**Graph Theorists Sketch Pad Write-up**

The first step in creating our graph theorists sketch pad was deciding what languages/libraries we were going to use. We thought up a few different ways of doing it, in python using the pygame graphics library, or a website using some javascript framework. Eventually we moved forward with writing our application in javascript. There's a couple good reasons for this. First is deployment. It's very easy to deploy a javascript application to the web. If we were to implement the application in python, we would have been limited to running the application locally without significant effort put into deployment. So the natural choice here was to continue with javascript. The second reason javascript was our language of choice for the project was because of the amount of libraries we could choose from. We ended up going with p5.js for its ease of use, but there were many other options to choose from. For deployment we used githubs website hosting service.

The whole implementation of our graph was built on top of two structures. A vertex class and edge class. The vertex class contains four main members. Position, id, and name. The id member was created to identify itself as an edge in our edgeList. The name portion made it easy to add name changing functionality as required by the instructions. The edge class contains the id’s of the endpoints of a single edge. With these two classes we now had what we needed to represent any graph in memory. The difficult part now was drawing them in a nice way.

Before drawing them there needed to be a way to actually place a vertex on the canvas, this was implemented through an onclick call back provided by p5js. When the canvas is clicked it will enter this function where we create a new vertex at the position of the mouse, and add it to the adjacencyList. What I've described here is what happens in PLACE\_VERTEX mode. Other modes cause the onclick() call to do different things.

There were five different modes in which each served different purposes, PLACE\_VERTEX, PLACE\_EDGE, SELECT, DELETE\_VERTEX, and DELETE\_EDGE. The place edge mode would update the edge list with a new edge class. Select would allow you to add edges to the selection list which was generally used to identify an object and perform some action with it. The delete vertex and delete edge modes just remove the selected vertex and all its edges from the edge and adjacency list.

For drawing our vertices it was simple. Draw a circle at the specified position of the vertex. Edges were more complicated. Because we needed support for parallel edges and loops, there had to be a way to visually display those. For parallel edges and loops the implementation involved using bezier curves to get a nice line. There also needed to be support for repositioning vertices, this just involved figuring out which vertex was under the cursor and modifying its position based on the movement of the cursor. Because of our edges implementation, the edges would follow along with the vertex.

So far the described implementation should satisfy the required functionality. Next up was the recommended features, information about numbers the graph can be easily pulled from the adjacency list. Using that we can display information such as the degree of each vertex, vertex count, edge count, and total degree. To detect the number of components a depth first search was done at each vertex that was not previously visited. The number of components corresponds to the number of times depth first search did not reach a vertex. To detect bridges another depth first search algorithm was used which traversed every edge, marking them as a bridge when needed. To detect if the graph is bipartite another depth first search algorithm was used which marked each vertex with a color. If the graph is two-colorable, then the graph is bipartite. For directed arcs we just placed an arrow based on the order of id's in the edge class.

That completes the recommended features section. A few bonus features were implemented, as well as our own bonus features. The adjacency matrix could be easily implemented using the adjacency list with a little bit of parsing. The cartesian product could take a cycle or complete graph as input and would calculate where to place vertices and edges such that it would be the cartesian product of two graphs and make it look nice.

The eigenvalues calculation was implemented using the power iteration method. This method iterates over a vector v a set number of multiplying it by the adjacency matrix and normalizing it. The more iterations performed the more accurate this method is to the real eigenvalue and eigenvector. After a loop the dot product of v and another iteration of v are returned as the eigenvalue. The eigenvector is just v.

Another bonus feature implemented was the physics system. There is a repulsive force and gravity force. To create a repulsive effect on each vertex it takes the sum of all forces on that vertex and updates the position. This can create really interesting spring like behavior in the graph. For the gravity effect there is a gravity strength variable that controls the speed at which the vertex approaches the ground. There was also an elasticity effect for elastic collisions with the ground.