Object Oriented Programming
Graph Editor

Ivana Akrum (s2861348)
Marco Breemhaar (s2990571)

June 19, 2017

1 Introduction

The graph editor is a program in which graphs can be made by connecting vertices to each other using edges. When the program is launched, the user is presented with a blank screen with a menu bar on top, with options `File` and `Edit`. Using the Edit menu, the user can immediately start designing a graph by adding/removing vertices and edges. Of course there is also functionality to undo and redo the edits.

Instead of starting from scratch, the user can also choose to load an existing graph from a file. This can be done by choosing the `Open` option from the `File` menu. The menu also contains a `Save` option to save a graph to a file.

In addition, the `File` menu also has options to exit the program, create a new graph from scratch, and to work in multiple frames.

The program is built using the Model-View-Controller pattern. This means that the program consists of three packages: the model, the view and the controller. The model represents the graph itself. The view takes care of actually displaying the graph on the screen. The controller contains everything necessary for the user to interact with the program.

There is one additional utility package which contains the methods that take care of loading and saving files.

2 The model package

The model represents the graph. Since a graph is in essence a very simple concept, we do not need a lot of classes for describing it.

2.1 GraphModel

The `GraphModel` class contains a list of all vertices and edges in the graph. To make edits possible, there are methods to do the following (here after referred to as *operations*):

- Add a vertex: creates a new instance of the `GraphVertex` class and adds it to the list of vertices.
- Remove a vertex: removes the edges that are connected to a given vertex and then removes that vertex itself.
- Add an edge: adds an edge to the list of edges and also adds it to the list of connections of two given vertices.
- Remove an edge: removes a given edge from the list of connections of connected vertices, and then removes the edge itself.

The operations should not be called directly, but are instead controlled by classes in the operations package. In this package, each operation has its own equivalent class that inherits from *AbstractUndoableEdit*. These classes were added to implement the undo and redo functionality.

Each of these classes follows the same template. The constructor of the class contains the set of actions that is normally performed when the method is called in the controller (which is more or less equivalent to the operations described above). The undo method explains what happens when the operation is undone. For example, undoing the removal of a vertex is equal to adding that same vertex to the model again. The redo method reverts the undo method. If the undo added a vertex, the redo removes it again. Simply said, the undo method does the opposite of what is done in the constructor of the class. The redo method, in turn, does the opposite of the undo method.

To implement the undo and redo functionality, there is an instance of an UndoManager in the GraphModel. This keeps track of all edits that happened, so that it can undo and redo them. The GraphModel contains methods ending in 'Undoable' that add instances of the associated classes to the UndoManager. These methods are, in turn, called in the controller to make a change in the model.

The GraphModel also contains an instance of the GraphVertex class called selected, which keeps track of the currently selected vertex, a temporary edge that connects the currently selected vertex to the mouse, and a state, which is 'idle' by default.

2.2 GraphVertex

As the name suggests, `GraphVertex` represents a vertex in the graph. A vertex has a name, a string and a list of edges it is connected to.

In the constructor of the graph, the name and the properties of the rectangle are set. The name and rectangle fields both have their own getter and setter to retrieve and change these values.

Using the `addConnection` and `removeConnection` methods, edges are added to the list of connected edges. This list can be retrieved as a whole using `getConnections`.

2.3 GraphEdge

The `GraphEdge` class is a very simple class. It has just to fields which contain the start and end vertices. These vertices are chosen in the constructor of the class. Both the start and the end have their own getter and setter to retrieve and change these values. However, the setters are not used in the current version of the program. They are left in since they may be useful in later versions of the program.

3 The view package

The view is rather straightforward. It contains two classes: GraphFrame, and GraphPanel. The GraphFrame creates the frame, which shows the panel as described in GraphPanel. GraphFrame does not have any methods, only its constructor. The constructors of GraphPanel and GraphFrame are similar: they contain methods that describe what the component looks like (its dimension, backgrond colour etcetera), as well as adding potential listeners. Both the frame and the panel have a mouse listener as well as a mouse motion listener, which keeps track of how the cursor moves over the component, as well as what actions (e.g. clicking) are performed with the mouse. The frame has these mouse listeners because it is possible to have more than one frame, and it is necessary to know what frame we are currently in. The panel has the mouse listeners so that it can keep track of the user input which will be processed by classes in the controller.

The GraphPanel displays the model by painting its vertices and its edges. The vertices are drawn as white rectangles, where the rectangles size and position are given in the GraphVertex class itself. They also have centered strings as names. Depending on whether the vertex is currently selected or not, the border and text are either blue or black. The edges are drawn as black lines

originating and ending in the (x, y) centers of the vertices its connects. If there is a temporary edge (which is the case if an edge is currently being added), that edge is drawn first in blue. The edges are drawn underneath the vertices.

4 The controller package

4.1 Menu bar

The menu bar is a simple `JMenuBar` which contains two `JMenus`, `File` and `Edit`. The file menu contain options to load/save a model, and to exit the program. It also has a sub-menu with the options to create a new model or a new window. The edit menu contains options to undo/redo and add/remove edges and vertices. Each each these options is a menu item that relies on an action. Clicking on the menu item, fires the action. The classes that describe these actions can be found in the actions package. Each action class has a method that determines when it is possible to fire the action. If the action is disabled, the associated menu item cannot be clicked on and is greyed out.

The action classes have a method actionPerformed that determines what happens when the action fires. As is noticeable, each action is equivalent to an operation in the model package. The actionPerformed method thus calls the method in the GraphModel that adds an instance of an operations class to the UndoManager (the methods ending in 'Undoable'). Since the action to be performed is described in the constructor of these classes, creating an instance of said class thus results in the action being performed.

An interesting design choice here is that the actions related to adding/removing edges only change states of the GraphModel. This is done because another vertex needs to be selected to know which vertices the edge connects. Further processing of these actions is done in the SelectionController class.

4.2 SelectionController

Finally, the controller package contains the SelectionController class. The clicking of items in menus does not have to be processed separately, but if the user clicks anything else, this does have to be processed separately. The SelectionController processes information regarding mouse events. The following mouse (motion) events are processed in the selection controller, though there exist more than just these:

- mouse press: when the left button of the mouse input is pressed
- mouse click: when the left button of the mouse input is pressed and released
- mouse move: when the cursor moves across the panel/frame
- mouse drag: when the cursor moves across the panel/frame while the left button is being pressed

To process both normal mouse events (pressing and clicking), and mouse motion events (moving and dragging) the SelectionController inherits from the MouseInputAdapter. Alternatively, it could have implemented both the MouseListener and MouseMotionListener, but in that case the class would be expected to implement all methods in said interfaces. Since not all mouse events lead to an action (releasing a pressed button, for example), we do not want to add all methods to the class.

Next to methods that determine what happens if a mouse event is detected, the SelectionController has multiple helper methods for processing the information. It also has an instance of a GraphVertex that keeps track of the most recently selected GraphVertex.

4.2.1 Mouse Drag

Dragging the mouse while holding the shift button on the keyboard allows the user to move vertices to different positions. The method that processes mouse drag events calls a helper method. These helper method updates the position of the currently selected vertex by using its setRect() method to change the rectangle that represents the vertex.

4.2.2 Mouse Move

Moving the mouse while an edge is being drawn, will lead to a line being drawn between the last selected vertex and the mouse cursor. This line is represented by an edge that connects the selected vertex with a new vertex that represents the mouse. Said vertex is actually a small rectangle (width and height of 1, no name) at the cursor position.

4.2.3 Mouse Click

If a double click is registered, a text field becomes visible and editable at the location of the selected vertex. This text field is in an instance of the NameField class found in the Controller. Its instance is actually in the GraphModel, but it is hidden in the view, unless a double click is registered. The text field allows the user to change the vertex name (max 20 characters) by inputting a name and pressing enter.

4.2.4 Mouse Press

The mouse press event can lead to different responses. To determine what has to happen, states were implemented in the GraphModel. Before anything else is processed, if the ctrl or alt button on the keyboard are also pressed, a helper method is called that enlarges (ctrl + mouse press) or shrinks (alt + mouse press) the currently selected vertex up to certain limits.

If these keyboard keys are not detected, the method checks if the mouse press was at the location of a vertex by going over all vertices in the vertex list of the GraphModel, and seeing if the mouse press was inside one of these vertices. This process starts at the end of the vertex list to make sure the most recently added vertex is selected. If so, it will update both its own selected vertex and that of the GraphModel to the newly selected vertex through the changeSelection() method.

Before it does this, however, it first checks the state the model is in. If we are currently adding an edge, the edge will be drawn between the newly selected (not yet updated) vertex, and the most recently selected vertex before this new vertex (stored in the selected instance of the SelectionController). Removing an edge works similarly, except that the edge is removed instead of drawn. After finishing the process of adding/removing an edge, the state in the GraphModel is set back to 'idle'.

All of the mouse press processes could have been processed by a mouse click, but the mouse press event is easier to detect than a mouse click.

5 The utility package

The utility package contains the methods to save and load graph files. These files are in a plain text (.txt) file format. They are structured as follows:

1. The first line contains the number of vertices and the number of edges.
2. The following lines contain the vertices' information. Each line represents one vertex. The first four values are the properties of its rectangle: x position, y position, width and height respectively. The last value is the name of the vertex.

3. The last set of lines represent the edges. Each line contains the start and end point of a vertex. The vertices are identified by an integer which corresponds to the index of the list of vertices in the model.

The values on all lines are separated by a space.

5.1 The save method

The save method has one argument which specifies the model that has to be saved. Using a `JFileChooser` we ask the user to specify where the model has to be saved. When we know this, we can start writing to the file. To do this, we use a PrintWriter.

Firstly, we want to know the sizes the list of vertices and the list of edges. These values are now written to the file.

After this, we iterate over the vertices. For each vertex, we take the x position, y position, width, height and name. These are then written to the file at once.

Now, we iterate over the edges. For each edge, using the `indexOf` method, we get the indices of the start and end vertices of the edges. These are now also written to the file.

After writing the edges to the file, we are done writing so we can close the writer. The whole writing process is done within a try/catch structure since we may get an IO exception in case something goes wrong while writing. When we catch this exception, we will print to the console that the saving process has failed. We also print the stack trace of the error.

5.2 The load method

The load method is a little more complicated than the save method, since we now have to reconstruct a model using the information of the save file, instead of just taking the model and writing its information to a file.

However, we start the same way, by asking the user for the file we have to load using a `JFileChooser`. Before we start writing to this file, we also want to make sure we aren't adding to an existing model, so we have clear the current model. To do this, the list of vertices and the list of edges are cleared.

Now, we start reading from the file. We do this using a buffered reader. This allows us to read one line at a time. This is very useful, since the information of the different graph elements are separated by new lines.

The first line is read and we save the values as number of vertices and number of edges. We need this information for the next step.

We now iterate over the lines that represent the vertices. We know how many vertices there are since we read that information in the first line. We split this line using the space as separator. This returns an array of which the first four values are stored in the x position, y position, width and height fields. The remaining part of the line is the name of the vertex, but since it can contain spaces, it may be split up into different indices of the array. Therefore, we use a string builder put all the remaining indices of the array into one string. Now, we create a new vertex with these properties and we add it to the model's list of vertices.

Using another for-loop, we iterate over the lines which represent the edges. The two values in the line are split and saved as index of the start and end vertices. Using the array of vertices we already built, we retrieve the vertices that belong to these indices. Now, we simply create a new edge with corresponding start and end vertices. The edge is also added to the connection list of both the start and the end vertex.

Now we are done reading so we can close the buffered reader. We only have to update the model to show it in the view.

In case any errors occurred, we catch a couple of exceptions. These could be caused by the save file not existing, the save file being corrupted/not the correct structure, or some other IO exception. In all cases, the cause is printed, alsong with the stack trace.