Aidan Nunn – 11658886

Graph Theorist's Sketchpad App

For my final deliverable for CptS 453 I chose to create a Graph Theorist's Sketchpad app. Originally, I wanted to use C# to design this app, but I'm not very familiar with using C# for graphics so I settled on using Python with the PyGame library instead. First, I would like to describe the user-facing features, then I would like to go in to how the program itself is designed.

The program runs out of a main.py file. This file contains a running while loop that runs until the program is quit. While running, the program displays a window with six buttons. These buttons are labelled as follows: 'Place Vertex', 'Delete Vertex', 'Place Edge', 'Delete Edge', 'Degree', and 'Color'. Additionally, the program initially displays the words 'Not Bipartite' and three zeroes, which will be explained later.

Clicking 'Place Vertex' allows the user to click on the screen to place a blue vertex circle. Vertices can be placed anywhere on the screen except along the top where the buttons and other counters are located. While a button is active, it will be highlighted grey until the action is completed. In the case of 'Place Vertex', once a blue vertex has been placed, the button will become unselected. I originally allowed for users to use a button as many times as they wanted before unselecting the button, but rapid placement or deletion came with its own set of problems due to how PyGame treats object coordinates. It was too easy for users to delete the wrong object, so the current set up ensure that users must be more considerate. One additional note is that vertexes cannot be placed on top of each other and overlap.

Clicking 'Delete Vertex' allows the user to remove a vertex from the screen. If the vertex has one or more edges attached, those edges will be removed as well.

'Place Edge' allows the user to place an edge between two vertices. After 'Place Edge' is selected, the user can click the two vertices they wish to connect. A red line will be drawn between them. Clicking the same edge twice will create a loop edge that starts and finishes at the selected vertex. Parallel edges can be drawn, but due to how I have PyGame drawing lines to represent edges, they will not be represented by two arced edges. However, the degree will still be updated correctly if parallel edges are present.

‘Delete Edge’ allows the user to remove a placed edge.

Clicking ‘Degree’ will activate “Degree Mode”, where the degrees of each vertex are shown. Selecting this button again turns off “Degree Mode”. Unlike the other buttons, this one does not turn off when another is selected.

Selecting ‘Color’ will allow the user to click a vertex to cycle it through a set of colors. The available colors are blue, red, green, purple, white, orange, yellow, grey, and pink.

The app has two more user-facing features. First, it can tell the user when the graph they have made is bipartite. The words ‘Not Bipartite’ will switch to ‘Bipartite’ when the current graph is bipartite. Also, the three zeroes I mentioned earlier represent the ‘m’, ‘n’, and ‘k’ values for the number of vertices, edges, and components, respectively. These will automatically update as vertices and edges are added and removed.

In total, these are all the included features of my app, based on the features document from the related announcement:

1. Graphical display of vertices and edges
2. Input of vertices and edges
3. Deletion of vertices and edges
4. Parallel edges
5. Loops
6. Ability to color or label vertices
7. Information about numbers of vertices and edges
8. Information about degrees of vertices
9. Information about components
10. Show whether a graph is bipartite

Now, I would like to explain how this program is designed. The program consists of two Python files, main.py and class_library.py. main.py initializes and runs the program by creating the graph class object that stores all of the graph information, setting up the running while loop, setting initial button settings, and displaying graphics. This file also has the code for checking for mouse click events, which is how the user interacts with the program. class_library.py has three classes, Vertex, Edge, and Graph, which represent the three objects of the program, vertices, edges, and the overall graph.

While running, the first thing that main.py does issue PyGame to get all mouse events and the current mouse position. I've set the refresh clock to 60hz, so the program refreshes 60 times a second. Then, the program iterates through each event (which in this program is just the current mouse position or a click) and does one of three things. If the mouse button has been clicked, and its position is over a button, the clicked button will be set to active, and all other buttons will be deactivated. If a button is active and the conditions for its function are met, the function will be carried out. And finally, if the mouse is hovering over a button, that button will be highlighted grey.

After events have been checked, the code for rendering graphics is reached. Here, all of the buttons are printed to their set locations. Changing values like the 'Bipartite' message and 'n', 'm', and 'k' values are displayed. The degrees for vertices are also displayed using code in this section, but only if "Degree Mode" is active. The most important parts of main.py are at the end. Here, I have two for-loops that iterate through all vertices and edges in the backend of the program. I used PyGame to draw circles for each vertex, and lines for each edge (or an arc if the edge is a loop). And finally, I run the update_graph() method of the graph class and refresh the on-screen display.

Moving on to the class_library.py file, as stated earlier, we have three classes to talk about. The first is the Vertex class. This class is used to create class object variables that represent each of the vertices on screen. The constructor takes one input variable, 'pos', which is a Pygame rect object that represents a rectangle with the circle drawn inside of it. This object holds the vertex's on-screen coordinates and is useful for calculating where edges should be drawn. The 'neighbors' variable is a list of other vertices that the current vertex has as neighbors via edge connections. 'Degree' is used to store the current degree of the vertex, 'visited' is a Boolean used during depth first search when we are determining 'k' and if the graph is bipartite. 'Color' is also used when checking for if the graph is bipartite, as we use the graph coloring method with two colors (in this case the colors "0" or "1" for simplicity). 'Display_color' and 'colors' are used when showing the vertex's color on the screen. The methods in this class are getters and setters that relate to the class variables.

The second class is Edge. This class is even simpler than Vertex. It is initialized with the 'pos' variable, which contains a PyGame rect line object, and the start and end positions of the edge, which are center coordinates of its starting and ending vertices.

The last class is Graph. This class stores the overall data for the program's graph and contains methods for updating the graph and adding and removing vertices and edges. It is initialized with a list of vertices and a list of edges, as well as variable for storing 'n', 'm', and 'k' values and a Boolean for whether the graph is bipartite. The first method I will talk about is the add_vertex() method. This method just appends a new Vertex class object to the graph's list of vertices. Similarly, add_edge() adds an Edge class object to the array of edge objects. It also makes the two connected vertices neighbors by iterating through the vertex list with nested for loops. In the first loop, it finds the start vertex of the edge, and in the second loop it finds the end vertex. These vertexes are both added to each other's 'neighbors' lists.

remove_vertex() gets a mouse position and searches the vertex array for a match. The match is removed. This method also removes the deleted vertex from each of its neighbors' 'neighbors' lists. remove_edge() is similar, removing the edge object from the edges list and removing neighbors.

Three of the most interesting methods are update_graph(), calc_k(), and check_bipartite(). update_graph() is run at the end of the program's running while loop and updates the state of the graph. First, it updates the values of 'n', 'm', and 'k'. I calculate 'n' by returning the length of the vertices list, and I calculate 'm' by returning the length of the edges list. update_graph() also checks if the graph is bipartite, and then it removes edges that don't have two vertex connections. To remove floating edges, the graph creates a list of all vertex center coordinates, then checks each edge's start and end coordinates. If either of the edge's coordinates are not in the list of vertex centers, that edge is removed.

Calculating 'k' and checking if the graph is bipartite both use depth first search (DFS). To calculate 'k', I iterate through all the vertices in the graph's vertex list. If I find a vertex whose 'visited' value is false, I run DFS on that vertex and set all the vertices that can be reached this way to visited. This visits all the vertices in a component of the graph, so when we return to the initial for loop iterating through the graph's vertices list, if a vertex is found that has not been visited, this means a new component has been found. I increment the components count by 1 and run DFS on it. Once the initial for loop is completed, I have a count of all components. The 'visited' values of each vertex are also set to False again. This uses the depth_first_search1() method.

To check if the graph is bipartite, I used a similar DFS algorithm, except with the addition of graph coloring. This is where the depth_first_search2() method comes in to play. check_bipartite() will run DFS2 similarly to how I ran DFS when calculating ‘k’, where I started with a for loop that iterates through all of the vertices in the graph. If a vertex is not visited, I run DFS and visit and color all the connected vertices. This way, we will still be able to check if the graph is bipartite even if it isn’t connected. If a visited vertex has a neighbor with the same color as it, the graph is not bipartite. Otherwise, it is bipartite.