Object-Oriented Programming
Programming Report
Graph Editor

s3750663 and s3301419

Jun 2020

1 Introduction

The *McGraph* graph editor is an application for creating, customizing and solving arbitrary graphs. It supports directed, undirected, and parallel edges, however it does *not* support looping edges. The program comes with a *solver* module that allows searching for the shortest path between two nodes (vertices), calculating the shortest distance from one node to every other node in the graph, as well as graph coloring. The solver is robust, and works for any graph that can be created inside of McGraph, including graphs with negative cycles. A relatively high degree of customization is supported, allowing the user to create visually pleasing graphs as they desire. Figure 1 shows how the McGraph application window looks.

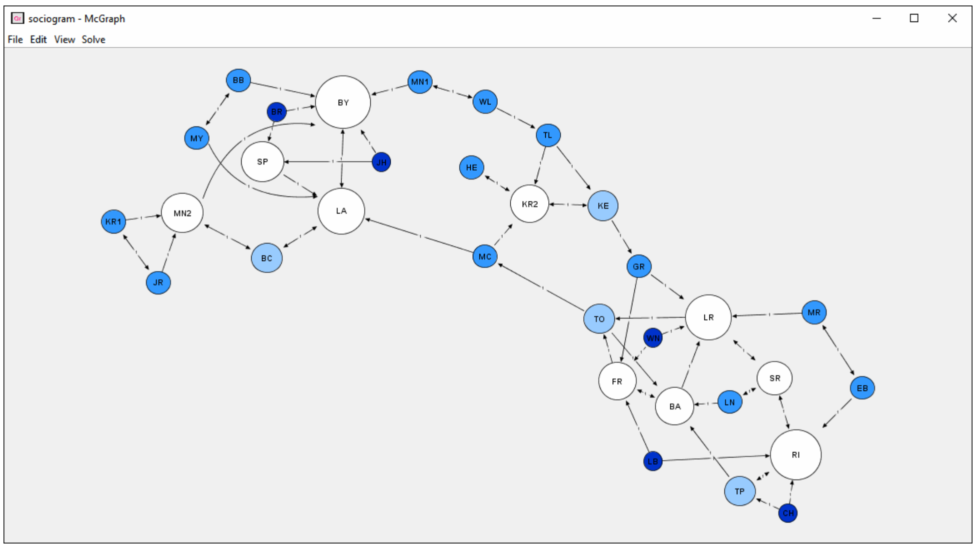


Figure 1: The application window of Mc Graph running on Windows 10.

1.1 Assignment Requirements

McGraph was created for the Graph Editor assignment for the Object Oriented Programming course at the University of Groningen. The assignment consisted of creating an application to manipulate undirected, simple, unweighted graphs without self loops in a graphical user interface (GUI). The following is a summary of the requirements for the assignment.

- The *graphs* consist of both *nodes* and *edges*.
- The graph should be graphically displayed to the user.
- Button to create a new graph.
- Buttons for adding and removing nodes and edges.
- User can label each node with some text that can be modified.
- Saving and loading functionality for the graph to a basic file format (`.graph` files).
- Buttons for saving and loading a graph from a file.
- Buttons are disabled when their preconditions are not met.
- Every operation should be *undo*-able, and *redo*-able.
- Nodes should be selected when clicked on.
- Nodes can be moved by *dragging*.
- When started from the *command-line* the application should load the graph given as the first command-line argument.

McGraph satisfies all of these requirements, but these are just a small fraction of the functionality that McGraph supports. Numerous extensions were added to the program that will be discussed in the following sections.

2 Program design

The project is divided into four main packages following the Model-View-Controller (MVC) pattern. These packages are `model`, `view` and `controller` with an addition of a `utils` package. In this section, we will go over those packages and discuss the interaction between components as well as some design decisions.

2.1 Model

The `model` package contains all logic of the application, independent from the user interface. It contains `Graph`, `Node`, a `GraphUndoManager`, and some other helper classes.

`Graph` is the main component here that keeps track of all nodes and edges in the graph. It does so by observing the edges and nodes and on update, notifies all observers (i.e. panel and buttons in order for them to determine whether they are enabled or visible). Similarly, an edge needs to keep track of its nodes in order to update its curve and weight position when its nodes change; this is also solved with the observer pattern.

Although observer pattern is supposed to be used for the view observing the model, we decided to also implement this between parts of the model, as we wanted to avoid storing references to the `Graph` in `Node` and `Edge`, as well as to avoid a node knowing if it has edges or not. This enabled us to abstract the interaction between these components.

Lastly, the model includes a `GraphUndoManager` which extends from javaâĂŹs `UndoManager` class and allows checking if an undo is possible. With this, we can check whether there have been unsaved changes.

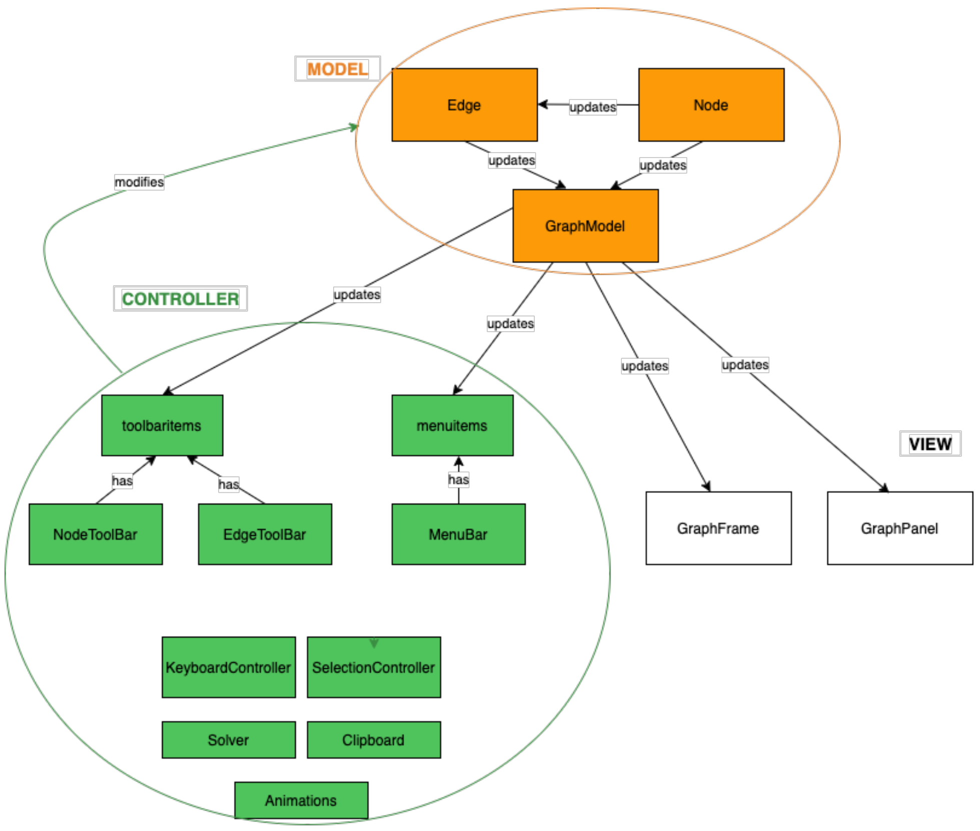


Figure 2: Design and MVC pattern in McGraph Editor.

2.2 View

The `view` displays information from the model to the user. It contains `Frame` and `Panel`, which inherit from `JFrame` and `JPanel` respectively.

`Frame` sets the frame of the application, and handles some things for loading a new frame and exiting the application by closing the frame, for example prompting the user to save unsaved changes on close. It observes the `Graph`, as it needs to update the title of the window depending on the file that is open and shows whether the graph contains unsaved changes.

The `Panel` handles how the model data is displayed to the user and thus also observes the `Graph`. It draws all edges and nodes and handles zooming in and out. Important to note is that the panel always draws the *visual* data of nodes and edges, as that ensures that `Animations` that update the visual data are displayed correctly.

2.3 Controller

The `controller` is responsible for interacting with user input and changing the model accordingly. It is structured as follows: it receives mouse input in the `SelectionController`, keyboard inputs in the `KeyboardController`, has a `MenuBar` with various menu items, and two different toolbars that pop up when either nodes or edges are selected respectively. Additionally, the user can also right click for a menu, which is handled in `PopUpMenu`.

Edits to the graph in any of those controller classes are not handled directly by the model, but rather through `UndoableEdits` in order to make edits undo- and redo-able. Edits for nodes and edges were abstracted, so that it can handle all possible edits to existing node and edges. The `NodeAndEdgeEdit` that does this keeps track of the old node and edge data, which is used in undo and redo. Undo basically resets the nodes' and edges' data to their old version. Redo resets the data to a new version.

Although this abstraction means instantiating many new NodeData and EdgeData objects, this massively reduces the complexity of the program, as all node and edge edits can be handled by a single edit class. It is also a highly reusable way to modify nodes and edges since if any new data is added to NodeData or EdgeData, the NodeAndEdgeEdit will work with these modified versions without any modification.

Additionally, the controller handles animations, meaning it changes the visual data of nodes and edges to animate something, for example a node temporarily increasing in size when clicked.

Furthermore, the controller contains the classes Clipboard and Solver. Both are implemented using the singleton design pattern, as the application at all times should have only one clipboard and solver.

Clipboard enables copy-pasting, while the Solver can find the shortest path on the graph, marks distances from a starting node to any other node, and performs graph coloring using a greedy algorithm.

Exploring the graph in the Solver is done using the Bellman-Ford algorithm which also works for graphs with negative cycles. In such graphs, the distance between 2 nodes can be infinitely negative, as paths can go through the same edges with negative weights over and over again.

Graph coloring in the Solver is done using a greedy algorithm. The exact algorithm used *does not* always find optimal graph colorings, however it does indeed find the optimal coloring for most graphs, as the algorithm assigns colors to nodes in a close to optimal order using a heuristic.

2.4 Utilities

The util package, contains utility and helper classes TextUtil, MathUtil, ListUtil, and KeyUtil.

KeyUtil solves some cross-platform issues, such as the delete key code not being mapped to the same key on Windows, Mac and Linux. It also includes interfaces that help abstract classes Additionally, utils contains a Diamond2D class, which fully implements the Shape interface and serves as a shape for drawing nodes.

As can be seen in Figure 2, interactions between the packages are as follows: view observes the Graph, Graph observes Edges and Nodes, Edges observe the Nodes they connect. Buttons and tool bars also observe the Graph so that they can be enabled and disabled at the correct times. The utils package is used throughout the whole application and thus not included in the diagram.

3 Evaluation of the program

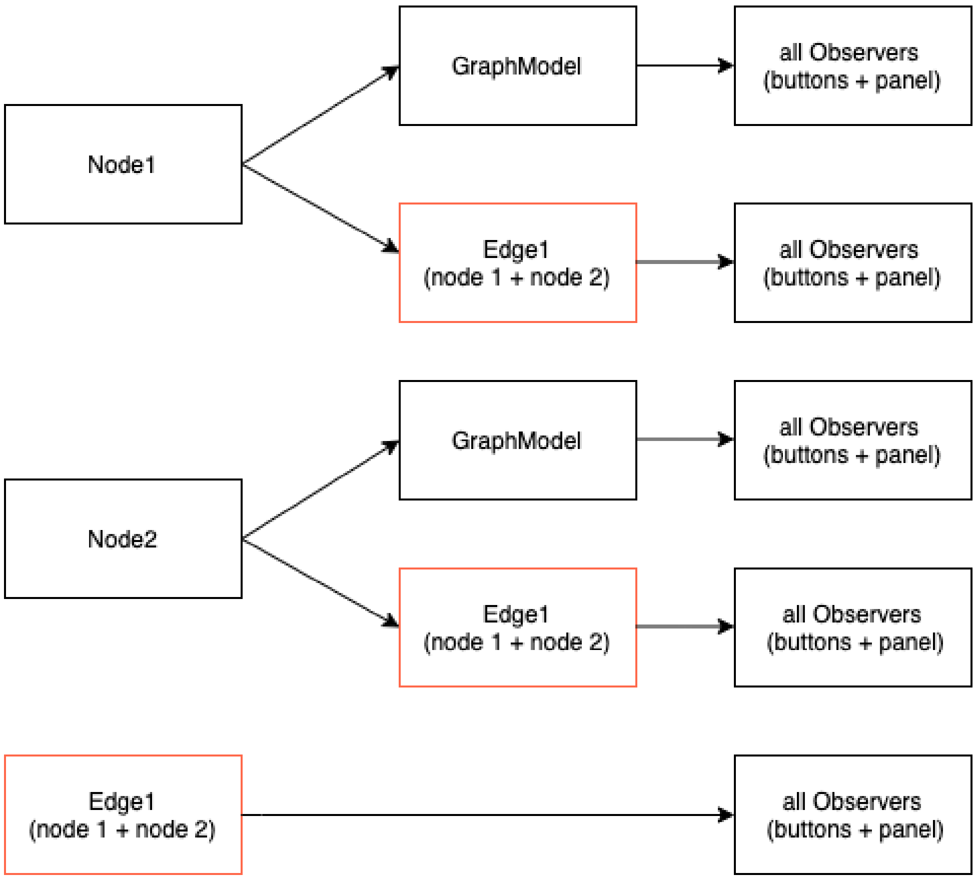
Overall, the program has evolved to be much greater than what we anticipated when starting out.

3.1 Testing

We tested our program with several graphs, and did not detect any bugs. When testing an large densely connected graph with 1024 nodes, the performance of the program obviously decreased. The GUI was still very responsive for small tasks such as selecting nodes, and changing its color. Solver operations still ran at a reasonable time. However, when *all* nodes and edges are selected at once and then dragged around the view, the responsiveness was quite low.

3.2 Performance

At the moment, edits are handled by iterating over the list of nodes and edges that are selected and editing each one. That decreases performance when editing 1000 nodes at once. Additionally, in the observables notify their observers on every change. That means that to move two nodes connected with one edge the following observers are notified by observables:


GraphModel
all Observers (buttons + panel)
Node1
Edge1 (node 1 + node 2)
all Observers (buttons + panel)
GraphModel
all Observers (buttons + panel)
Node2
Edge1 (node 1 + node 2)
all Observers (buttons + panel)
Edge1 (node 1 + node 2)
all Observers (buttons + panel)

(1).png (1).png

Figure 3: Notifications to observers when changing 2 nodes that are connected with 1 edge.

Both nodes notify the `Graph` and their edge, thus edge and graph are notified twice. Additionally the node itself also registers movement and notifies the `Graph`. Although the user moves three elements, nine notifications are sent and the graph redraws about five times. Important to note is also that this is for a tiny movement, so this sequence of events can be called hundreds of times a second. This is obviously very inefficient, however in most common use cases the GUI remains fast and responsive, so we decided against optimizing it.

3.3 Drawing Performance

When drawing the graph on the screen, we make sure to only draw the nodes and edges that are actually visible (view frustum culling), and we also avoid drawing small details when the user zooms out the view to the point where the detail is not visible. All in all the drawing performance of the program is reasonable.

3.4 Code

We've had quite some problems with the text-field that is used to update a node's text. We display the text on the node as it changes, which presents as a difficulty when it came to implemented undo-functionality. Thus, the old name must be stored when the user starts changing it, and when the user is done changing, the edit should be registered. We implemented this using a focus listener: focus gained stores the old names and focus lost registers the edit. However, in some cases the focus is not lost, e.g. when the user presses the same node again that they were editing. That meant that the edit was not registered and we manually had to lose the focus of the text field when a user presses a node. We encountered similar problems for example with undoing a moving node. This being said, often the undo could have been implemented in a simpler form, for example by prompting the user to press enter to register the change, or not displaying the change until the user clicks 'ok' or presses 'enter'.

4 Extension of the program

In this section, the extensions we added to our McGraph Editor are described shortly:

- **Directed edges and weights:** The graph includes directed edges, either to the first or second node or even both. Additionally, edges have weights attached to them that can be modified.
- **Graph Solver:** The edge weights and directions are used in the solver that we implemented. It can find the shortest path on the graph, as well as mark distances from a start node to all other nodes. For this a modification of the Bellman-Ford algorithm is used, as we also allow for *negative weights*. The solver is implemented using the Singleton design pattern.
- **Animations:** Our graph includes animations, which respond to user input: when a user clicks a node, it temporarily pops up and gets bigger. When hovering over the buttons to change the color of the nodes/edges, the nodes/edges get animated in that color for as long as the user hovers over the button. Afterwards animations are stopped and the normal color shows back up.
- **Keyboard shortcuts:** The user can use keyboard shortcuts for almost every action. For example, control+Z for undoing an action, control+Y for redoing it, and so forth.
- **Multi-selection:** When clicking control (or command on mac), the user is able to select multiple nodes and edges. Additionally clicking command while pressing and moving the mouse will result into a selection rectangle that selects every node and edge within it. The multiple selected nodes/edges can be edited at the simultaneously (e.g. resizing, changing text, etc.).
- **Copy-paste:** The graph has a Clipboard, that clones nodes and edges and can add them back in. Clipboard is similar to Solver, as it is also a Singleton class.
- **Changing styles of nodes and edges:** Shapes of nodes as well as styles of edges can be changed. Nodes can have the shape of a rectangle, a rounded rectangle, an ellipse or a diamond. This diamond classes was also implemented ourselves (`Utils.Diamond2D`). Edges on the other hand can have the form of elbow-joint, curve or spline.
- **Selecting and moving edges:** Edges can be selected with a mouse click for easy deletion, as well as for moving the edge.
- **Resizing nodes:** Nodes can be resized. Resizing the node to a very small size will eventually lead to the fact that the node is turned around.
- **Zooming in and out:** The graph lets the user zoom in and out, as well as move the view around.
- **UI extensions:**
  - *Custom colors:* Both nodes and edges have the possibility of customizing colors. Nodes have three different colors that can be changed: fill color, border color and text color.
  - On right click, a *pop up menu* shows up for the user to choose potential actions.

    - The McGraph has a *custom icon* as a logo that shows up when instead of the default icon of swing the frame is opened.
    - *Custom look and feel* is used for the swing application to make it feel more native to the operating system.
    - *Tool bars:* when clicking on a node or edge, the respective tool bar is opened. From here edits, such as changing the node text, shape, style, color, etc. can be made.

- **Cross-platform behavior:** It doesn't only implement a custom look and feel, but also adds some cross-platform functionality, such as a delete button that works on both Windows and Mac, and menu key shortcuts that are native to the respective platforms.
- **Dirty tracking:** We extended the `UndoManager` of java, in order to use it for tracking whether the graph has unsaved changes. The user can see whether there are unsaved changes, an asterisk * will appear next to the graph name in the window title. The user will also be prompted to save when closing the frame.
- **Extended file saving and loading:** As extensions were added to the graph, these also needed to be saved. Save-files can be read from two different formats: the standard format from the assignment requirements, as well as a custom format that includes more data, such as colors, text size, etc.

5 Process evaluation

The assignment was worked on for the full duration until the deadline. The initial basic requirements were completed within the first 2 days, and the rest of the time was spent implementing the extensions and bug fixing. Once we were happy with the program, this report was written.

We found the initial setup of the program, and all the necessary requirements very easy to achieve. The most difficult part of the assignment was implementing all the numerous extensions. Many bugs were discovered during development that resulted from the complex interactions between all of the components of the system.

The most troubling bug we discovered had to do with the order events are triggered in. When a node is selected, the text field used to set the text of the node gets updated. This causes the `valueChanged` event of the text field to fire - which in turn causes the text of all selected nodes to change.

From working on this assignment we mostly learned how to deal with complex, and large applications with many interconnected systems.

6 Conclusions

The McGraph graph editing program satisfies all of the requirements of the assignment, as goes well beyond the requirements with its numerous extensions. Due to the complexity of the program and the short time span in which we could work on it, the program likely has many bugs that were not discovered yet. However, we did not observe any buggy behavior in our preliminary tests, and all core features function as expected.

The code base for the program is reasonably maintainable we believe. Since this is the case, it would be nice to extend the program further in the future. One feature that we were meaning to add but couldn't due to time constraints would be looping edges. This would complete the functionality of the program in a sense, since this is the only limitation we currently place on the type of graphs that the editor supports. Another useful future extension would be to implement a proper, optimal graph coloring algorithm, as currently the graph coloring function of the solver uses a faulty heuristic which may result in sub-optimal coloring.

In conclusion, we are very happy with the state of the program, and we have learned a lot during its implementation.