time is reduced, making the algorithm more efficient.

Below represents a table of the heuristic functions cost between two vertices. The main talking point would be the cost between Manhattan and Euclidean heuristics. For the relatively straight paths, we see a slight increase in the cost of the Manhattan heuristic. Though minimal, this isn't compared to the cost of a diagonal path, which is greatly larger compared to the Euclidean heuristic. This could result in different paths being taken, which may be inefficient.

Paths	Manhattan Cost	Euclidean Cost	Squared Euclidean
22->17 (Relatively Straight)	135	120.934	14625
17->11 (Diagonal)	246	175.071	30650
11->16 (Relatively Straight)	155	144.42	20857
16->21 (Diagonal)	101	89.805	8065
21->17 (Relatively	182	128.888	16612

Straight)			

The final graph created is a room based graph. Four rooms are created, each with a path connecting between adjacent rooms. Each vertex is placed along a coordinate plan. Additional obstacles are placed within each room which can help create judgement in the algorithm to find the best solution in reaching the node. Due to the grid based nature of the room, the most suitable algorithm would have been A* with a Manhattan heuristic, as that heuristic is most suited for a grid based layout.

Appendix

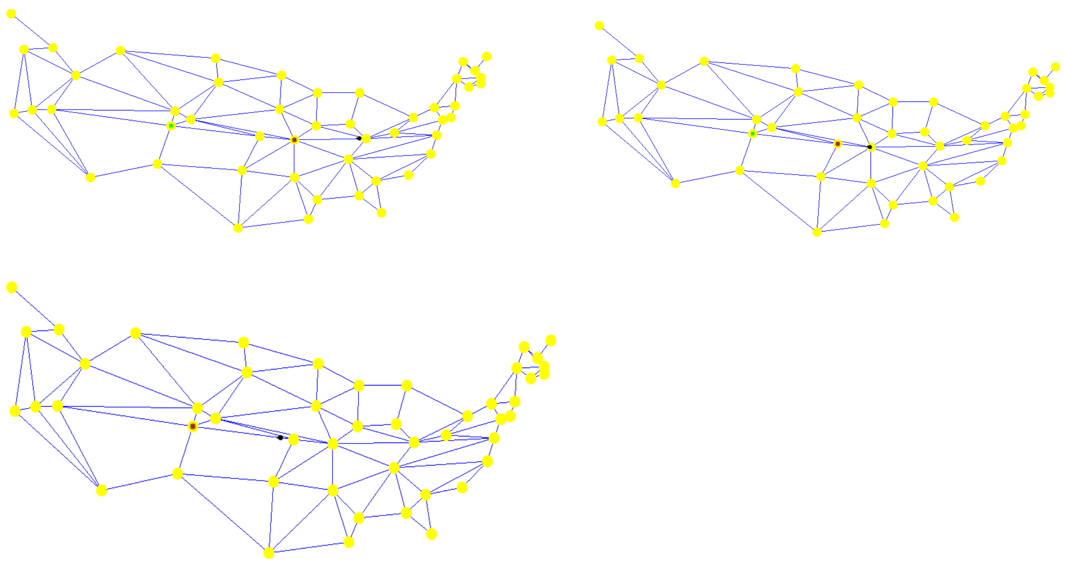


Figure 1: Images of an Entity following a path using Default Dijkstra Algorithm.

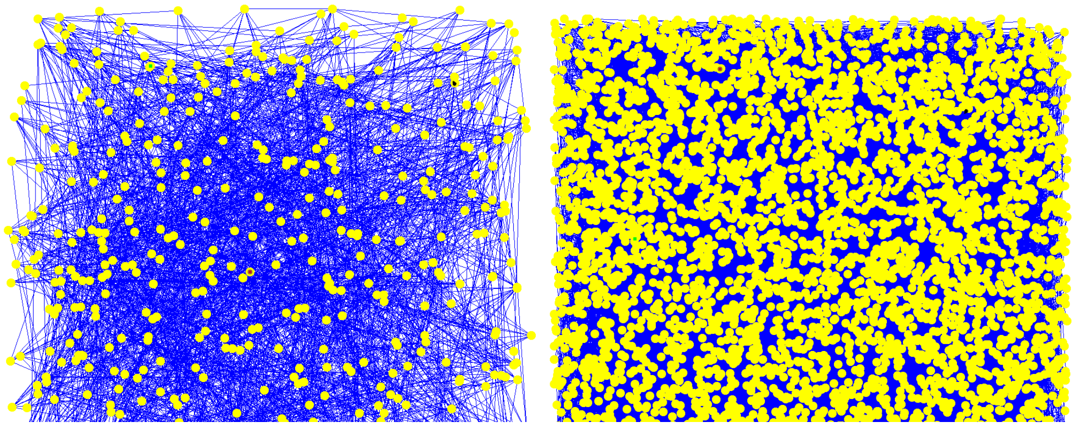
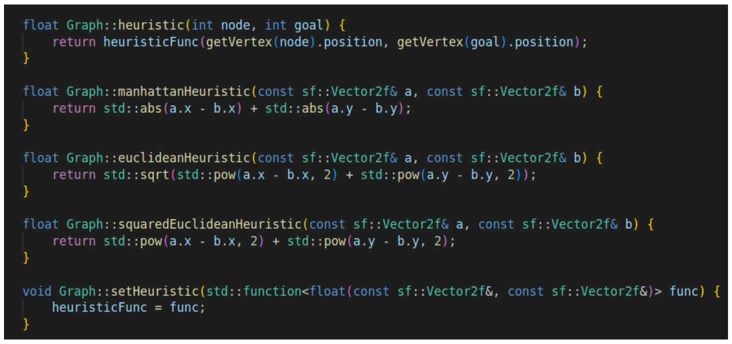


Figure 2: Images of randomly generated vertices and edges (500 and 5000 vertices).



```cpp
float Graph::heuristic(int node, int goal) {
    return heuristicFunc(getVertex(node).position, getVertex(goal).position);
}

float Graph::manhattanHeuristic(const sf::Vector2f& a, const sf::Vector2f& b) {
    return std::abs(a.x - b.x) + std::abs(a.y - b.y);
}

float Graph::euclideanHeuristic(const sf::Vector2f& a, const sf::Vector2f& b) {
    return std::sqrt(std::pow(a.x - b.x, 2) + std::pow(a.y - b.y, 2));
}

float Graph::squaredEuclideanHeuristic(const sf::Vector2f& a, const sf::Vector2f& b) {
    return std::pow(a.x - b.x, 2) + std::pow(a.y - b.y, 2);
}

void Graph::setHeuristic(std::function<float(const sf::Vector2f&, const sf::Vector2f&)> func) {
    heuristicFunc = func;
}
```

Figure 3: Image showing the different heuristic functions.

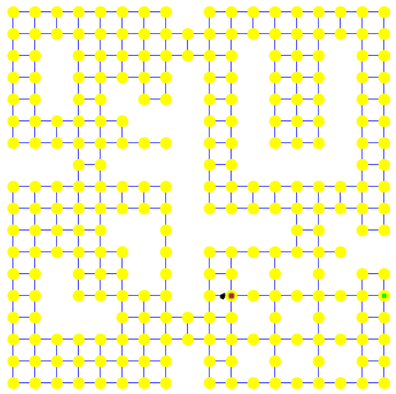


Figure 4: Image showing the grid based graph with obstacles.