Homework 3 Reflection

Before even deciding on what my first graph would be, I had to think of how I wanted my graph to be represented. I went off of the class textbook interface for the graph representation as that is the book then used when implementing Dijkstra's and A*. Some changes were mostly just things that I thought would be useful for the objects to have going down the line when creating the individual graphs as well as testing and using the pathfinding algorithms. The Vertex class holds an id number which is really just the index of the vertex in the list of vertices stored in the graph for ease of access and for more information when debugging. I also had the vertex store a list of outgoingEdges from it in order to help out the graph function that the interface had desired in the textbook which would get added to when the addEdge function was called, adding the newly created edge to the "from" vertex list. I also made these vertices hold a position, this was something that I had not initially planned for and just assigned weights (as you will see in my hand authored graph in the appendix, you can see the positions assigned in the code) but I learned that a position would prove very useful for the different A* heuristics as well as also streamlining the graph representation of the game world. The Edge class simply holds the to and from vertices as well as its weight. The Graph class really just holds a lot of functions that streamline the creation and information gathering of the graphs. I have a default constructor that just makes a graph with no vertices or edges and then a constructor that takes in a number of vertices and a number of edges. The latter constructor is what will generate a random graph based on the number or vertices and edges but I will go into that a little later. The Graph then has the capability of adding vertices to itself and putting it in the list of vertices for access later when adding an Edge (given two integer indexes for the vertices) as well as getting a vertex from the graph. The addEdge function also determines the weight of the Edge through just finding the distance between the positions associated with the vertices passed in.

The graph that I had authored was based off of the map in The Legend of Zelda Tears of the Kingdom (SEE PART 1 APPENDIX). I made a graph of the key locations in the game and the paths between them (adding some discrepancy myself to make the graph a little more interesting). I chose this as (other than being from a game I like) it made use of positional vertices which would match what I would be doing later in the homework assignment for the pathfinding in my game world. Then for my large graph, I created an algorithm to "randomly" generate a graph. The graph isn't completely random as I was worried about finding a way to ensure that the graph would be fully connected on every run of the program. I was thinking about making it so that you would check for full connection and then continue to add more edges and then check again later but that would be extremely computationally expensive. I couldn't figure out a way that wouldn't be very expensive in a random graph so I ended up generating the graph by connecting all the vertices in one big cycle, ensuring that a path would always exists between any two vertices and then for the remaining number of edges requested, they would be done so at random.

The implementation of Dijkstra's algorithm and the A* algorithms were both taken from the class textbook. In this I had to implement the Records and Pathfinding list classes. The record just contained information but the Pathfinding list had a little more nuance to what you could do with it. My structure for storing the records ended up being a vector that was sorted

from the smallest total estimated cost to the largest and was just done so by going up through the list and inserting the element at the correct spot in the vector. This isn't quite what a priority queue or priority heap is which is what the textbook had recommended for better runtimes and I didn't really want to create a whole priority queue class to be able to give it the ability to edit the list outside of just the first element so I just had the vector be sorted so that the getSmallest function would have O(1) runtime. The performance of the algorithms can be seen in the appendix (PART 2) where the graphs were run through the algorithms and their time to completion output. Dijkstra and A* were relatively close in the first small graph but then had a vast time difference for the big graph, with 100,00 vertices and 500,000 edges, where A* vastly improved the search time (~12 times difference).

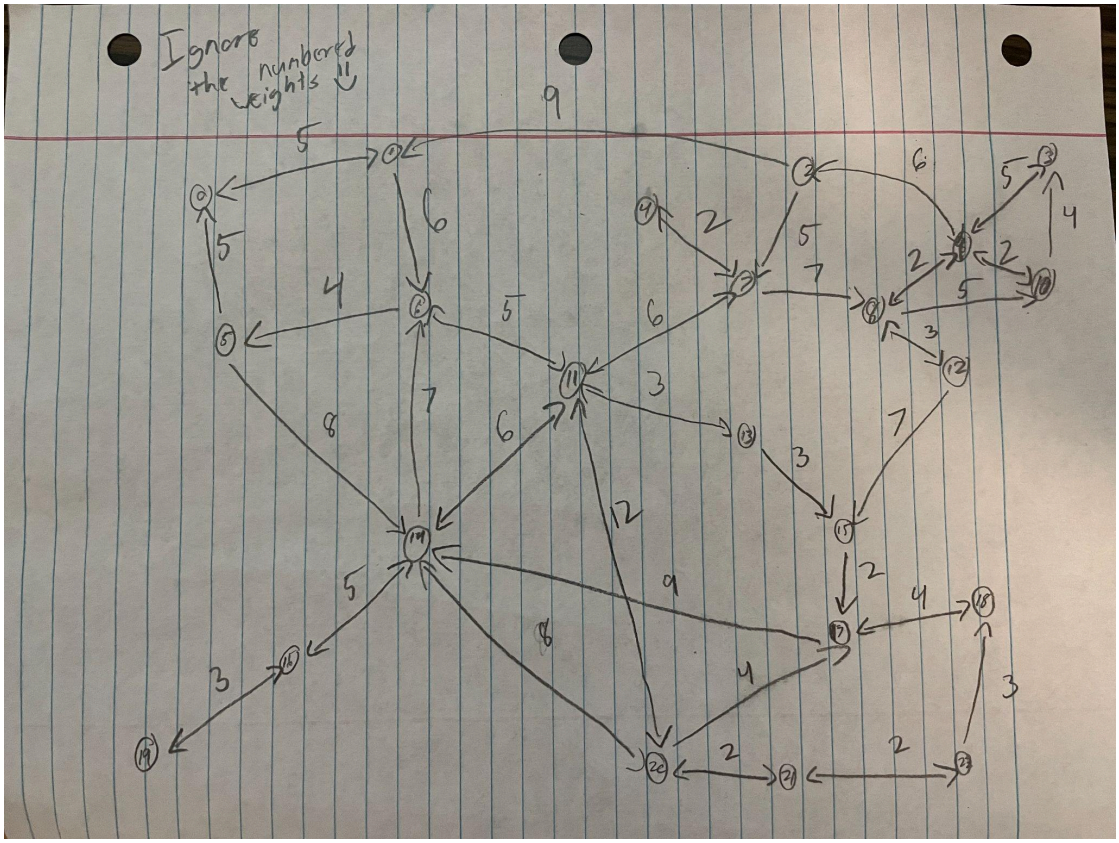
For the heuristics I chose two heuristics that deal with distance between two points in space as my vertices hold their own positions, they would make sense as estimators. The two I chose were Manhattan and Euclidean distance. When I ran the algorithms on the graphs that I had made, Manhattan outperformed Euclidean distance by ~1.23 times in the small graph while later the Euclidean outperformed Manhattan by ~1.26 times the other way around with the large graph. This seems to be the case as Manhattan could home in on the goal very quickly which worked very well with the smaller graph but then takes long in the larger graph as the overestimation eventually hurts the algorithm and thus it has to make more runs to make better calculations making the performance worse there. In the long run, I decided it would be best for me to use the Euclidean heuristic for the A* pathfinding algorithm for my game world graph as the difference in time it took for the smallest graph is something that players will not be able to notice, while at the larger end, a second difference will definitely be noticeable to the players.

When putting it all together in my game world, I decided to use the thile graph division scheme so I had to make a new Vertex and Graph class to support how I wanted these objects to interact with sfml and the game world. The Vertex now holds a RectangleShape that will dictate the area in the game world that it will be in charge of as well as a "valid" bool that will keep track of if any game objects are in the RectangleShape. Then the Graph holds the behavior for initializing the graph based off of the game world. The constructor takes in the size of the window/play area, a vector for the size of the RectangleShapes to be when representing the vertex area, and then a list of objects that exist in the game world. Then the constructor figures out how many vertices can fit in each row and column in the game world (granted that they are a factor of the game space size) and then constructs the vertices with the rectangle shapes in the correct locations. Then it goes through every vertex and determines if it is valid or not based on if their rectangle shapes overlap with any of the shapes in the game world. Then it iterates through the vertices again and connects it to each vertex around it (ensuring that it won't connect to a vertex on the left if there is no vertex on the left, same for other directions) as long as both vertices are valid. Then the graph has a function to quantize a position to a specific vertex in the node that is used for player location as well as the mouse location on click. Then the vertex has a function for localization that just outputs the position of its rectangle shape (with origin at the center). Then in my main loop, whenever a mouse click event is detected I grab the position of the mouse and quantize it to the vertex in that location as well as the boid's position and then creating a Euclidean distance heuristic with the end vertex (the mouse position vertex) and then inputting that into the A* algorithm. I then grab the output path

and call arrive and align to the next vertex, and once the boid reaches that vertex it calls arrive and align to the next vertex until the end of the path is reached (PART 4).

Appendix

Part 1: The Legend of Zelda map based graph (Ignore the weights, these were made into distances based off the the in game coordinates of these locations in game using [this website]:



Part 2/3: This shows the individual paths found with the different algorithms and their time to completion listed below them. They all found the same path and this was the case for all of the examples that I chose.

```
smoore@GamePuter:~/csc484/AndrewMooreHW3/Part1-3$ ./main
0-2112.14-1;1-1871.24-6;6-1299.28-11;11-1159.24-13;13-1235.17-15;
Small Dijkstra = 6.8e-05

0-2112.14-1;1-1871.24-6;6-1299.28-11;11-1159.24-13;13-1235.17-15;
Small AStar Euclidean = 2.1e-05

0-2112.14-1;1-1871.24-6;6-1299.28-11;11-1159.24-13;13-1235.17-15;
Small AStar Manhattan = 1.7e-05

0-14.8661-36233;36233-58.6941-88176;88176-10.7703-6563;6563-32.2025-37965;37965-57.8705-95346;95346-37.0135-
13282;13282-30-74526;74526-43.2666-2568;
Big Dijkstra = 33.2745

0-14.8661-36233;36233-58.6941-88176;88176-10.7703-6563;6563-32.2025-37965;37965-57.8705-95346;95346-37.0135-
13282;13282-30-74526;74526-43.2666-2568;
Big AStar Euclidean = 2.65394

0-14.8661-36233;36233-58.6941-88176;88176-10.7703-6563;6563-32.2025-37965;37965-57.8705-95346;95346-37.0135-
13282;13282-30-74526;74526-43.2666-2568;
Small AStar Manhattan = 3.31758
```

```
smoore@GamePuter:~/csc484/AndrewMooreHW3/Part1-3$ ./main
0-2112.14-1;1-1871.24-6;6-1299.28-11;11-1159.24-13;13-1235.17-15;
Small Dijkstra = 0.000367

0-2112.14-1;1-1871.24-6;6-1299.28-11;11-1159.24-13;13-1235.17-15;
Small AStar Euclidean = 1.9e-05

0-2112.14-1;1-1871.24-6;6-1299.28-11;11-1159.24-13;13-1235.17-15;
Small AStar Manhattan = 1.6e-05

0-14.8661-36233;36233-58.6941-88176;88176-10.7703-6563;6563-32.2025-37965;37965-57.8705-95346;95346-37.0135-
13282;13282-30-74526;74526-43.2666-2568;
Big Dijkstra = 33.1098

0-14.8661-36233;36233-58.6941-88176;88176-10.7703-6563;6563-32.2025-37965;37965-57.8705-95346;95346-37.0135-
13282;13282-30-74526;74526-43.2666-2568;
Big AStar Euclidean = 2.62717

0-14.8661-36233;36233-58.6941-88176;88176-10.7703-6563;6563-32.2025-37965;37965-57.8705-95346;95346-37.0135-
13282;13282-30-74526;74526-43.2666-2568;
Small AStar Manhattan = 3.28364
```

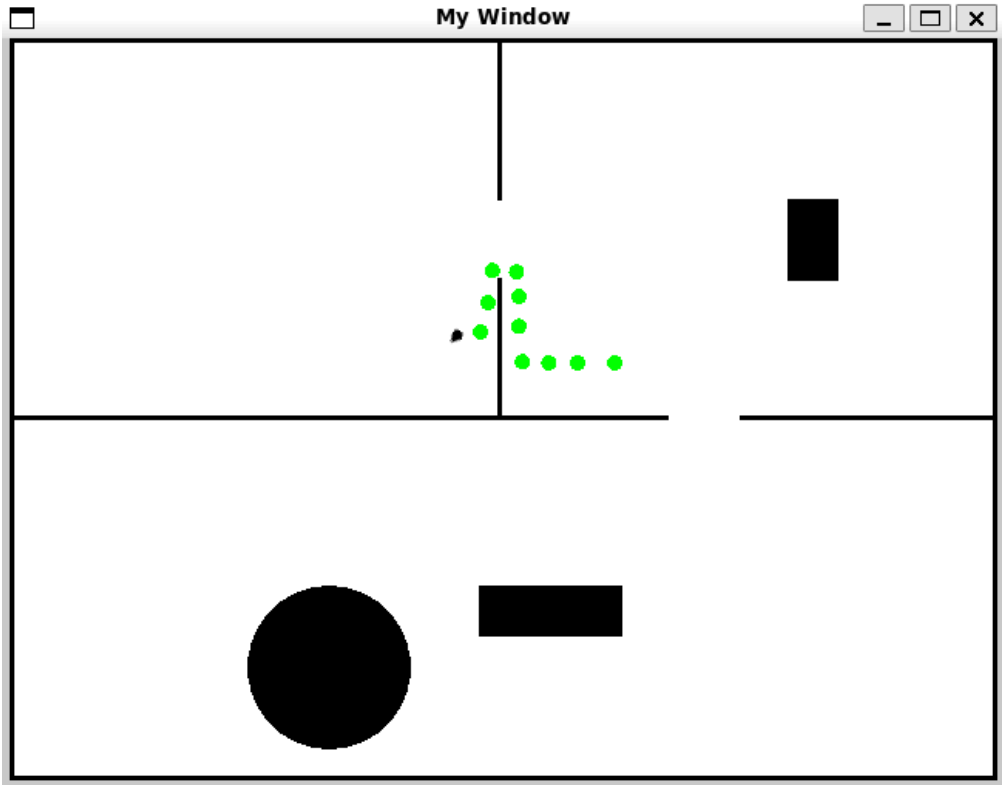
(slightly diff large graph below)

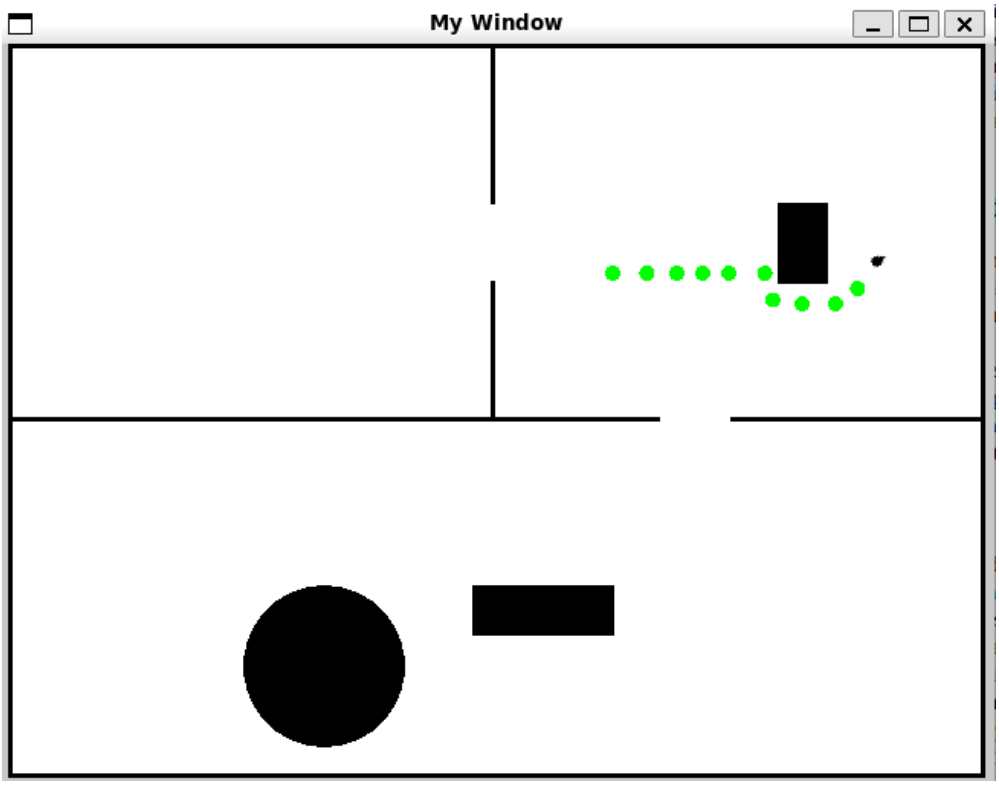
```
0-9.84886-36262;36262-13.6015-36231;36231-26.6271-99533;99533-10.4403-56038;56038-14.3178-56039;56039-50.990
2-95461;95461-36.7151-46959;46959-76.922-2568;
Big Dijkstra = 8.11694

0-9.84886-36262;36262-13.6015-36231;36231-26.6271-99533;99533-10.4403-56038;56038-14.3178-56039;56039-50.990
2-95461;95461-36.7151-46959;46959-76.922-2568;
Big AStar Euclidean = 0.304388

0-9.84886-36262;36262-13.6015-36231;36231-26.6271-99533;99533-10.4403-56038;56038-14.3178-56039;56039-50.990
2-95461;95461-36.7151-46959;46959-76.922-2568;
Small AStar Manhattan = 0.47165
```

Part 4: Showing off that





The interaction with the circle shape is interesting as I decide on validity of a vertex based off of the SFML rect objects which means that the circle collision area is actually a square