

## **DOCUMENTATION**

### **OVERVIEW OF PROJECT**

Upon starting the application, user will be provided with a blank canvas to work with. The canvas will include options to either generate/color a graph *or* refer to help topics.

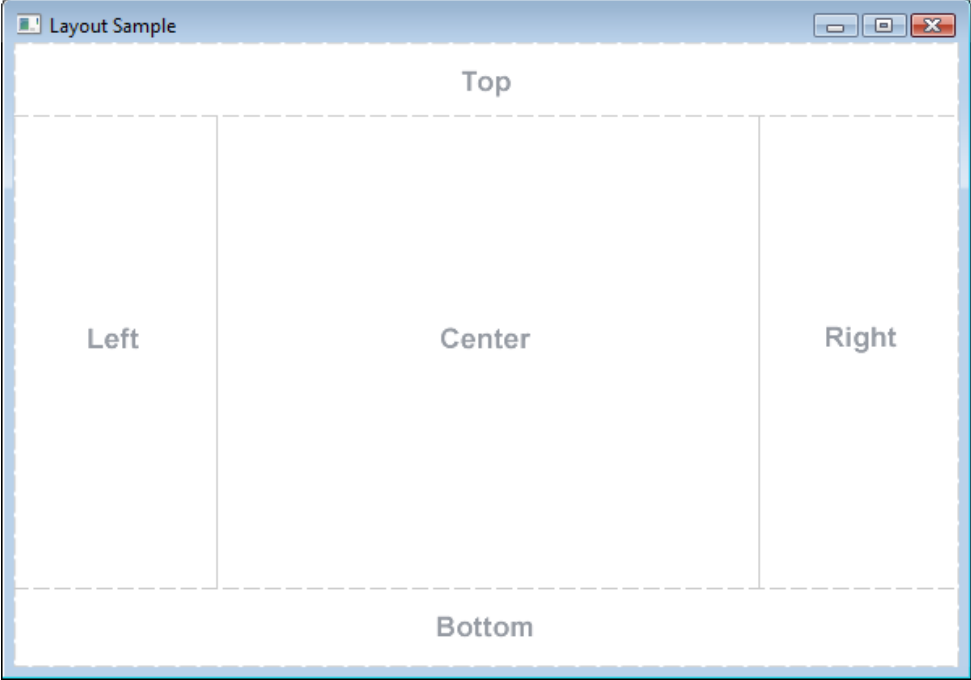
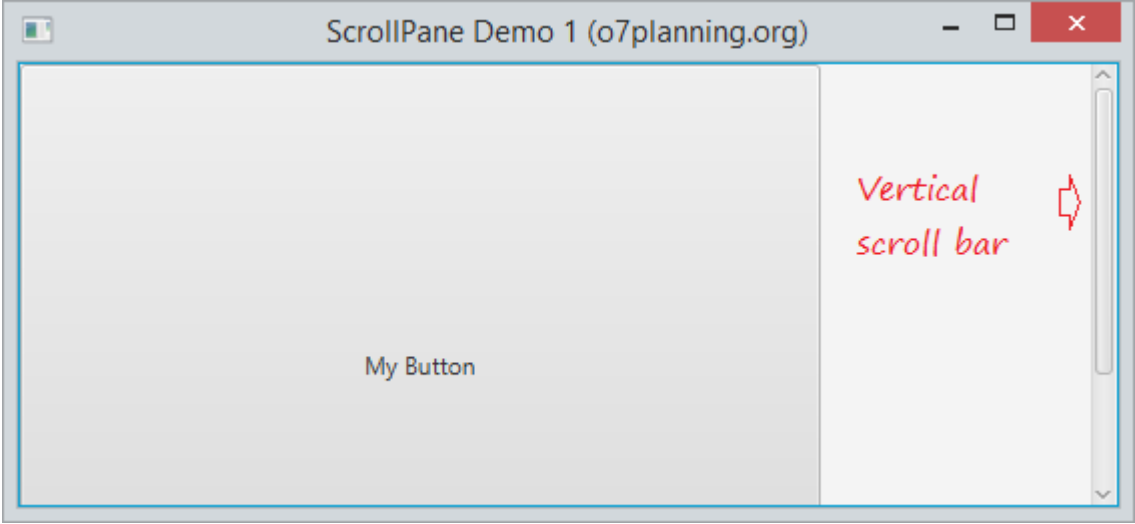
The help topics are available throughout the program's session at any time.

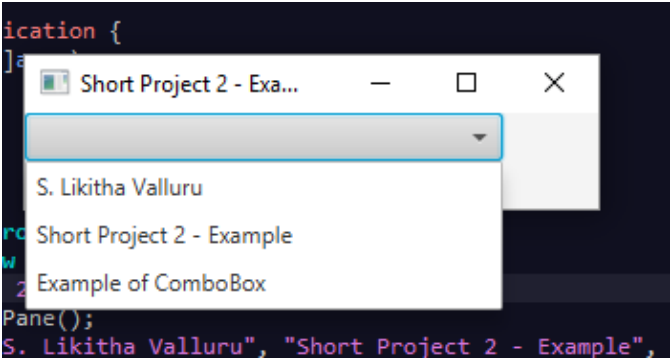
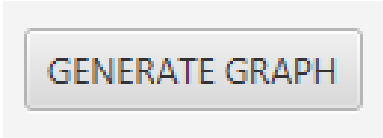

The options for generating and coloring of the graph are available as per user's actions. For example, for obvious logical reasons, a graph cannot be generated without selecting a type of graph. A graph cannot be colored if there is no graph at all. So, such features are enabled and disabled as per user's actions.

### **TERMINOLOGY**

The terminology used throughout this document is displayed in the following table to avoid any unnecessary confusion. These definitions are also included within the code also.

<b>BASIC DEFINITIONS</b>	
<b>Stage</b>	Running a JavaFX application means that a window will be opened. This window is referred to as the <i>Stage</i> in this toolkit.
<b>Scene</b>	The 'content inside of the window' is referred to as the <i>Scene</i> .
<b>BorderPane</b>	It's one of the layouts available in <i>Stage</i> .

	 <p>Image credits:</p> <p><a href="https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/BorderPane.html">https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/BorderPane.html</a></p>
<p><b>ScrollPane</b></p>	<p>Another type of pane that enables scrolling in a stage.</p>  <p>Image credits: <a href="https://o7planning.org/en/10857/javafx-scrollpane-tutorial">https://o7planning.org/en/10857/javafx-scrollpane-tutorial</a></p>
<p><b>ComboBox</b></p>	<p>A selection box that opens a drop-down menu to allow the user to choose freely from a given set of pre-defined options.</p>

	
<b>Button</b>	<p>A button that will perform an action upon mouse click (or any other operation).</p> 
<b>HBox</b>	<p>A type of layout that stores objects in a straight, horizontal manner.</p> 
<b>VBox</b>	<p>A type of layout that stores objects in a vertical manner.</p> <p>(The two lines of text here are stored in a VBox before printing them inside the stage.)</p>
<b>MouseEvent <i>e</i> (or <i>event</i>)</b>	<p>The mouse events are represented by <i>e</i> or <i>event</i>. Sometimes, the program will read a mouse event <i>inside</i> another mouse event handler.</p> <p>In this case, a new identifier <i>cannot</i> be used until and unless the mouse event variable is declared static. Since mouse events are extremely sensitive and important to the program, these variables are not declared as static and the default identifier <i>e</i> or <i>event</i> is used. (If declared static for every mouse event, there will be about 10-15 new variables that will be difficult to keep track of.)</p>

## UNDERSTANDING IDENTIFIERS

The following is a list of the *most important* identifiers used in this program. (Importance is measured by frequency of usage.)

Note: Some may have been omitted because they were not crucial enough for the data structures (such as identifiers used for setting flags, loop conditions, etc.). There were other such identifiers as well that were used for the *graphics* portion of the program. Some of these are also omitted.

But those that may help in aiding the understanding of data structures are included and documented carefully. Each important identifier is (mostly) used for one data structure. So, it's documented accordingly.

**Important Note: ALL identifiers are defined in camel-notation to increase ease of understanding.**

### IDENTIFIERS FOR GRAPHICS [ONLY FOR THE MAIN SCREEN]

SCREEN VISUALS (JavaFX)	
<b><i>mainStage</i></b> ( <i>Stage</i> )	As explained above, a stage is the window used to display the running application. In this case, the stage is denoted as <i>mainStage</i> .
<b><i>mainScene</i></b> ( <i>Scene</i> )	The content of the window is referred to as the <i>mainScene</i> .
<b><i>border</i></b> ( <i>BorderPane</i> )	The <i>BorderPane</i> was chosen to allow more room for the user to work with. The only areas that the user won't be able to use are the top and bottom portions because they're being used to hold buttons, combo boxes, etc.
<b><i>graphBox</i></b> ( <i>ComboBox&lt;String&gt;</i> )	<p>The <i>graphBox</i> was initially of type <i>ObservableList</i> because it was the most popular option used in Oracle documents. So, therefore, I used it as well. But there were unforeseen complications with <i>ObservableList</i> because it prevented even the smallest of operations, including selecting a certain element of an index of the <i>ComboBox</i>. Therefore, <i>ObservableList</i> was changed to type <i>String</i>.</p> <p>There are <i>nine</i> types of graphs that the user can select from.</p>
<b><i>colorBox</i></b> ( <i>ComboBox&lt;String&gt;</i> )	<p>The <i>colorBox</i> was initially of the type <i>ObservableList</i> as well, but it was changed to <i>String</i> after going into coding.</p> <p>The <i>colorBox</i> is like <i>graphBox</i>, except that it shows options for different colors.</p> <p>There are <i>eight</i> available colors for the user to choose from.</p>
<b><i>operations</i></b> ( <i>ComboBox&lt;String&gt;</i> )	The <i>operations</i> box allows user to select from a provided list of possible operations on the graph.

	<i>BEFORE</i> using a <i>ComboBox</i> , each operation was stored as a separate button, but that did not prove to be useful. Each button is going to be clicked multiple times and to invoke a separate handler for each button took a lot of heap memory and the program was considerably slow. So, buttons were abandoned.
<b><i>generateGraph</i></b> ( <i>Button</i> )	A button that generates a graph upon mouse click. <i>However</i> , an error dialog box <i>will</i> show if the user doesn't select any type of graph. (In this case, the dialog box is simply <i>another</i> stage.)
<b><i>clearGraphColors</i></b> ( <i>Button</i> )	Clear the graph of all colors and set it to <b><i>Color.BURLYWOOD</i></b> . (Default color for each node)
<b><i>clearScreen</i></b> ( <i>Button</i> )	Clear the entire screen and provide user with a fresh canvas.
<b><i>quitApplication</i></b> ( <i>Button</i> )	Upon clicking this button, it's assumed that the user no longer wants to use the application and therefore, the application session is closed, and all data is erased and freed from used memory locations (this is automatically done with Java's garbage collector).
<b><i>errorStage</i></b> ( <i>Stage</i> )	This is like a dialog box that pops up when the user doesn't select a type of graph but clicks on <i>Generate</i> .
<b><i>typeOfGraph</i></b> ( <i>String</i> )	When the user selects a graph type from <i>graphBox</i> , that graph type is stored in this variable.
<b><i>selectedColor</i></b> ( <i>Color</i> )	When the user selects a color from <i>colorBox</i> , that color is stored in this variable.
<b><i>operation</i></b> ( <i>String</i> )	When the user selects a color from <i>operations</i> , that operation is stored in this variable.
<b><i>circle</i></b> ( <i>Circle</i> )	This is just shown (here in the documentation) as an arbitrary variable. There are many other identifiers of the type <i>Circle</i> and these variables are used to denote the vertices. This variable is defined in every graph class.
<b><i>line</i></b> ( <i>Line</i> )	Like <b><i>circle</i></b> , the <b><i>line</i></b> variable is also arbitrarily shown in this document. There are many other identifiers of the type <i>Line</i> and these variables are used to denote the edges.

<b><i>vertices</i></b> <i>(ArrayList&lt;Circle&gt;)</i>	<p>Keeps a track of the list of the vertices drawn in the scene.</p> <p>This ArrayList will manage every graph. Since only one graph will be dealt with at a time, only one object of ArrayList is initialized. If a new graph is chosen or if screen is cleared, then this list will be reset.</p>
<b><i>edges</i></b> <i>(ArrayList&lt;Line&gt;)</i>	<p>Keeps a track of the list of the edges drawn in the scene.</p> <p>This ArrayList will manage every graph. Since only one graph will be dealt with at a time, only one object of ArrayList is initialized. If a new graph is chosen or if screen is cleared, then this list will be reset.</p>
<b><i>listOfColors</i></b> <i>(ArrayList&lt;Color&gt;)</i>	<p>Keeps a track of the list of the colors during graph coloring.</p> <p>This ArrayList will add the colors used in the graph so far. If a new graph is chosen, screen is cleared, <i>or</i> colors are reset, then this list will be reset as well.</p>

## IDENTIFIERS FOR GRAPHS

GRAPH GENERATION: <b><i>TREE</i></b>	
<b><i>drawTree</i></b> ( <i>Tree</i> )	An object of the class <i>Tree</i> . It is used whenever the user selects the <i>Tree</i> data structure.
<b><i>flag</i></b> ( <i>Boolean</i> )	<p>It's a little complicated to explain in <i>this</i> table, so this variable's importance will be explained in further detail near the graph's code.</p> <p>But briefly explaining, this variable determines if a new vertex is to be added. For example, if the user wants to add an edge between two existing vertices, then this flag will be set to <i>false</i>. If the flag is set to <i>true</i>, then it means that the user wants to add an edge between an existing and a new vertex.</p>
<b><i>planar</i></b> ( <i>Boolean</i> )	Keeps status of whether or not the newly drawn edge is planar to existing the graph

GRAPH GENERATION: <b><i>2-TREE</i></b>
--

<b><i>drawTwoTree</i></b> ( <i>TwoTree</i> )	An object of the class <i>TwoTree</i> . It is used whenever the user selects the <i>2-tree</i> data structure.
<b><i>selectCounter</i></b> ( <i>int</i> )	Briefly explaining (it will be explained further with the code itself), in order to add two edges in a 2-tree, two adjacent vertices must be selected. Only then can a new vertex and edges can be added. This counter determines whether or not a user selected two vertices.
<b><i>selectFlag1</i></b> ( <i>Boolean</i> )  <b><i>selectFlag2</i></b> ( <i>Boolean</i> )	These two flags correspond to the two areas that the user clicked. The event handler determines whether the clicked areas belong <i>inside</i> of existing vertices.  If either of them is <i>false</i> , then it means that two vertices were not selected. In that case, the user will be prompted to select vertices properly.
<b><i>selectFlag</i></b> ( <i>Boolean</i> )	If both flags are true, then only can the user insert a new vertex.
<b><i>adjacencyFlag</i></b> ( <i>Boolean</i> )	If <i>selectFlag</i> is <i>true</i> , then a method is called to check if the vertices are adjacent or not.
<b><i>selectEventVertex1</i></b> ( <i>Circle</i> )  <b><i>selectEventVertex2</i></b> ( <i>Circle</i> )	These variables correspond to the two vertices that the user clicked on. Order of clicking does not matter, so either variable is for either of the vertices.

<b>GRAPH GENERATION: <i>MAXIMAL OUTER PLANAR GRAPH (abbreviated as MOP)</i></b>	
<b><i>drawMOP</i></b> ( <i>MOP</i> )	An object of the class <i>MOP</i> . It is used whenever the user selects the <i>MOP</i> data structure.

<b><i>selectCounter</i></b> (int)  <b><i>selectFlag1</i></b> (Boolean)  <b><i>selectFlag2</i></b> (Boolean)  <b><i>selectFlag</i></b> (Boolean)  <b><i>adjacencyFlag</i></b> (boolean)  <b><i>selectEventVertex1</i></b> (Circle)  <b><i>selectEventVertex2</i></b> (Circle)	<p>The same variables are used both for 2-trees and MOPs. The only difference for MOPs is that this class also checks for outerplanarity as well and this will be further discussed in the data structures portion.</p>
---	---

GRAPH GENERATION: <b><i>K4 PLANAR GRAPH</i></b>	
<b><i>drawK4Planar</i></b> ( <i>K4Planar</i> )	<p>An object of the class <i>K4Planar</i>. It is used whenever the user selects the <i>K4 planar graph (adding a vertex &amp; three edges)</i> data structure.</p>
<b><i>triangles</i></b> ( <i>ArrayList&lt;Polygon&gt;</i> )	<p>This list is not displayed to the user at all. This is <i>solely</i> for the programmer's perspective.</p> <p>In this case, the <i>Polygon</i> parameter refers to a triangle. In a K4 planar graph, a user clicks in the middle of the graph. Based on his mouse click, a vertex <i>and</i> three new edges are added.</p> <p>Broadly speaking, the K4 graph is made up of triangles. The <b><i>triangles</i></b> list stores the list of triangles that result from each mouse click.</p> <p>Each time the user clicks, his mouse click is run through this list. Whichever triangle contains the mouse click, that triangle is split into three more triangles and nodes/edges are added.</p> <p>It's worth to note that <i>ALL</i> triangles are added into this list. A point may be debated on here: <i>more than one triangle will contain the mouse click</i>.</p> <p>To solve this issue, the loop will run until the <i>end</i> of the list. As the list progresses, the size of triangles grows smaller. Whichever recent</p>



	triangle contains this point, that triangle is split (it's not deleted from the list; three more are added).
--	--

GRAPH GENERATION: <i>PLANAR CUBIC GRAPHS</i>	
<b><i>drawK4PlanarCubic</i></b> ( <i>K4PlanarCubic</i> )	An object of the class <i>K4PlanarCubic</i> . It is used whenever the user selects the <i>K4 planar cubic graph</i> data structure.
<b><i>drawK6PlanarCubic</i></b> ( <i>K6PlanarCubic</i> )	An object of the class <i>K6PlanarCubic</i> . It is used whenever the user selects the <i>K6 planar cubic graph</i> data structure.
<b><i>drawK8PlanarCubic</i></b> ( <i>K8PlanarCubic</i> )	An object of the class <i>K8PlanarCubic</i> . It is used whenever the user selects the <i>K8 planar graph</i> data structure. (This is used for all three types of K8 planar cubic graphs).
<b><i>counter</i></b> ( <i>int</i> )	This counter is initially set to zero. This keeps track of how many times the user selects edges. A selection of an edge will increment the counter. If the user clicks twice, then this counter is reset.
<b><i>xcoordinate1</i></b> ( <i>double</i> )  <b><i>ycoordinate1</i></b> ( <i>double</i> )	The coordinates for the first mouse click are stored in these two variables.
<b><i>xcoordinate2</i></b> ( <i>double</i> )  <b><i>ycoordinate2</i></b> ( <i>double</i> )	The coordinates for the second mouse click are stored in these two variables.
<b><i>drawIndices[]</i></b> ( <i>double</i> )	This stores indices of the new edges and vertex that have to be added into the graph ( <i>provided the user's mouse clicks follow the rules of graphs</i> ).

## IMPORTANT METHODS AND CLASSES

This project is composed of a few classes and methods. Each class is meant to do one certain aspect of the program. Each aspect (for the most part) is recorded in a video.

The following table just gives a brief description of each class. The methods that are associated with data structures are discussed extensively in the next section, supported by the entire code of that method.

<b><i>MainProgram</i></b> (Class)	<p>Class containing the <i>main ()</i> method, <i>start ()</i> method (for starting the JavaFX application), etc.</p> <p>The program extends <i>javafx.application.Application</i> (to initialize the application) and implements <i>EventHandler&lt;ActionEvent&gt;</i> (to handle mouse events)</p>
<b><i>main ()</i></b> (Method of no return type)	<p>The <i>main ()</i> method in JavaFX is not used at all <i>except</i> for initializing the JavaFX application using:</p> <p><i>launch(args);</i></p> <p><i>//args=arguments passed through the main () method</i></p>
<b><i>start ()</i></b> (Method of no return type)	<p>The method is responsible for displaying the stage along with the scenes added in the stage.</p>
<b><i>addGraphColoring ()</i></b> (Method of return type <b><i>HBox</i></b> )	<p>The <i>border</i> (BorderPane) is designed using two parts:</p> <ol style="list-style-type: none"> <li>Add options for implementing and coloring graphs</li> <li>Add options to refer to help topics.</li> </ol> <p>So, <i>addGraphColoring ()</i> adds buttons and combo boxes in the top layer of the <i>BorderPane</i>. These options are stored within an <i>HBox</i> and this is returned to use it inside of the stage.</p>
<b><i>addHelpTopics ()</i></b> (Method of return type <b><i>HBox</i></b> )	<p>This is the second part of <i>border</i>. (Bottom of the stage)</p> <p>This adds buttons and combo boxes in the bottom layer of the <i>BorderPane</i>. These options are stored within an <i>HBox</i> and this is returned to use it inside of the stage.</p>
<b><i>graphColoring ()</i></b> (Method of void return type)	<p>A common method for all nine graphs. It checks whether the user is coloring the graph properly.</p>
<b><i>checkForAdjacency ()</i></b> (Method of return type <b><i>Boolean</i></b> )	<p>A common method for 2-trees and MOPs. It checks whether the user has chosen two adjacent vertices or not.</p>

<b><i>addVerticesAndEdges</i></b> () (Method of return type <b><i>double[]</i></b> )	A common method for all <i>five</i> planar cubic graphs <i>AND</i> the <i>tree</i> graph type. As per the method's title, the function of this method is to add new edges and vertices as per the user's mouse events.
<b><i>Tree</i></b> (Class)	The tree data structure is implemented within this class.
<b><i>addVerticesAndEdges</i></b> () (Method of return type <b><i>double[]</i></b> )	(Not to be confused with the previous method.)  This method is solely for the tree. This method calls another method which checks planarity separately.
<b><i>planarity</i></b> () (Method of return type <b><i>Boolean</i></b> )	Determines whether the newly drawn edge is planar or not.
<b><i>adjust</i></b> () (Method of return type <b><i>CubicCurve</i></b> )	In case where graph is not planar, then this method will adjust that edge and display a curved edge.
<b><i>TwoTree</i></b> (Class)	The 2-tree data structure is implemented within this class.
<b><i>insertVertexAndDrawEdges</i></b> (Method of return type <b><i>double[]</i></b> )	This method exists in BOTH 2-trees and MOPs ( <i>in both TwoTree and MOP classes</i> ) and as per the title of method, it's used to add a vertex and draw edges into the graph.
<b><i>MOP</i></b> (Class)	The maximal outerplanar graph (MOP) data structure is implemented within this class.
<b><i>checkForOuterPlanarity</i></b> (Method of return type <b><i>Boolean</i></b> )	Exclusive for MOP graphs and checks for outerplanarity of the graph's structure.
<b><i>K4Planar</i></b> (Class)	The K4 graph (adding three edges and one vertex) data structure is implemented within this class.
<b><i>splitIntoTriangles</i></b> () (Method)	A method that is in <i>K4Planar</i> class which keeps track of divided areas (in the form of polygons) and adds a vertex along with three edges.
<b><i>K4PlanarCubic</i></b> (Class)	The K4 planar cubic graph data structure is implemented within this class.

<b><i>K6PlanarCubic</i></b> (Class)	The K6 planar cubic graph data structure is implemented within this class.
<b><i>K8PlanarCubic</i></b> (Class)	The K8 planar cubic graph data structure is implemented within this class.
<b><i>initialization</i></b> () (Method available in every class that is related to a graph)	<p>This method is implemented in <i>every</i> graph class. Whenever the user selects a graph and clicks on generate for the first time, an initial graph is shown. This initial graph is shown in the stage through this method.</p> <p>(Graph class of type K8 has three different graphs available. Therefore, the methods in <i>K8PlanarCubic</i> are labelled as <i>initializationTypeI</i> (), <i>initializationTypeII</i> (), and <i>initializationTypeIII</i> ().)</p>

## INTERNAL DATA STRUCTURES

### Storage

The most crucial part of this project is to be able to sync the vertices *and* edges without involving the user in any way. As per the rules of this project, in terms of space, adjacency/incidence lists/matrices were not used.

The space complexity in terms of storage was  $O(V+E)$ . An explicit matrix or linked list was not used. Instead, as mentioned above in list of identifiers, *vertices* (*ArrayList*) and *edges* (*ArrayList*) were used to store data. Each existing and new vertex is stored in *vertices*. Each new edge, along with old edges, are stored in *edges*. Mouse events are used to handle/detect vertices and edges.

Graphically speaking, each edge starts at a vertex's center and ends at another vertex's center. Therefore, it was easy to keep track of the graph without initiating an  $n^2$  complexity.

So, the space complexity was the sum of (total number of vertices  $V$ ) and (total number of edges  $E$ ) =  $O(V+E)$ .

### Data Structures

***Note: To preserve the integrity of formatting and indentation available in Eclipse IDE, screenshots were taken instead of pasting the entire code.***

For the entire project, no sophisticated data structures were needed. I've used *ArrayLists*, which are very similar to standard arrays, but provide better functionality. Other than this, no other data structures were used.

As per the rules of this project, adjacency matrices, adjacency lists, and incidence matrices were rejected. Therefore, to be on the safer side, no matrices and no linked lists are used. Since graphs are non-linear, abstract-type data structures and since lists and matrices were out of question, the next best option that I considered was using arrays in form of *ArrayLists* in Java.

Initializing the graph types didn't take any sophisticated data structure. Vertices were added in the form of circles and edges were added in the form of lines. (Both of these shapes are available in Java.) The initialization was easy because it was statically determined. All the shapes present in initialization were respectively added into *vertices* and *edges* (both are of the data type *ArrayList*).

An example of the initialization for a K4 planar graph is shown below:

```
static class K4Planar {  
  
    public void initialization() {  
  
        // CIRCLE 1  
        System.out.println("K4-CIRCLE1");  
        Circle circle1 = new Circle();  
        circle1.setCenterX(700);  
        circle1.setCenterY(80);  
        circle1.setRadius(20);  
        circle1.setFill(Color.BURLYWOOD);  
  
        // CIRCLE 2  
        System.out.println("K4-CIRCLE2");  
        Circle circle2 = new Circle();  
        circle2.setCenterX(300);  
        circle2.setCenterY(600);  
        circle2.setRadius(20);  
        circle2.setFill(Color.BURLYWOOD);  
  
        // CIRCLE 3  
        System.out.println("K4-CIRCLE3");  
        Circle circle3 = new Circle();  
        circle3.setCenterX(1100);  
        circle3.setCenterY(600);  
        circle3.setRadius(20);  
        circle3.setFill(Color.BURLYWOOD);  
  
        // CIRCLE 4  
        System.out.println("K4-CIRCLE4");  
        Circle circle4 = new Circle();  
        circle4.setCenterX(700);  
        circle4.setCenterY(380);  
        circle4.setRadius(20);  
        circle4.setFill(Color.BURLYWOOD);  
    }  
}
```

```

// LINES
Line line1 = new Line(700, 80, 300, 600);
Line line2 = new Line(300, 600, 1100, 600);
Line line3 = new Line(1100, 600, 700, 80);
Line line4 = new Line(700, 80, 700, 380);
Line line5 = new Line(300, 600, 700, 380);
Line line6 = new Line(1100, 600, 700, 380);

// Keep track of divided triangles
Polygon polygon1 = new Polygon();
polygon1.getPoints().addAll(new Double[] { 700.0, 80.0, 700.0, 380.0, 300.0, 600.0 });
Polygon polygon2 = new Polygon();
polygon2.getPoints().addAll(new Double[] { 300.0, 600.0, 700.0, 380.0, 1100.0, 600.0 });
Polygon polygon3 = new Polygon();
polygon3.getPoints().addAll(new Double[] { 700.0, 80.0, 700.0, 380.0, 1100.0, 600.0 });

vertices.add(circle1);
vertices.add(circle2);
vertices.add(circle3);
vertices.add(circle4);

edges.add(line1);
edges.add(line2);
edges.add(line3);
edges.add(line4);
edges.add(line5);
edges.add(line6);

triangles.add(polygon1);
triangles.add(polygon2);
triangles.add(polygon3);
}

public double[] splitIntoTriangles(double xcoord, double ycoord) {
}

```

As seen above, the initialization is *VERY* basic and took almost zero efforts to make except for placing the edges and vertices in appropriate coordinates.

There is one point that I would like to address: the extensive use of lines. One may ask why so many lines? Why not use a triangle or quadrilateral or polygon? The reason for that is because it's complex to analyze. And there would be no proper structure to the graph's form of storage at all. Since graphs are completely unpredictable and dynamically drawn, one can't expect a graph to be divided in triangles only or rectangles only. Therefore, lines were used to edges. Only the initialization of each graph type is a bit redundant.

So, each graph's initialization is like the code posted above. So, code for the initialization of the graphs is not discussed in the further sections of the document.

## TREE

The tree's initial structure is a simple node. Only one vertex will be displayed to the user.

First, the event handler will be invoked. The code is displayed below:

```

case "draw":
    switch (typeOfGraph)
    {
    case "tree":
        mainScene.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                System.out.println("INSIDE DRAW TREE EDGE");
                double positions[] = drawTree.addVerticesAndEdges(event.getX(), event.getY());
                System.out.println("AFTER INDICE");
                for (int i = (int) positions[0]; i < vertices.size(); i++)
                    border.getChildren().add(vertices.get(i));
                for (int i = (int) positions[1]; i < edges.size(); i++)
                    border.getChildren().add(edges.get(i));
                System.out.println("Drawn");
                mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
            }
        });
        break;
    }
}

```

From the event handler, the method to draw new edges is invoked. After returning to this handler again, the new edges/vertex are displayed using a *for* loop. (Only the new ones are displayed. The method returns the starting point of the new additions to the ArrayLists *vertices* and *edges*.)

The methods in the *Tree* class are as follows:

```

public double[] addVerticesAndEdges(double xcoord, double ycoord) {
    double indices[] = { vertices.size(), edges.size() };

    circleTree.setRadius(20);
    circleTree.setFill(Color.BURLYWOOD);
    circleTree.setVisible(true);

    // Determine whether or not the user selected a starting vertex initially
    for (index = 0; index < vertices.size(); index++) {
        /*
         * If clicked point is within a vertex, break from the loop and store that
         * vertex.
         */
        if (vertices.get(index).contains(xcoord, ycoord)) {
            circleTree = vertices.get(index);
            break;
        }
    }

    if (true) {
        line.setStartX(circleTree.getCenterX());
        line.setStartY(circleTree.getCenterY());

        // Observe the user's movements as he moves the mouse around.
        circleTree.setOnMouseDragged(e -> {
            line.setEndX(e.getX());
            line.setEndY(e.getY());
            line.setVisible(true);
            // Check for planarity while dragging.
            if (planarity(e.getX(), e.getY()))
                ;
        });

        // Observe the user's movements as he releases the mouse.
        circleTree.setOnMouseReleased(e -> {
            System.out.println("Mouse released");
        });
    }

    return indices;
}

```

```

System.out.println("Mouse released");

boolean ifElse = true;

for (int i = 0; i < vertices.size(); i++) {
    /*
     * If the user's starting and ending vertices are the same, then loop is broken.
     */
    if (circleTree.contains(e.getX(), e.getY())) {
        System.out.println("BROKEN");
        ifElse = false; // user can return back to the event handler.
    }
    if (vertices.get(i).contains(e.getX(), e.getY())) {
        /* User decided to draw an edge between two existing vertices. */
        flag = true;
        traverse = i;
        System.out.println("FLAG=TRUE");
        break;
    }
}
/*
 * If starting & ending vertices are same, then user will be redirected back to
 * event handler.
 */
if (ifElse) {
    /* User drew an edge between an existing vertex to another existing vertex. */
    if (flag) {
        // line.setEndX(vertices.get(traverse).getCenterX());
        // line.setEndY(vertices.get(traverse).getCenterY());

        // Check for planarity
        if (!planarity(e.getX(), e.getY())) {
            curve = adjust(e.getX(), e.getY(), line.getStartX(), line.getStartY());
            border.getChildren().add(curve);
            edges.add(new Line(line.getStartX(), line.getStartY(), line.getEndX(), line.getEndY()));
            return;
        }
    }

    Line draw = new Line(line.getStartX(), line.getStartY(),

```



graphviz.java

```
        edges.add(new Line(line.getStartX(), line.getStartY(), line.getEndX(), line.getEndY()));
        return;
    }

    Line draw = new Line(line.getStartX(), line.getStartY(),
        vertices.get(traverse).getCenterX(), vertices.get(traverse).getCenterY());
    edges.add(draw);
    flag = false;
    border.getChildren().add(draw);
}

/* User drew an edge from an existing vertex to a new vertex. */
else {
    System.out.println("INSIDE ELSE!");

    // Initialize new vertex.
    Circle circle2 = new Circle();
    circle2.setCenterX(e.getX());
    circle2.setCenterY(e.getY());
    circle2.setVisible(true);
    circle2.setRadius(20);
    circle2.setFill(Color.BURLYWOOD);

    if (!planarity(e.getX(), e.getY())) {
        curve = adjust(e.getX(), e.getY(), line.getStartX(), line.getStartY());
        border.getChildren().add(curve);
        vertices.add(circle2);
        edges.add(new Line(line.getStartX(), line.getStartY(), line.getEndX(), line.getEndY()));
    }

    /* If there is no issue of planarity, then draw a new edge. */
    Line draw = new Line(line.getStartX(), line.getStartY(), e.getX(), e.getY());
    vertices.add(circle2);
    edges.add(draw);
    border.getChildren().add(circle2);
    border.getChildren().add(draw);
}
}
```

```

        vertices.add(circle2);
        edges.add(draw);
        border.getChildren().add(circle2);
        border.getChildren().add(draw);
    }
    });
}

return indices;
}

public static boolean planarity(double xcoord, double ycoord) {
    /*
     * If any of the points of the line are a part of the existing edges, then the
     * method will return false
     */
    for (int i = 0; i < edges.size(); i++) {
        if (edges.get(i).contains(xcoord, ycoord))
            return false;
    }
    return true;
}

public static CubicCurve adjust(double lineX, double lineY, double xcoord, double ycoord) {
    CubicCurve cubic = new CubicCurve();
    cubic.setStartX(lineX);
    cubic.setStartY(lineY);
    cubic.setControlX1(xcoord);
    cubic.setControlY1(ycoord);
    cubic.setControlX2(lineX);
    cubic.setControlY2(lineY);
    cubic.setEndX(xcoord);
    cubic.setEndY(ycoord);
    return cubic;
}
}

```

All of the objects in this project are either corresponding to a vertex or to an edge. These vertices and edges are in *ArrayLists*. These are the only data structures that are prominently used. Whenever the user makes a new addition, the data is added into the lists.

The colors for the vertices are not stored in a separate list. A vertex's assigned color is stored in the same list. Each vertex is represented by a small circle and this circle has *setFill()* method that Java provides by default.

So, no other data structures were necessary. Rather than data structures, the way that data was displayed was the biggest challenge. Making sure that unnecessary data isn't shown and verifying that needed data is stored safely was one of the significant challenges in this program, *especially* during the designing of this graph.

## 2-TREE

The only difference between MOPs and 2-trees is that the latter graph type does not check for planarity. Therefore, this graph was easier to implement than the others. To add a new edge, the selected vertices must be adjacent.

*First and foremost*, the user must select two vertices. Only after it's *confirmed* that the user selected vertices and not random places, the above method is called.

The event handling for SELECTION OF VERTICES is same for MOPs and 2-trees. Event handling for INSERTION OF VERTEX is quite different, especially since MOPs check for outerplanarity.

The following shows selection of vertices:

```
switch (typeOfGraph) {
case "MOP":
case "2tree":
    mainScene.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent mouseEvent) {
            /*
             * If the user is selecting his FIRST vertex, then DON'T CALL THE ADJACENCY
             * METHOD YET! Use selectCounter to keep track of user's activities. If
             * selectCounter==0, then wait. Else, call method and make selectCounter=0;
             */
            if (selectCounter == 0) {
                selectCounter++;
                selectEvent1X = mouseEvent.getX(); // selectEvent1 = coordinates of first event
                selectEvent1Y = mouseEvent.getY();

                for (int i = 0; i < vertices.size(); i++) {
                    if (vertices.get(i).contains(selectEvent1X, selectEvent1Y)) {
                        selectEventVertex1 = vertices.get(i);
                        // vertices.get(i).setFill(Color.CRIMSON);
                    }
                }
                selectFlag1 = draw2Tree.selectVertex(mouseEvent.getX(), mouseEvent.getY());
            }
            // User selected his second coordinate
            else if (selectCounter == 1) {
                selectCounter++;
                selectEvent2X = mouseEvent.getX(); // selectEvent2 = coordinates of second event
                selectEvent2Y = mouseEvent.getY();
                selectFlag2 = draw2Tree.selectVertex(mouseEvent.getX(), mouseEvent.getY());
                // Check whether this event is INSIDE of a vertex.
                for (int i = 0; i < vertices.size(); i++) {
                    if (vertices.get(i).contains(selectEvent2X, selectEvent2Y)) {
                        selectEventVertex2 = vertices.get(i);
                        // vertices.get(i).setFill(Color.CRIMSON);
                    }
                }
            }
            selectCounter = 0;
        }
    });
}
```

```
// BOTH EVENTS ARE INSIDE VERTICES.
// It's been confirmed that user selected vertices.
selectFlag = selectFlag1 && selectFlag2;
System.out.println("SELECTION!");

/*
 * Check for adjacency before inserting a vertex. If user selected TWO vertices,
 * i.e., if selectFlag==true, then proceed to check for adjacency of those two
 * vertices.
 * If not, prompt the user to select vertices properly.
 */
if (selectFlag) {
    adjacencyFlag = checkForAdjacency(selectEventVertex1.getCenterX(),
        selectEventVertex1.getCenterY(), selectEventVertex2.getCenterX(),
        selectEventVertex2.getCenterY());
    if (adjacencyFlag)
        System.out.println("ADJACENTTTTTTTTTTTTT!!!!!!");
} else {
    Stage errorStage = new Stage();
    errorStage.initOwner(mainStage);
    errorStage.setTitle("ERROR!");

    VBox errorDialogBox = new VBox(20);
    errorDialogBox.getChildren()
        .addAll(new Text("Error: Please select vertices properly!"), new Text(
            "If you need any help, please choose help topics from the bottom of the window."));

    Scene errorScene = new Scene(errorDialogBox, 500, 75);
    errorStage.setScene(errorScene);
    errorStage.show();
}
}

mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
}

}

});
}

break;
```

Next is checking for adjacency of those vertices. In order to call this method, *selectFlag=true*.

It's worth noting that both MOPs and 2-trees will utilize this method.

```
// For 2-trees and MOPs, check if selected vertices are adjacent.
public boolean checkForAdjacency(double xcoord1, double ycoord1, double xcoord2, double ycoord2) {
    boolean event1 = false, event2 = false;
    for (int i = 0; i < edges.size(); i++) {
        /*
         * if two vertices are adjacent, then it means that have 1 common end point. The
         * two events check for that. event1 ==> checks for one of the two edges event2
         * ==> checks for 2nd edge.
         */
        event1 = (edges.get(i).getStartX() == xcoord1) && (edges.get(i).getStartY() == ycoord1)
            && (edges.get(i).getEndX() == xcoord2) && (edges.get(i).getEndY() == ycoord2);
        event2 = (edges.get(i).getStartX() == xcoord2) && (edges.get(i).getStartY() == ycoord2)
            && (edges.get(i).getEndX() == xcoord1) && (edges.get(i).getEndY() == ycoord1);
        if (event1 || event2) {
            System.out.println(event1 + " " + event2);
            return true;
        }
    }
    return false;
}
```

Next is inserting a vertex and adding new edges that connect this new vertex to the larger graph. This event handler will be *separate* for 2-trees and MOPs.

```
case "insert":
    switch (typeOfGraph) {
    case "2tree":
        mainScene.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                if (selectFlag) // 2 adjacent vertices have been selected.
                {
                    if (adjacencyFlag) { // vertices ARE adjacent
                        // therefore, call the method to insert vertex and draw edges.

                        /*
                         * drawIndices[0] = index where new vertex is vertices[] is present.
                         * drawIndices[1] = indices where the addition of new edges in edges[] starts
                         * from.
                         */

                        drawIndices = draw2Tree.insertVertexAndDrawEdges(event.getX(), event.getY());
                        for (int i = (int) drawIndices[0]; i < vertices.size(); i++)
                            border.getChildren().add(vertices.get(i));
                        for (int i = (int) drawIndices[1]; i < edges.size(); i++)
                            border.getChildren().add(edges.get(i));
                        mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
                    } else {
                        //if vertices are NOT adjacent => call error dialog box.
                        Stage errorStage = new Stage();
                        errorStage.initOwner(mainStage);
                        errorStage.setTitle("ERROR!");

                        VBox errorDialogBox = new VBox(20);
                        errorDialogBox.getChildren().addAll(
                            new Text("Error: Selected vertices are not adjacent!"), new Text(
                                "If you need any help, please choose help topics from the bottom of the window."));

                        Scene errorScene = new Scene(errorDialogBox, 500, 75);
                        errorStage.setScene(errorScene);
                        errorStage.show();
                        mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);

                        errorStage.setScene(errorScene);
                        errorStage.show();
                        mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
                    }
                }
                mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
            }
        });
        break;
    case "MOP":
```

The above code shows the event handler for INSERTING a vertex. The following code shows the method that *actually* defines where the new insertions should be.

```

// Outerplanarity is not even checked because 2-trees don't guarantee it.
public double[] insertVertexAndDrawEdges(double vertexX, double vertexY) {
    double indices[] = new double[2];

    Circle circle = new Circle();
    circle.setCenterX(vertexX);
    circle.setCenterY(vertexY);
    circle.setRadius(20);
    circle.setFill(Color.BURLYWOOD);

    // Edges are drawn.
    Line line1 = new Line(vertexX, vertexY, selectEventVertex1.getCenterX(), selectEventVertex1.getCenterY());
    Line line2 = new Line(vertexX, vertexY, selectEventVertex2.getCenterX(), selectEventVertex2.getCenterY());

    /*
     * Sizes of array are stored in indices[] because I only want to draw the edges
     * that are not displayed yet. If ENTIRE vertices[] or edges[] arrays are
     * returned, then ALL edges will be drawn over again and this is both a waste of
     * time and space.
     */
    indices[0] = vertices.size();
    indices[1] = edges.size();

    vertices.add(circle);
    edges.add(line1);
    edges.add(line2);

    return indices;
}

```

## MAXIMAL OUTER PLANAR GRAPH

To avoid confusion, I will not repost the screenshots of the event handler for *selection* of vertices. It's literally the same as 2-trees which is shown in the above section. The difference between them came only during the *insertion of vertices*.

The MOP graph *ALSO* requires the calling of *checkForAdjacency ()* method. The screenshot of the method is shown in the 2-tree, but it will once again be showed here as well (since it's very small).

```

// For 2-trees and MOPs, check if selected vertices are adjacent.
public boolean checkForAdjacency(double xcoord1, double ycoord1, double xcoord2, double ycoord2) {
    boolean event1 = false, event2 = false;
    for (int i = 0; i < edges.size(); i++) {
        /*
         * if two vertices are adjacent, then it means that have 1 common end point. The
         * two events check for that.
         * event1 ==> checks for one of the two edges
         * event2 ==> checks for 2nd edge.
         */
        event1 = (edges.get(i).getStartX() == xcoord1) && (edges.get(i).getStartY() == ycoord1)
            && (edges.get(i).getEndX() == xcoord2) && (edges.get(i).getEndY() == ycoord2);
        event2 = (edges.get(i).getStartX() == xcoord2) && (edges.get(i).getStartY() == ycoord2)
            && (edges.get(i).getEndX() == xcoord1) && (edges.get(i).getEndY() == ycoord1);
        if (event1 || event2) {
            System.out.println(event1 + " " + event2);
            return true;
        }
    }
    return false;
}

```

In addition to checking for adjacency, this method should also check for outerplanarity of the graph.

The code for outerplanarity is displayed below:

```
public boolean checkForOuterPlanarity(double vertexX, double vertexY) {
    for (int i = 0; i < edges.size(); i++) {
        /*
         * Outerplanarity means that edges should not intersect with the inside of the
         * graph. There will be TWO edges drawn in a MOP. SO: CHECK IF EACH EDGE
         * INTERSECTS WITH ALREADY EXISTING EDGES. This method checks whether the new
         * edges follow outerplanarity. These edges are not yet displayed onto the
         * screen, but are identified in geometry.
         */

        // NEW EDGE 1 ==> Check whether it intersects with any of the inner edges
        if (Line2D.linesIntersect(vertexX, vertexY, selectEventVertex1.getCenterX(),
            selectEventVertex1.getCenterY(), edges.get(i).getStartX(), edges.get(i).getStartY(),
            edges.get(i).getEndX(), edges.get(i).getEndY())) {
            /*
             * IF THEY DO INTERSECT, MAKE SURE THAT INTERSECTION DOESN'T MEAN HAVING A
             * COMMON ENDPOINT!!!!!!!!!!
             */
            if ((edges.get(i).getStartX() == selectEventVertex1.getCenterX()
                && edges.get(i).getStartY() == selectEventVertex1.getCenterY())
                || (edges.get(i).getEndX() == selectEventVertex1.getCenterX()
                && edges.get(i).getEndY() == selectEventVertex1.getCenterY()))
                ;
            else
                return false;
        }

        // NEW EDGE 2 ==> Check whether it intersects with any of the inner edges
        if (Line2D.linesIntersect(vertexX, vertexY, selectEventVertex2.getCenterX(),
            selectEventVertex2.getCenterY(), edges.get(i).getStartX(), edges.get(i).getStartY(),
            edges.get(i).getEndX(), edges.get(i).getEndY())) {
            /*
             * IF THEY DO INTERSECT, MAKE SURE THAT INTERSECTION DOESN'T MEAN HAVING A
             * COMMON ENDPOINT!!!!!!!!!!
             */
            if ((edges.get(i).getStartX() == selectEventVertex2.getCenterX()
                && edges.get(i).getStartY() == selectEventVertex2.getCenterY())
                || (edges.get(i).getEndX() == selectEventVertex2.getCenterX()
                && edges.get(i).getEndY() == selectEventVertex2.getCenterY()))
                ;
            else
                return false;
        }
    }

    return true;
}
```

Addressing this comment:

“IF THEY DO INTERSECT, MAKE SURE THAT INTERSECTION DOESN'T MEAN HAVING A COMMON ENDPOINT!!!!!!!!!!”

The condition that follows this comment is extremely important. Without this condition, no edge will be displayed because the program thinks that the user is violating outerplanarity. Intersection of inner edges *don't* include ‘intersections’ with endpoints. To avoid this, the nested *if-else* is used.

Therefore, the event handler for inserting new vertex is shown below:

```
case "MOP":
    mainScene.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            if (selectFlag) // 2 adjacent vertices have been selected.
            {
                if (adjacencyFlag) { //if vertices are adjacent
                    //if new additions are outerplanar
                    if (drawMOP.checkForOuterPlanarity(event.getX(), event.getY())) {
                        System.out.println("OUTERPLANAR!!!");
                        drawIndices = drawMOP.insertVertexAndDrawEdges(event.getX(),
                            event.getY());
                        for (int i = (int) drawIndices[0]; i < vertices.size(); i++)
                            border.getChildren().add(vertices.get(i));
                        for (int i = (int) drawIndices[1]; i < edges.size(); i++)
                            border.getChildren().add(edges.get(i));
                        mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
                    }
                    //if not outerplanar, initiate a dialog box.
                    else {
                        Stage errorStage = new Stage();
                        errorStage.initOwner(mainStage);
                        errorStage.setTitle("ERROR!");

                        VBox errorDialogBox = new VBox(20);
                        errorDialogBox.getChildren().addAll(
                            new Text("Error: Outerplanarity is not preserved! Try again!"),
                            new Text(
                                "If you need any help, please choose help topics from the bottom of the window."));
                        Scene errorScene = new Scene(errorDialogBox, 500, 75);
                        errorStage.setScene(errorScene);
                        errorStage.show();
                        mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
                    }
                }
                //if not adjacent at all, initiate a dialog box.
            }
        }
    });
```

```
        errorStage.show();
        mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
    }
}
//if not adjacent at all, initiate a dialog box.
else {
    Stage errorStage = new Stage();
    errorStage.initOwner(mainStage);
    errorStage.setTitle("ERROR!");

    VBox errorDialogBox = new VBox(20);
    errorDialogBox.getChildren().addAll(
        new Text("Error: Selected vertices are not adjacent!"), new Text(
            "If you need any help, please choose help topics from the bottom of the window."));
    Scene errorScene = new Scene(errorDialogBox, 500, 75);
    errorStage.setScene(errorScene);
    errorStage.show();
    mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
}
mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
});
break;
case "help-topics": {
```



The *actual* insertion of vertices is also the same as 2-trees. The method is identical to the method of insertion in 2-trees, so it's not displayed here.

## K4 PLANAR GRAPH

The code for initialization is posted in the beginning of this section.

As for the data structures, the entire credit goes to the usage of *triangles* (*ArrayList<Polygon>*). Otherwise, I can safely say that it would've been *very* difficult in implementing this graph. After initialization (done automatically upon generating the graph), the user can insert new vertices or choose to color the graph. In the case he chooses to add new vertices, he will have click somewhere within the graph.

Usually, that '*somewhere*' will be enclosed in a triangle. This triangle is detected by the *triangles* *ArrayList*. If the point clicked is within one of the triangles in the list, then a point is automatically into the graph. That point will become the new vertex and three new edges will be drawn, connecting the new vertex's center to the enclosing triangle.

The event handler for the K4 planar graph is posted below:

```
case "k4planar": {
    mainScene.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent mouseEvent) {
            System.out.println("mouse click detected! " + mouseEvent.getSource());
            newIndices = drawK4Planar.splitIntoTriangles(mouseEvent.getSceneX(),
                mouseEvent.getSceneY());
            for (int i = (int) newIndices[0]; i < vertices.size(); i++)
                border.getChildren().add(vertices.get(i));
            for (int i = (int) newIndices[1]; i < edges.size(); i++)
                border.getChildren().add(edges.get(i));
            mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
        }
    });
    break;
}
```

The code for the actual method for splitting into triangles is posted below:

```

public double[] splitIntoTriangles(double xcoord, double ycoord) {

    double indices[] = new double[3];

    if (xcoord == 0 || ycoord == 0)
        return indices;

    Circle circle = new Circle();
    circle.setCenterX(xcoord);
    circle.setCenterY(ycoord);
    circle.setRadius(20);
    circle.setFill(Color.BURLYWOOD);

    // System.out.println("AFTER RETURN STATEMENT");
    int index = -1;
    for (int i = 0; i < triangles.size(); i++) {
        if (triangles.get(i).contains(xcoord, ycoord))
            index = i;
    }

    // The points forming the area of polygon are denoted as point1, point2, point3
    double point1X = triangles.get(index).getPoints().get(0);
    double point1Y = triangles.get(index).getPoints().get(1);
    double point2X = triangles.get(index).getPoints().get(2);
    double point2Y = triangles.get(index).getPoints().get(3);
    double point3X = triangles.get(index).getPoints().get(4);
    double point3Y = triangles.get(index).getPoints().get(5);

    // New edges are initialized.
    Line line1 = new Line(point1X, point1Y, xcoord, ycoord);
    Line line2 = new Line(point2X, point2Y, xcoord, ycoord);
    Line line3 = new Line(point3X, point3Y, xcoord, ycoord);

    /*
     * The original polygon is divided into 3 smaller polygons. These polygons are
     * not displayed. They're used for geometrical purposes to identify the

```

```

/*
 * The original polygon is divided into 3 smaller polygons. These polygons are
 * not displayed. They're used for geometrical purposes to identify the
 * triangles.
 */
Polygon polygon1 = new Polygon();
polygon1.getPoints().addAll(new Double[] { point1X, point1Y, point2X, point2Y, xcoord, ycoord });
Polygon polygon2 = new Polygon();
polygon2.getPoints().addAll(new Double[] { point1X, point1Y, point3X, point3Y, xcoord, ycoord });
Polygon polygon3 = new Polygon();
polygon3.getPoints().addAll(new Double[] { point2X, point2Y, point3X, point3Y, xcoord, ycoord });

// System.out.println(index);

indices[0] = vertices.size();
indices[1] = edges.size();
indices[2] = triangles.size();

vertices.add(circle);

edges.add(line1);
edges.add(line2);
edges.add(line3);

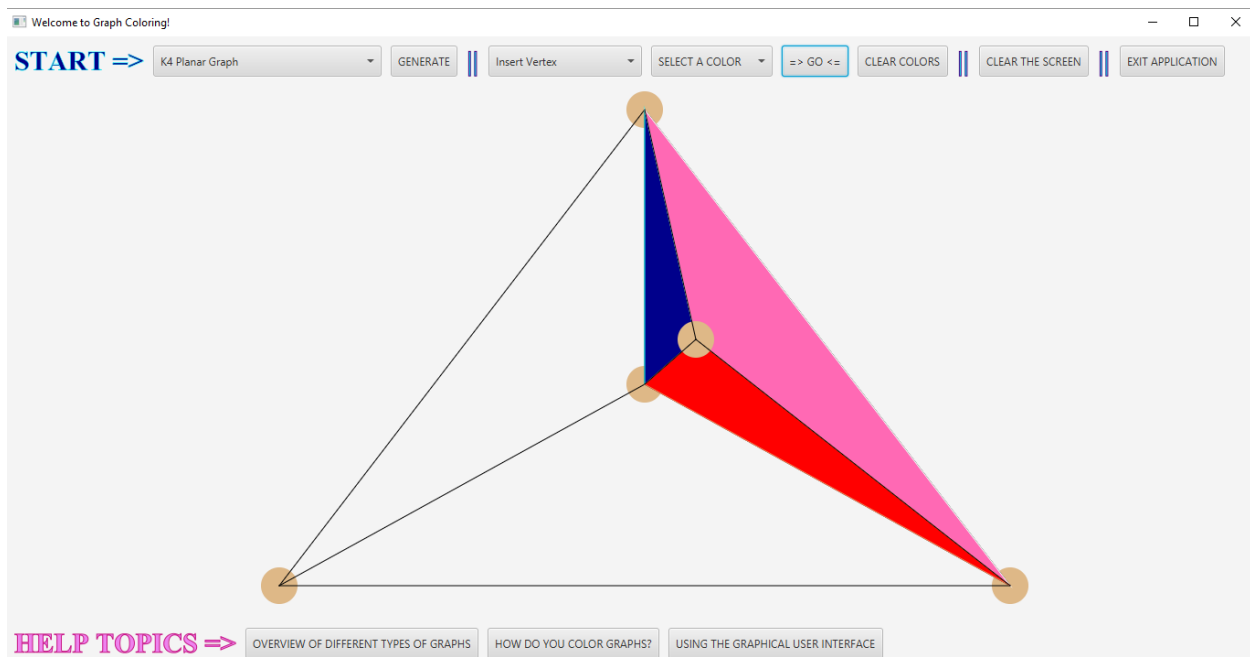
triangles.add(polygon1);
triangles.add(polygon2);
triangles.add(polygon3);

return indices;
}

```

As you can see, the code is relatively small, but it is quite redundant. The polygons above (in this case, they're all triangles) take coordinates as parameters.

In case it's difficult to visualize, the following is a picture of the code's infrastructure. Obviously, this will *NOT* be displayed to the user. It's only for understanding purposes. The three colored triangles will be added into the list of polygons.



## PLANAR CUBIC GRAPHS (K4, K6, K8)

All five planar cubic graphs have the same event handler.

Adding vertices and edges is same for all five types of the planar cubic graphs. The only difference is that the code for the initialization of the five graphs will differ.

```
break;
case "k4planarcubic":
case "k6planarcubic":
case "k8planarcubic1":
case "k8planarcubic2":
case "k8planarcubic3":
    mainScene.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            System.out.println("INSIDE HANDLE");
            if (counter == 0) {
                counter++;
                xcoordinate1 = event.getX(); // user made only 1 selection, so do nothing
                ycoordinate1 = event.getY();
                mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
            } else if (counter == 1) {
                counter = 0; // user made a second selection, so counter=0
                xcoordinate2 = event.getX();
                ycoordinate2 = event.getY();
                drawIndices = addVerticesAndEdges(xcoordinate1, ycoordinate1, xcoordinate2,
                    ycoordinate2);
                for (int i = (int) drawIndices[0]; i < vertices.size(); i++)
                    border.getChildren().add(vertices.get(i));
                for (int i = (int) drawIndices[1]; i < edges.size(); i++) {
                    try {
                        border.getChildren().add(edges.get(i));
                    } catch (IllegalArgumentException e) {
                    }
                }
                mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
            }
        }
    });
    break;
}
```

The code that actually acknowledges user's selections and makes sure that the user selected two edges from an empty cycle is listed below in screenshots:

```

    * For the planar cubic graphs ==> K4, K6, K8 (all 3 types). Total: 5 graphs.
public double[] addVerticesAndEdges(double xcoord1, double ycoord1, double xcoord2, double ycoord2) {

    double indices[] = new double[2];

    Circle circle1 = new Circle();
    circle1.setRadius(20);
    circle1.setFill(Color.BURLYWOOD);

    Circle circle2 = new Circle();
    circle2.setRadius(20);
    circle2.setFill(Color.BURLYWOOD);

    boolean circleFlag1 = false, circleFlag2 = false;
    indices[0] = vertices.size();

    for (int i = 0; i < edges.size(); i++) {
        if (edges.get(i).contains(xcoord1, ycoord1)) {
            circle1.setCenterX(xcoord1);
            circle1.setCenterY(ycoord1);
            vertices.add(circle1);
            circleFlag1 = true;
        }
    }

    for (int i = 0; i < edges.size(); i++) {
        if (edges.get(i).contains(xcoord2, ycoord2)) {
            circle2.setCenterX(xcoord2);
            circle2.setCenterY(ycoord2);
            vertices.add(circle2);
            circleFlag2 = true;
        }
    }

    Line edge1 = new Line(), edge2 = new Line(), edge3 = new Line(), edge4 = new Line(), line = new Line();

    /*
    * When a user adds a new within an empty cycle, delete the old edges and draw

```

```

/*
 * When a user adds a new within an empty cycle, delete the OLD edges and draw
 * new edges. Each affected edge will break into 2 edges. The old one is deleted
 * and in its place, 2 new ones are added and joined at the same place the user
 * clicks on.
 */
int indexOfEdgeDelete1 = -1, indexOfEdgeDelete2 = -1;
// Circle vertex1, vertex2;
if (circleFlag1 == true && circleFlag2 == true) {
    for (int i = 0; i < edges.size(); i++) {
        if (edges.get(i).contains(xcoord1, ycoord1)) {
            indexOfEdgeDelete1 = i;
            edge1 = new Line(circle1.getCenterX(), circle1.getCenterY(), edges.get(i).getStartX(),
                edges.get(i).getStartY());
            edge2 = new Line(circle1.getCenterX(), circle1.getCenterY(), edges.get(i).getEndX(),
                edges.get(i).getEndY());
        }
        if (edges.get(i).contains(xcoord2, ycoord2)) {
            indexOfEdgeDelete2 = i;
            edge3 = new Line(circle2.getCenterX(), circle2.getCenterY(), edges.get(i).getStartX(),
                edges.get(i).getStartY());
            edge4 = new Line(circle2.getCenterX(), circle2.getCenterY(), edges.get(i).getEndX(),
                edges.get(i).getEndY());
        }
    }
    line = new Line(xcoord1, ycoord1, xcoord2, ycoord2);
}
/*
 * To make sure that edges are not deleted due to wrong indices, first, delete
 * that edge that has the HIGHER index of the two.
 */
if (indexOfEdgeDelete1 < indexOfEdgeDelete2) {
    edges.remove(indexOfEdgeDelete2);
    edges.remove(indexOfEdgeDelete1);
} else {
    edges.remove(indexOfEdgeDelete1);
    edges.remove(indexOfEdgeDelete2);
}
}

```

```

        edges.remove(indexOfEdgeDelete1);
        edges.remove(indexOfEdgeDelete2);
    }

    /*
     * The edges that are added are given thickness, so that user faces less
     * difficulty while clicking an edge.
     */
    edge1.setStrokeWidth(7);
    edge2.setStrokeWidth(7);
    edge3.setStrokeWidth(7);
    edge4.setStrokeWidth(7);
    line.setStrokeWidth(7);

    /* The edges are added into the static array. */
    edges.add(edge1);
    edges.add(edge2);
    edges.add(edge3);
    edges.add(edge4);
    edges.add(line);

    indices[1] = 0;
    System.out.println("EDGES  " + edges.size());
    return indices;
}

```

## REJECTED DATA STRUCTURES

The most crucial part of this project is to be able to sync the vertices *and* edges without involving the user in any way. As per the rules of this project, in terms of space, adjacency/incidence lists/matrices were not used and therefore, rejected. Linked lists, stacks, queues, and other abstract data types were rejected as well because ArrayLists were extremely feasible in terms of time.

This program demanded a lot of usage from graphics, so managing time was one of the sensitive challenges. Space didn't prove to be that much of a problem, so the main focus was on managing time complexity in a way so that the user does not notice a visible lag while running his interface.

Arrays were considered first, but ArrayLists are an advanced version of arrays. Hence, these lists were used, and other data structures were rejected with another thought.

## SCREEN'S VISUALS (FROM USER'S PERSPECTIVE)

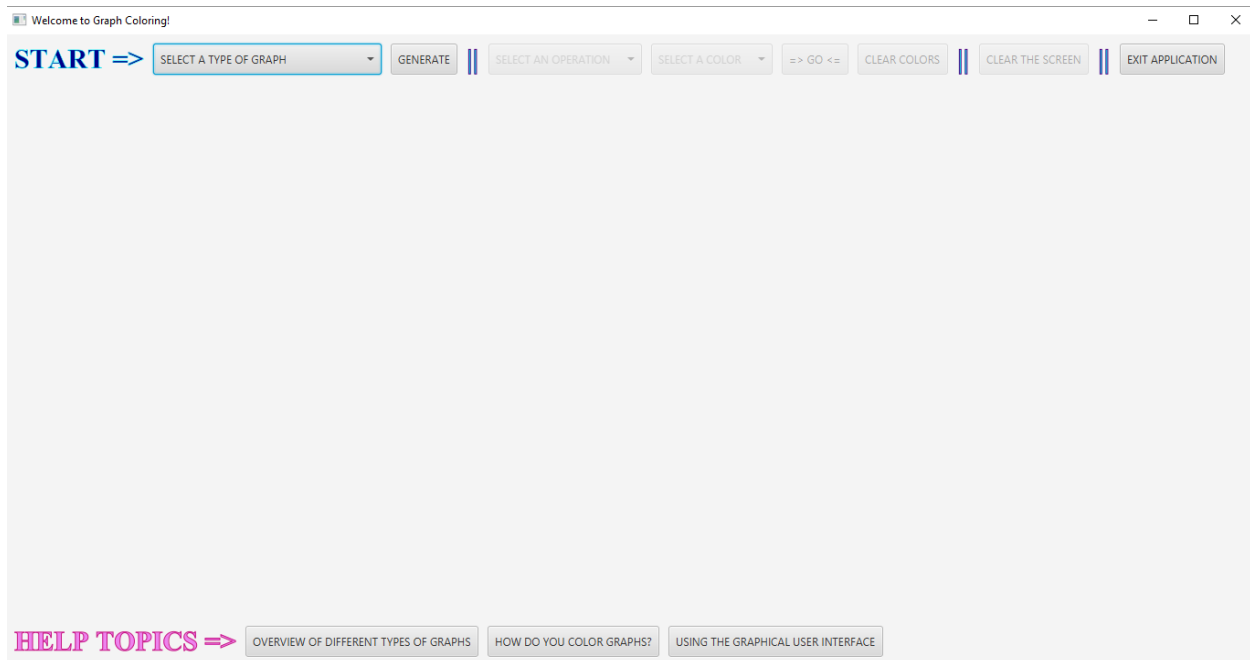
For the visuals, data structures were not used. It really wasn't about what structures were used. The entire interface depended on the *order* that data is organized rather than *how* it is organized.

One of the main problems encountered during programming is that for any method to hold any value whatsoever, the results of that method *must* be displayed (i.e., stated) *explicitly* within the *start ()* method.

Even if a method was called from the *start ()* method, the results weren't properly displayed. That was one of the problems encountered during development. Even though there were plenty of return statements, the *start ()* method would not display anything on the screen. Basically, everything should be added onto the *border* first (for proper layout organization). The *border* is added into the *mainScene (Scene)* which is later added and displayed through *mainStage (Stage)*.

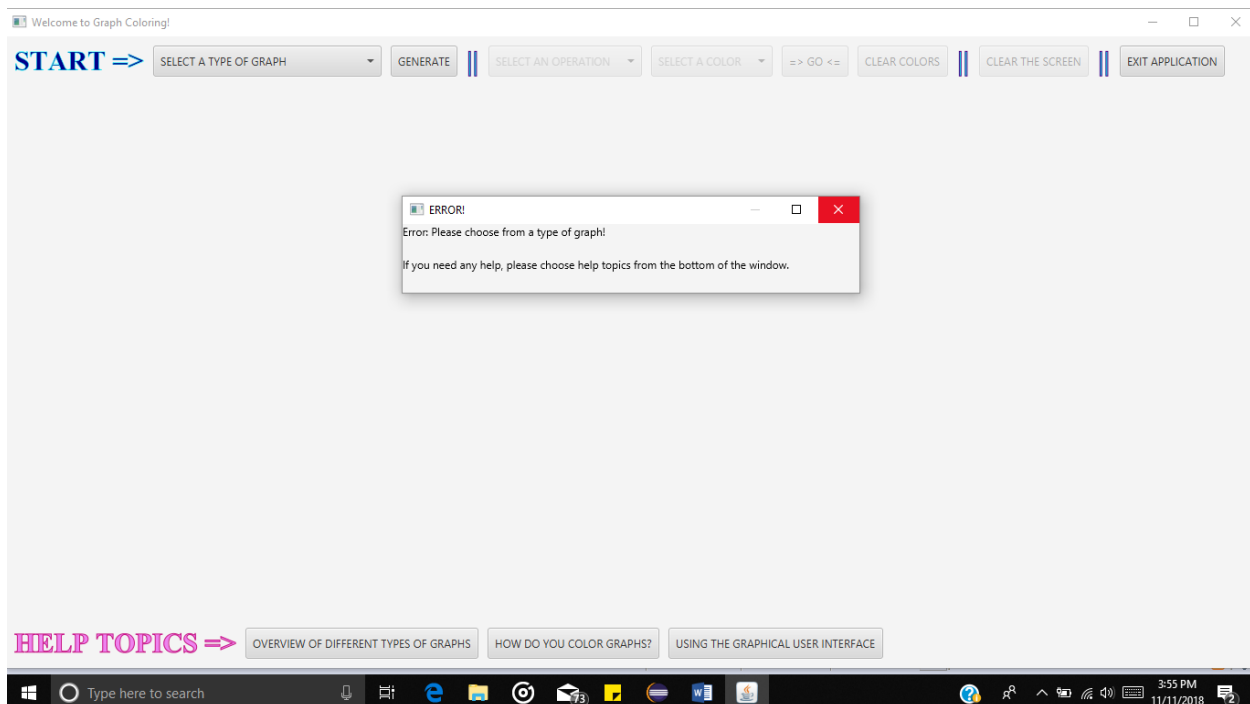
Many of the buttons and combo boxes were made completely static because they had to be utilized all throughout the program and it didn't seem feasible to reference the variable every time.

The following screenshot is a basic startup of the JavaFX application.



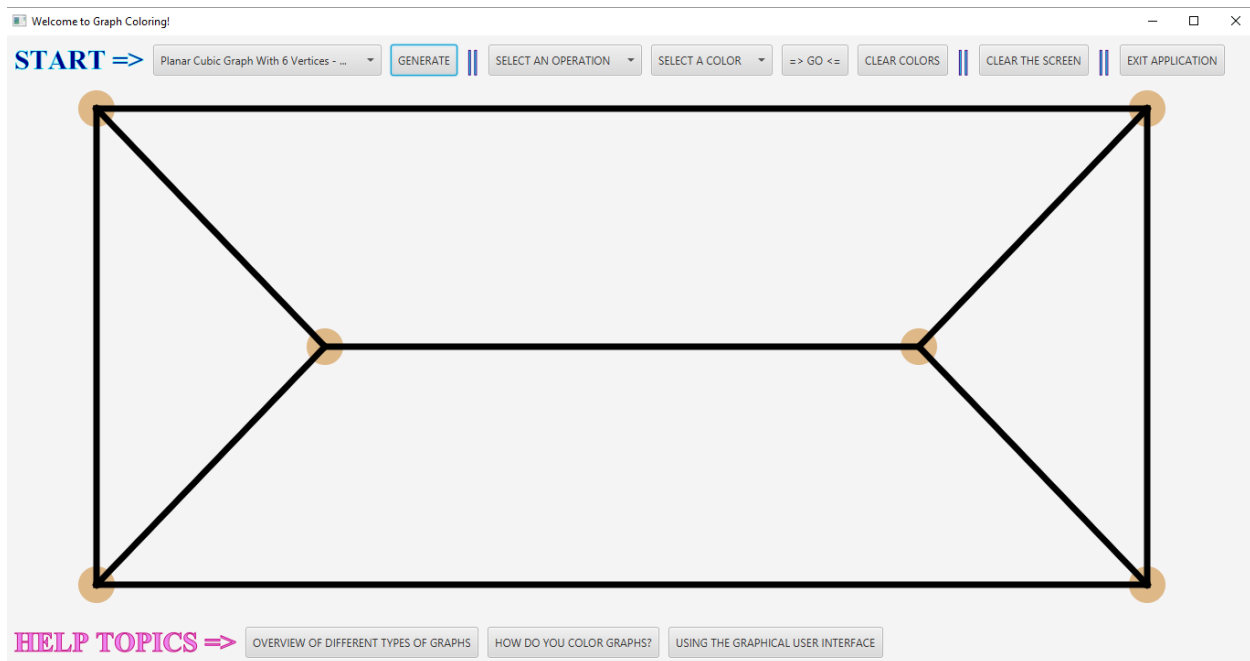
It's important to note that *all* buttons and combo boxes (in the top row) other than *graphBox* (*ComboBox*) and *generateGraph* (*Button*) will be disabled until and unless the user selects a graph and clicks on the button to generate it.

If the user still does *not* select a type of graph but clicks on generate anyway, then an error stage will open and prompt the user to select a type of graph before continuing.





If and *only if* the user selects a graph will all the buttons and combo boxes will be available for use. Depending upon the user's graph selection, the objects will be enabled.

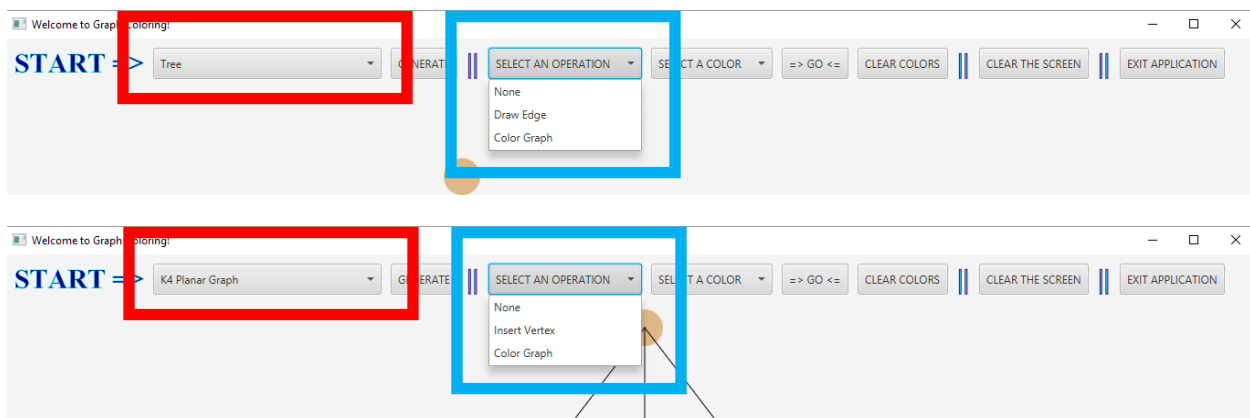


Just as a side note, in the above graph, the vertices of the graph are currently *not* colored. The tan, brownish color is chosen as a default color for the vertices.

The user can interact with the options as needed.

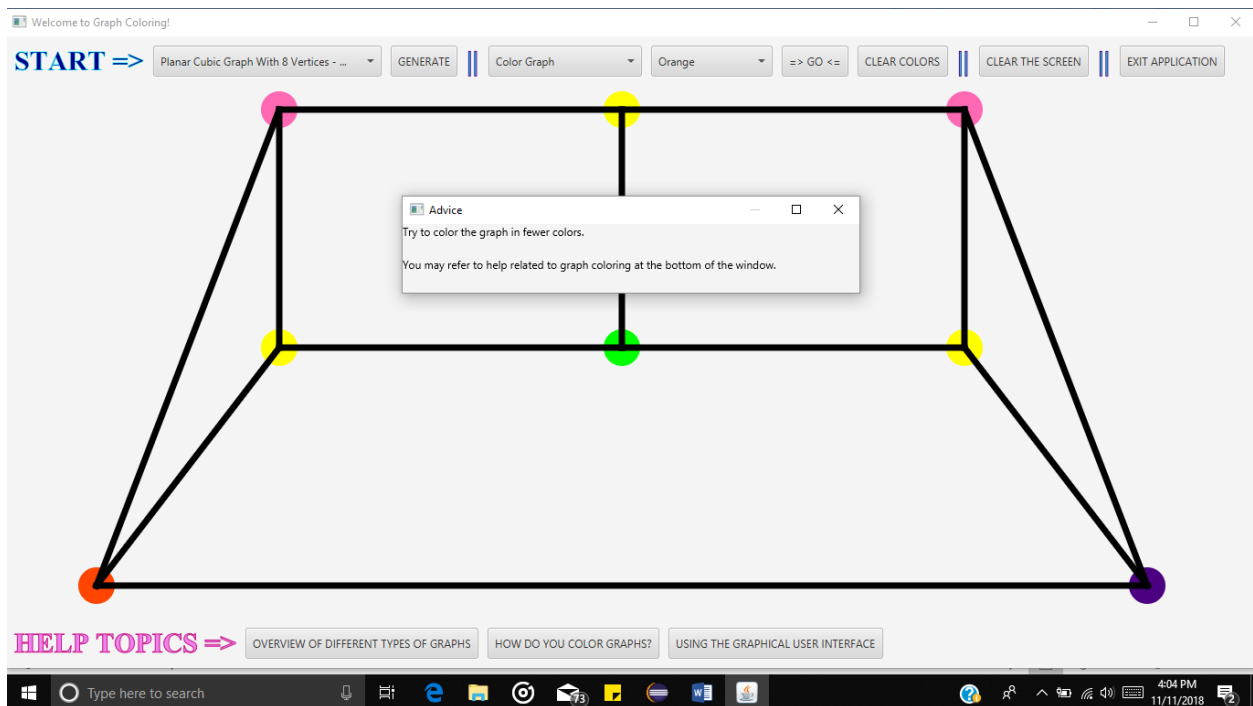
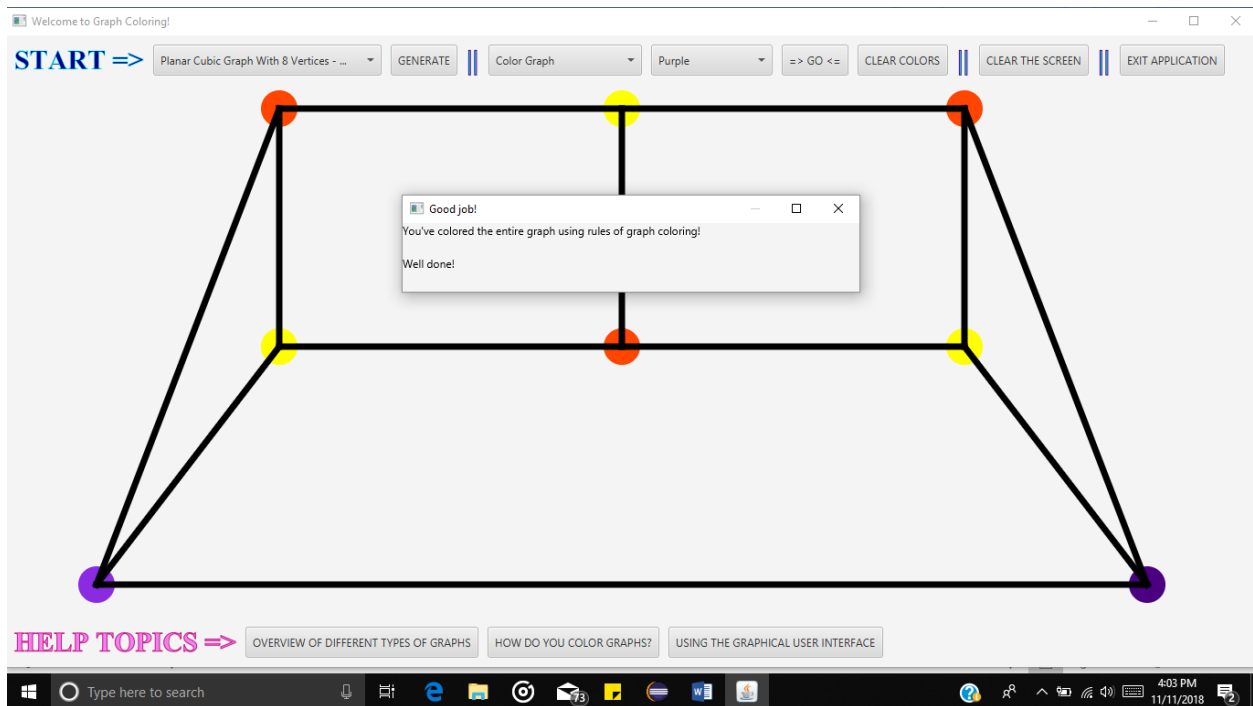
If he clicks on “*CLEAR THE SCREEN*”, then the window will revert to its original state (the first picture).

Each type of graph has *different* options to choose from. For example, a tree will have the options “*Draw Edge*” or “*Color Graph*”, whereas a K4 planar graph will have the options “*Insert Vertex*” or “*Color Graph*”. The only option that every graph has in common is “*Color Graph*”.

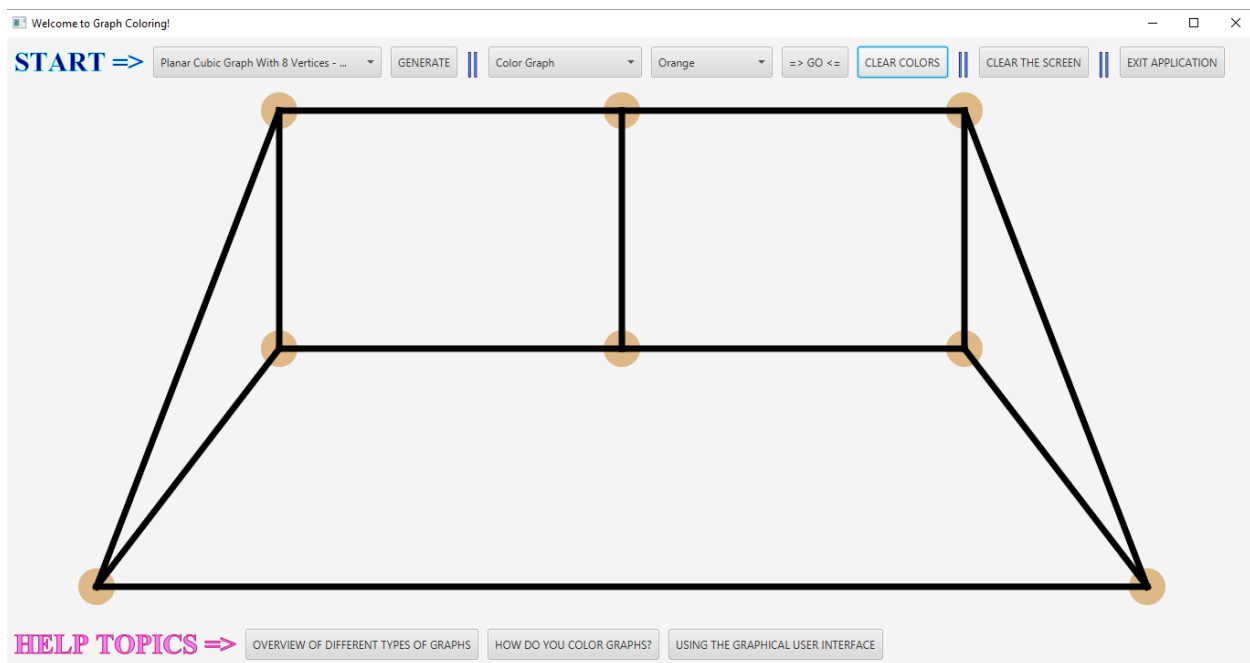


User must specifically choose “*Color Graph*” in order to be able to color. He can’t just select a color and press “*Go*”.

ALSO, this entire project assumes that the maximum number of colors needed for efficient graph coloring is *four*. If the user manages to color the entire graph with less than or equal to four colors, then he’ll be applauded. Else, he’ll be advised to use fewer colors.



Next, if user clicks on “*CLEAR COLORS*”, he will see the original figure again with brownish vertices. All the colors will be cleared.



Displaying the screen’s visuals did not take any complicated data structures. It was merely about understanding how JavaFX works and how stages (window) and scenes (content inside window) are organized.

As explained in the very beginning of this document (in the identifiers section), the buttons, combo boxes, and text inside of those boxes are simply stored and displayed onto the scene in a *specific* order.

For the types of operations, initially, all operations are available. *But*, since the *operations* (*ComboBox*) is disabled (until user generates a graph), it’s not visible to the user. When a graph is generated, the unneeded operations are *removed*.

```

case "2tree":
    colorBox.setDisable(false);
    clearGraphColors.setDisable(false);
    clearScreen.setDisable(false);
    operations.setDisable(false);
    go.setDisable(false);

    operations.getItems().remove("Draw Edge");

    vertices = new ArrayList<Circle>();
    edges = new ArrayList<Line>();
    draw2Tree.initialization();
    for (int i = 0; i < vertices.size(); i++)
        border.getChildren().add(vertices.get(i));
    for (int i = 0; i < edges.size(); i++)
        border.getChildren().add(edges.get(i));
    break;

```

In the above code snippet, the type of graph that user chose is a 2-tree. The first five lines of code is enabling the buttons again. The next piece of code *removes* "Draw Edge". The next few lines of the code initializes the 2-tree.

The code snippets are similar to the other types of graphs as well. Each time a new graph is chosen the *vertices* and *edges ArrayLists* are reset.

```

case "2tree":
    colorBox.setDisable(false);
    clearGraphColors.setDisable(false);
    clearScreen.setDisable(false);
    operations.setDisable(false);
    go.setDisable(false);

    operations.getItems().remove("Draw Edge");

    vertices = new ArrayList<Circle>();
    edges = new ArrayList<Line>();
    draw2Tree.initialization();
    for (int i = 0; i < vertices.size(); i++)
        border.getChildren().add(vertices.get(i));
    for (int i = 0; i < edges.size(); i++)
        border.getChildren().add(edges.get(i));
    break;

```

Next is graph coloring. In graph coloring, each edge's endpoints were vital. In this project, each edge's endpoints are not in some randomly placed positions by the user. They all coincide with some vertex's center. Therefore, using these edges' endpoints, finding out the color associated with that vertex was easy.

First, the event handler is called:

```
e {
    switch (operation) {
        case "color": {
            mainScene.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
                @Override
                public void handle(MouseEvent mouseEvent) {
                    int index = vertices.size() + 1;
                    boolean flag = true;
                    /*
                     * Call graphColoring() method. If the coordinate that the user clicked on
                     * matches a vertex in the list of vertices, then call graphColoring() method.
                     */
                    for (traverseIndex = 0; traverseIndex < vertices.size(); traverseIndex++) {
                        if (vertices.get(traverseIndex).contains(mouseEvent.getX(), mouseEvent.getY())) {
                            index = traverseIndex;
                            graphColoring(vertices.get(traverseIndex).getCenterX(),
                                vertices.get(traverseIndex).getCenterY());
                        }
                    }

                    /*
                     * In the case where no colors are used yet (all vertices are null of any
                     * color), then don't bother checking with any conditions.
                     */
                    if (listOfColors.size() == 0) {
                        vertices.get(index).setFill(selectedColor);
                    }
                    /*
                     * ELSE: listOfColors contains the list of colors that the user used so far in
                     * all of the vertices.
                     */
                    else {
                        for (traverseIndex = 0; traverseIndex < listOfColors.size(); traverseIndex++) {
                            /*
                             * If the user picks a color that was ALREADY assigned to the ADJACENT VERTICES
                             * OF THE SELECTED VERTEX, then don't let user perform the action. Instead
```

```

/*
 * If the user picks a color that was ALREADY assigned to the ADJACENT VERTICES
 * OF THE SELECTED VERTEX, then don't let user perform the action. Instead,
 * display an error.
 */
if (selectedColor.equals(listOfColors.get(traverseIndex))) {
    Stage errorStage = new Stage();
    errorStage.initOwner(mainStage);
    errorStage.setTitle("ERROR!");

    VBox errorDialogBox = new VBox(20);
    errorDialogBox.getChildren()
        .addAll(new Text("Error: Please choose another color!"), new Text(
            "If you need any help, please choose help topics from the bottom of the window."));
    Scene errorScene = new Scene(errorDialogBox, 500, 75);
    errorStage.setScene(errorScene);
    errorStage.show();
    flag = false;
    break;
}
}
}
if (flag) {
    vertices.get(index).setFill(selectedColor);
}
boolean checkIfGraphIsCompletelyColored = true;
Set<Color> usedColors = new LinkedHashSet<Color>();
for (int i = 0; i < vertices.size(); i++) {
    if (vertices.get(i).getFill().equals(Color.BURLYWOOD)) // if even one node isn't
                                                            // colored.
        checkIfGraphIsCompletelyColored = false;
    else
        usedColors.add((Color) vertices.get(i).getFill());
}
}

/*
 * This project assumes that the maximum number of colors to be used for
 * efficient graph coloring is FOUR
 */

```

```

    /*
     * This project assumes that the maximum number of colors to be used for
     * efficient graph coloring is FOUR.
     */
    if (checkIfGraphIsCompletelyColored) {
        if (usedColors.size() > 4) { // if user used more than 4 colors
            Stage errorStage = new Stage();
            errorStage.initOwner(mainStage);
            errorStage.setTitle("Advice");
            VBox errorDialogBox = new VBox(20);
            errorDialogBox.getChildren()
                .addAll(new Text("Try to color the graph in fewer colors."), new Text(
                    "You may refer to help related to graph coloring at the bottom of the window."));
            Scene errorScene = new Scene(errorDialogBox, 500, 75);
            errorStage.setScene(errorScene);
            errorStage.show();
        } else if (usedColors.size() <= 4) { // if user used less than 5 colors
            Stage errorStage = new Stage();
            errorStage.initOwner(mainStage);
            errorStage.setTitle("Good job!");
            VBox errorDialogBox = new VBox(20);
            errorDialogBox.getChildren().addAll(
                new Text("You've colored the entire graph using rules of graph coloring!"),
                new Text("Well done!"));
            Scene errorScene = new Scene(errorDialogBox, 500, 75);
            errorStage.setScene(errorScene);
            errorStage.show();
        }
    }
    mainScene.removeEventFilter(MouseEvent.MOUSE_PRESSED, this);
}
});
}
break;

case "insert":
    switch (typeofGraph) {

```

The method that performs the actual graph coloring is provided below:

```

* Implement rules of graph coloring ==> Common method for all graphs
public void graphColoring(double vertexX, double vertexY) {
    // This method notes the colors of the ADJACENT vertices
    listOfColors = new ArrayList<Color>();
    for (int j = 0; j < edges.size(); j++) {

        /*
        * Each edge's starting and ending points will correlate with a vertex's center.
        * In this case, compare the ends of the edge to see which colors the adjacent
        * vertices are placed in. Compare with the starting ends first.
        */

        /*
        * Run a loop and see each vertex's adjacent vertices' used colors. Add these
        * used colors into listOfColors<>
        */
        if (edges.get(j).getStartX() == vertexX && edges.get(j).getStartY() == vertexY) {
            for (int k = 0; k < vertices.size(); k++) {

                if (vertices.get(k).getCenterX() == edges.get(j).getEndX()
                    && vertices.get(k).getCenterY() == edges.get(j).getEndY()) {

                    if (vertices.get(k).getFill().equals(Color.RED))
                        listOfColors.add(Color.RED);
                    else if (vertices.get(k).getFill().equals(Color.ORANGERED))
                        listOfColors.add(Color.ORANGERED);
                    else if (vertices.get(k).getFill().equals(Color.YELLOW))
                        listOfColors.add(Color.YELLOW);
                    else if (vertices.get(k).getFill().equals(Color.LIME))
                        listOfColors.add(Color.LIME);
                    else if (vertices.get(k).getFill().equals(Color.BLUE))
                        listOfColors.add(Color.BLUE);
                    else if (vertices.get(k).getFill().equals(Color.INDIGO))
                        listOfColors.add(Color.INDIGO);
                    else if (vertices.get(k).getFill().equals(Color.BLUEVIOLET))
                        listOfColors.add(Color.BLUEVIOLET);
                    else if (vertices.get(k).getFill().equals(Color.HOTPINK))
                        listOfColors.add(Color.HOTPINK);
                }
            }
        }
    }
}

```



```

    }
}
}
// Similarly, check the OTHER ends of the edges.
if (edges.get(j).getEndX() == vertexX && edges.get(j).getEndY() == vertexY) {
    for (int k = 0; k < vertices.size(); k++) {
        if (vertices.get(k).getCenterX() == edges.get(j).getStartX()
            && vertices.get(k).getCenterY() == edges.get(j).getStartY()) {
            if (vertices.get(k).getFill().equals(Color.RED))
                listOfColors.add(Color.RED);
            else if (vertices.get(k).getFill().equals(Color.ORANGERED))
                listOfColors.add(Color.ORANGERED);
            else if (vertices.get(k).getFill().equals(Color.YELLOW))
                listOfColors.add(Color.YELLOW);
            else if (vertices.get(k).getFill().equals(Color.LIME))
                listOfColors.add(Color.LIME);
            else if (vertices.get(k).getFill().equals(Color.BLUE))
                listOfColors.add(Color.BLUE);
            else if (vertices.get(k).getFill().equals(Color.INDIGO))
                listOfColors.add(Color.INDIGO);
            else if (vertices.get(k).getFill().equals(Color.BLUEVIOLET))
                listOfColors.add(Color.BLUEVIOLET);
            else if (vertices.get(k).getFill().equals(Color.PINK))
                listOfColors.add(Color.PINK);
        }
    }
}
}
}
}

```

## METHOD OF DEBUGGING

The code was debugged as thoroughly as possible, especially because I faced genuine troubles with the last project's graphics implementation. This was done by inserting a print statement at almost every single crucial operation.

Therefore, *please* ignore the print statements that may appear within the lines of code. They were only used for understanding the program's status and to track the user's dynamic whereabouts.

## TECHNOLOGY

- Software: this program was implemented and run using Eclipse (Photon version) IDE.
- Hardware: Windows 10, 64-bit Operating System with Intel- i5 processor

## REFERENCES

1. **Oracle's Documentation** (the specific links used are provided as many as possible within the code itself (wherever applicable))
2. Eclipse and IntelliJ's **default provided code**
3. **Class notes:** In addition to the class notes, I also utilized my notebook from my UG in India where I had taken Graph Theory in the last semester (January-April 2018).

4. I do want to point out that I watched **tutorials** from the YouTube channel *thenewboston* but that was near the middle to end of the deadline for the first short project. I hadn't realized that applets were out of date and long gone until I watched the tutorials for JavaFX. I did want to convert the applet code into JavaFX for my *first* short project, but it was already way too late by then. So, the tutorials that I watched then were a huge help for me in developing the second short project.