Programming Report
Graph Editor

Philip Oetinger (s2966018)
Said Faroghi (s3000966)
CS7

June 20, 2017

1 Introduction

The program is a basic graph editing program. It has the ability to save and load as many graph models the user has memory for. Each model contains vertices and edges that connect the vertices in any orientation and layout the user wishes for. The models can be switched between and viewed/edited in the window pane through two methods of interaction. The first is a traditional button bar system with a movable and detachable button bar with common functions add/remove edges and vertices, rename vertices, undo, and redo actions. The user can also use the right click on the screen to open a useful ”quick action” that does things based on the context of the location of click. For example, right clicking an object would give options of rename or delete. The program attempts to keep the information displayed in a user friendly environment.

2 Program Organization

2.1 Model

The model contains the GraphModel, GraphVertex, and GraphEdge. These three classes are all of the objects required to store, manipulate, and view a graph. The vertices each have a name and rectangle. An edge connects two vertices in the graph. The model takes all of these objects and stores them in arrays so they can be used. These classes are the data of the program thus are obviosuly part of the model.

The main class of the model(GraphModel) is the observable, such that View can observe it for changes and then update itself when it does so. Interestingly enough, it is also an observer, because some changes are happening outside of the scope of the model. For example, renaming a vertex changes the name of the vertex, but that's not recorded in the model because the method runs in the GraphVertex class. However if the GraphModel observes the vertices, then the vertex notifies the model of a change, and then the model notifies the view of a change, therefore updating the view.

2.2 View

The view contains a GraphFrame, and a GraphPanel. These two objects run all the graphical needs of the program. The frame displays the menubar, and the panel shows the graph model visually as well as a button bar to help the user manipulate the data.

The view observes the model, as per MVC requirements, so that its notified of any changes in the model in order to update itself when it does so. Upon an update, the components are repainted.

2.3 Controller

The beef of the program is in the many controllers that the user interacts with. There are three main ways that the user can edit the graph. The first being the ButtonBar. This is a bar that can freely float and contains all the actions that the user may wish to do. The user could add/remove vertices and edges, rename a vertex, undo, and redo previous actions. The button bar provides a traditional method of data entry and manipulation.

The next main controller is the Popup Menu. This little menu shows up when the user right clicks and allows for ”quick access” to actions. For example if the user right clicks somewhere in the draw area, the action ”Add Vertex” will appear in the popup menu. This quickly adds a vertex with a custom name exactly where the user clicked. The user can do everything the button bar can do right from the popup menu.

The Menu Bar is a controller that does program wide actions such as saving and loading, or managing models to be edited. The user may even open a new window to display the model to another screen if they wish.

The last controllers are those behind the scenes that interact with the user through listening to the mouse. The dragging and selection controllers can receive information from the mouse and enable further actions to take place.

All of the actual actions that the button bar and popup window do are also contained within an ”Undoable Action” class. This way actions are stored and can be undone or redone at a later time. This way the user can quickly restore an accidental deletion of a vertex.

3 Component Explanation

3.1 Model

3.1.1 GraphModel

Data

The GraphModel contains a list of vertices and a list of edges. It also stores a list of vertex and edge lists. This enables the GraphModel to not only know it’s current state, but other states if the user wishes to work on several graphs at once. It also contains the X and Y coordinates of the mouse if needed.

Methods

GraphModel has basic vertex and edge manipulation methods like adding vertices, deleting vertices, and likewise with edges, as well as get general information from them like how many edges and vertices exist. It also includes a function to switch between models.

Saving and loading is done in the model as well, using a custom saving method on the format based on the example graphs given, however without the first line informing how many edges and vertices there are.

Basic undo and redo functions exist in the model as well.

3.1.2 GraphVertex

Data

The vertex’s name and a rectangle determnining the dimensions and location of the vertex exist in this class.

Methods

Vertex manipulation methods like naming the vertex, and setting/getting the size and location of the rectangle representing the vertex, exist here.

3.1.3 GraphEdge

Data

It contains a list of two vertices that this edge connects.

Methods

A function returning one of the vertices it connects exists here.

3.2 View

3.2.1 GraphFrame

Data

All the code necessary to buld the entire frame (and as such, the graph editor itself) reside here. It contains the panel itself and all the necessary controller components needed to run the program.

Methods

No methods except for the constructor, which contains the frame-building code.

3.2.2 GraphPanel

Data

The panel draws the model on the screen. It draws them edge first, then the vertex, and finally the name. This way the pieces layer correctly on top of each other.

Methods

All methods related to drawing on the screen exist here. There is a method to draw the edges, a method to draw the vertices, and the paintComponent method, where these two auxiliary methods are called. The paintComponent method actually draws the objects on the screen using the coordinates of the edge's connecting vertices, and the rectangle object stored in the vertex. Finally the vertex is automatically resized to fit the name of it.

3.3 Controller

3.3.1 MenuBar

Data

The menu bar contains two JMenus to control two different types of actions. Program specific ones like saving and loading are in the File Menu, and model specific ones in the Model menu. The menu bar contains all the code for it's buttons as well since they are only created and used once when starting the program. They also are not being changed often so do not need their own separate file.

Methods

Each menu item is added with a custom action listener that executes the proper action. For example when saving the method gets the path from the user, and then sends it to the model's save method.

3.3.2 ButtonBar

Data

The ButtonBar is what the user interacts with to manipulate the data in the model. It contains the buttons: Add Vertex, Remove Vertex, Add Edge, Remove Edge, Rename Vertex, Undo, and Redo. The button bar is like the menu bar in that it contains all the setup code for it's buttons. It is only a few lines and won't need individual classes for each button.

Methods

Each button is added as a JButton with a custom action listener that runs the appropriate undoable edit in the model. There is also a method to disable and enable buttons when they are needed. There is also a method to toggle what buttons are visible based on if a vertex is selected or not.

3.3.3 Popup Menu

Data

The popup menu actually contains two popup menus - one when you right click the background to add a vertex, and another when you right click a vertex, to manipulate the vertex via different options. It also contains the mouse coordinates of the click. The rest of the code is the code for the buttons in the menus.

Methods

In the constructor, each button is added as a JButton with a custom action listener that runs the appropriate undoable edit in the model. The action listeners are similar to what runs in the buttonbar, since it contains the same options. Upon mouse release method, the buttons are added to the menu.

3.3.4 Dragging Controller

Data

This is used to take care of vertex dragging actions. It contains the model, and the start coordinates of the drag.

Methods

Upon mouse pressed, the start coordinates of the click are fetched and the start coordinates of the drag are calculated. Upon mouse dragged, the coordinates of the vertex are changed to reflect the changing coordinates of the mouse dragging.

3.3.5 Selection Controller

Data

This has a model and the buttonbar. The button bar is needed to set what buttons are enabled or not, based on whether a vertex is selected.

Methods

Upon mouse pressed, a method is called to check whether the mouse lies within the boundaries of a vertex, and if so, select that vertex. If multiple vertices exist in that same location, we select the top vertex.

Interesting extra note

In the GraphFrame component of View, where these controller objects are created, this controller had to be created BEFORE the dragging controller. Because both of these objects contain a ”mousePressed” function, they are both run at the ”same” time when the mouse is pressed, but not in reality, because there is only one thread running.

The dragging controller depends on the selected vertex to get its coordinates, so obviously this mousePressed needs to run first, before the other one. Interestingly the solution is to create this object before the other one, however we are unaware of the technicalities as to why. Creating the dragging controller before this one results in funky errors because it fetches the coordinates of the vertex before its actually selected.

3.3.6 Undoable Actions

Data

Each un-doable class is a separate action that manipulates the model and stores the changes made incase the user wishes to undo or redo the action.

Methods

Each un-doable action has a constructor that takes the model and possibly any additional parts it needs to modify the model. It then performs the changes by calling the model methods. It also stores the change in the Undoable Edit (itself).

Each Un-doable also has an undo and redo method defined. The first line is always `super.undo();` which tells the manager that an undo has been performed that that the action can be re-done. After that it is action specific on how to undo and redo an action, using the change stored in the object itself.