Introductory information: For all graphs where the edges are drawn, the edges will only be drawn once the graph search algorithm has been able to reach the goal. There is an additional variable the determines whether the graph is “detailed” enough to draw edges (mostly for the room graph implementation). I have made it so that clicking outside of the area of graph will still map back to a location on the map in order to prevent any null exceptions or out of index bounds.

EU & Random Graphs (Part 1): I have created a very crude representation of the European Union – a total of 25 nodes (ignoring some of the really small nations). The routes I provided it are a mixture of land and ship connections. They do not include any air travels because that would basically connect all of my vertices together (I’m already giving some of my vertices a lot of connections). The reason why I included sea routes was to try and meet the x3 edges requirement. The second one is generated via a random seed determining the location of all the vertices. The vertices all make edges with other vertices within a set range. When creating these vertices and setting the range, they have to played around with a little, or else we’ll have a situation where there are too many edges and causes my program to slow down too much. Comfortable numbers I have found for my method include the following: (1000 vertices - 1 range, 7000 vertices - 0.6 range, 20000 vertices – 0.3 range). These numbers are plugged into the following method to generate maps: generateCustomRandomMap(vertices, range). The implementation of this random graph generation follows the Naïve algorithm for random geometric graph generation. It is the primary reason why I had to play around with the vertices-range ratio.

Dijkstra's & A* Algorithm (Part 2): In order to see the implementation of the Dijkstra's algorithm that I have, you need to go to "pathfinding" and first uncomment line 28. Next you can uncomment any of the following lines within setup():

- Lines 34-36 (European Union Map)
- Lines 39-41 (Random Map)
- Lines 44-46 (Upside Down Rooms Map)

Depending on the lines uncommented, you can see Dijkstra's algorithm running on it. You will also have to comment out any other portion within the setup() that modifies the search, image, or graph variable.

In order to see the implementation of A* that I have, you follow the same steps as Dijkstra's, but instead you uncomment any of the following lines:

- Line 29 (Manhattan Heuristic)
- Line 30 (Euclidean Heuristic)
- Line 31 (Hand Written (Power) Heuristic)

While I was working on my A* algorithm, one of the most notable mistakes I made was not properly calculating and storing the cost from start for each node. This led me to an "A* algorithm" performing pretty much identically to a greedy best first search. This was something I didn't notice until I was working on the graph containing rooms (this graph follows a grid based format). I could easily see that the algorithm was not following the right path – something that was difficult to notice in my other graphs either because the length of the potential paths were either too short, or too long. The Random graph using many vertices demonstrated pretty well to me the potential costs of the different graph structures. For the following data, I used 20000 vertices with a range of 0.3. The A* algorithm visited/revisited nodes a total of 889 times, while my Dijkstra's algorithm visited a total of 10800 nodes. This shows as a graph grows bigger and the start/goal start further apart, the Dijkstra's algorithm will take far longer to reach the goal than A*.

Although I didn't use anything to get specific times for how long the algorithms took, the Dijkstra's algorithm in this case took 2-3 minutes in contrast to the roughly 15-30 seconds the A* algorithm took.

Heuristics (Part 3): I threw together several heuristics for this portion of the project. I, however, only want to showcase 3 of them in my code. The main ones are Manhattan (line 29), Euclidean (line 30), and Handwritten named Power (line 31).

Note: For the following, I am using 10000 vertices with a range of 0.5. For Manhattan, because I have the edges calculated based on distance between them, Manhattan will consistently overestimate the distance between nodes unless they are on the same x or y axis. This makes it an inadmissible heuristic that visits/revisits more nodes (405) in contrast to the admissible Euclidean heuristic (149). This suggests that the heuristic is overestimated by a decent margin (you only reap the benefits of a inadmissible heuristic performance wise if you are within a small margin of overestimating according to the textbook). The Euclidean is guaranteed to be admissible (underestimating) based on the edges I used for the graph. Both Manhattan and Euclidean visually on the large graph appear to have similar paths mainly because of the zoomed out nature of the graph. The most interesting heuristic I wrote by hand was what I named Power. The calculation for the heuristic is as follows: "Math.pow(goal.x, 2) - Math.pow(start.x, 2) + Math.pow(goal.y, 2) - Math.pow(start.y, 2))". This sends it off in a completely incorrect direction until it is roughly diagonal to the target. After which, the algorithm will branch out in two directions – one direction pursuing the goal, while the other side heads the opposite way.

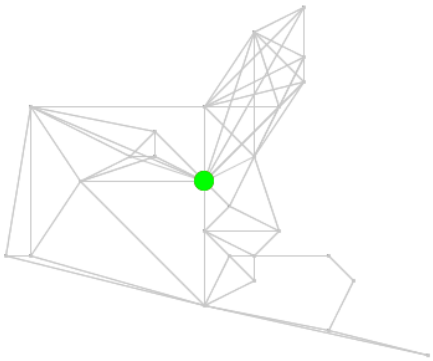
I tried playing around with having a constant or random heuristic too for A*. While they were fun to look at, I can only describe them as random in directions being taken. I can't really give a pattern they took reliably.

Path Following (Part 4): In order to get this portion running, comment out all the other parts from the other sections in setup() and uncomment line 48 (pathFollowing()) from the code. This will allow the interactable graph using A* Euclidean heuristics. If a different heuristic is to be used, go to the pathFollowing method and adjust the heuristic being sent to Asearch at the very beginning.

You can find a video of the path following in action in the folder storing the project. There is a small bug where if directly select an obstacle, the Boid can travel on top of it, and then go past it once it is sitting on top of it. Meshing together a division scheme, path following, and boid movement was surprisingly simple yet easy to mess up at the same time. I was for the most part able to move everything in without much refactoring. The major changes that I've had to do were mostly with how I fed the path information to the path following algorithm provided. I had to reverse the list and cast it to a different type of object in order to get the path following algorithm to cooperate. Increasing the scale within the path following algorithm also allowed my follow the path and avoid an implementation of collision detection. I decided to go with my kinematic arrive implementation to get the nicest appearing movement on the screen. If I took the time to make the graph more dense for this portion, I could probably eliminate the small collision issue I get occasionally.

Part 1:

European Union representation w/ paths to take



Part 3:

The interesting path Power heuristic takes: