Hanoi University of Science and Technology

School of Information and Communication Technology

PROJECT REPORT

Software Engineering

Visualization of Algorithms

Minimum Spanning Tree and Shortest Path in Graph Theory

Group1

Nguyễn Duy Tiến – 20194857

Trần Quang Hải – 20194755

Trần Lâm - 20194787

Table of Contents

1 Introduction

1.1 Purpose

This document provides a detailed description of the User Management Module, the user group, and their functions available at run time. Document describing the purpose and features of the system, interfaces, and constraints of the system to be implemented in response to external stimuli.

1.2 Scope

The purpose of the software is to create the user management module, the role of the user, and the functions that the user / user role can use at runtime.

This software is offline and open to everyone. Each time the user selects a function on the menu, the interface corresponding to that function will be displayed.

1.3 References

- Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
- SRS-UGMS-Sample-VN.doc, Nguyen Thi Thu Trang

1.4 How to run (from command line)

To get started, fire up your most trusted sh, bash and run:

```
$> java -jar GraphAlgoVisualizer.jar
```

If you do not have Java, time to install one!

If you are Window user, .exe is for you!

1.5 Source code

The source code and everything related for this project is provided at: https://github.com/RevolNoom/Visualizer/tree/master

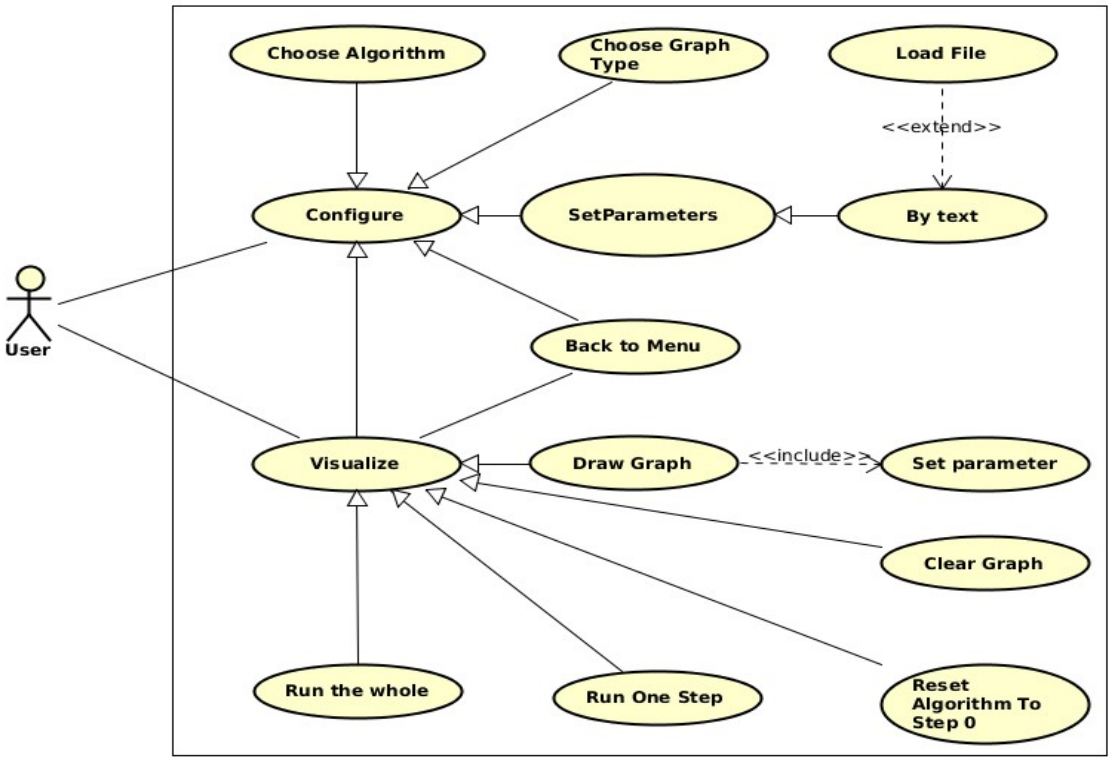
2 General Description

2.1 Agents

This software has only one agent, which is the user. They have full functionality to interact with software via the Graphical User Interface.

2.2 Use case overview diagram

The user can choose the algorithm that they want to be performed, setting the parameters needed and ask the software to visualize through each process of the algorithm. To set the parameters, user can tick the option they want, and input the data for the graph. Graph input can be loaded from a file, typed by hand. When the user is satisfied, they can continue to visualize their graph. After that, tinkering the graph is still available in the form of clicking the drawing board.

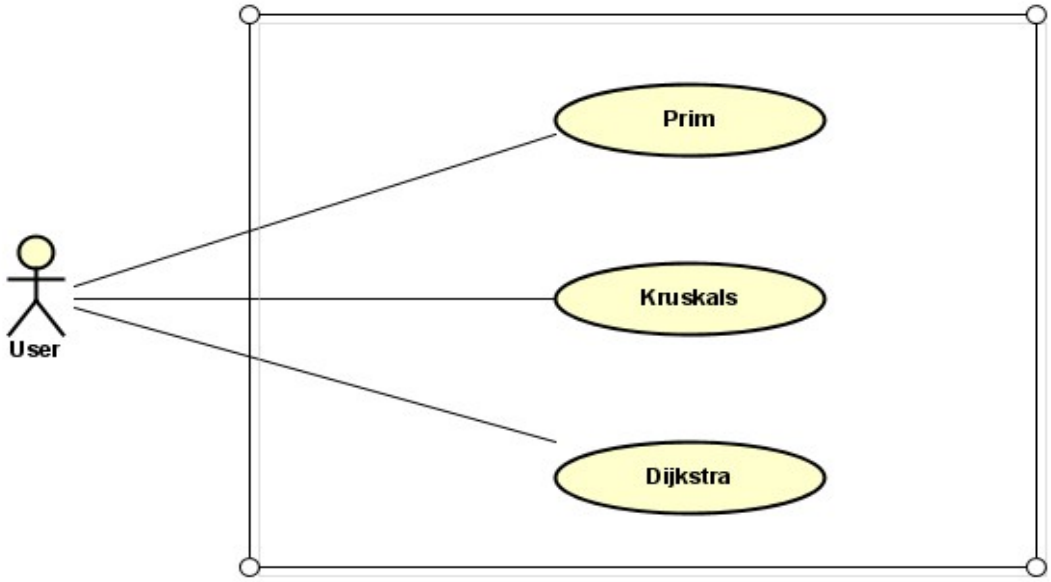


Finally, the fun part. The user will create their graph and run the algorithm with it. User can cancel the running process anytime and redraw their graph, or change the algorithm, graph type. The use cases in this general use case diagram are the composite

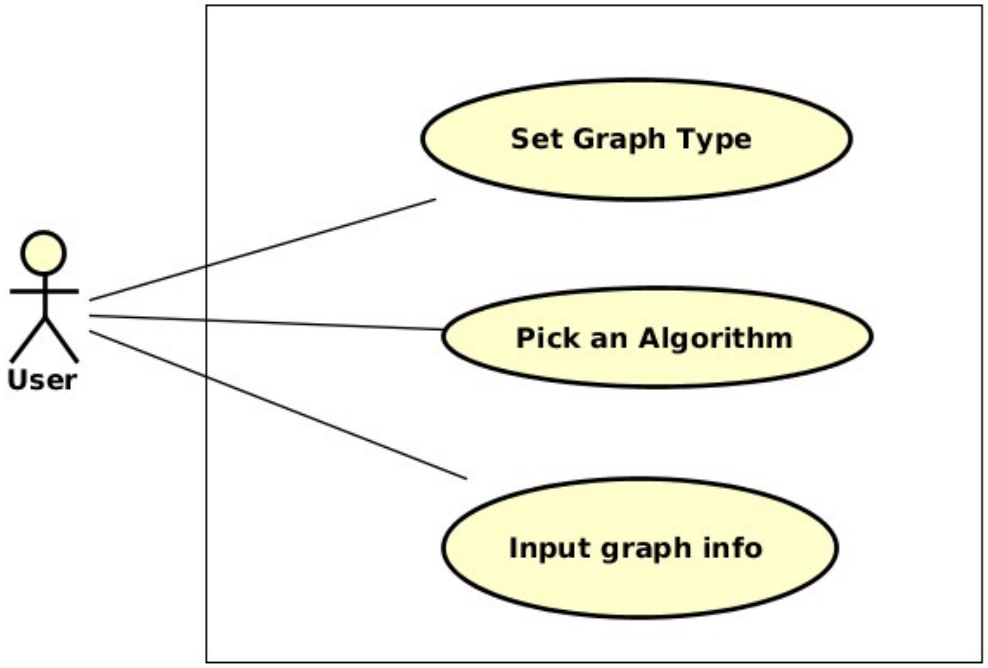
use cases of a group of use cases. Details of these use cases are given in the breakdown diagrams in the following section.

2.3 Use case decomposition diagram

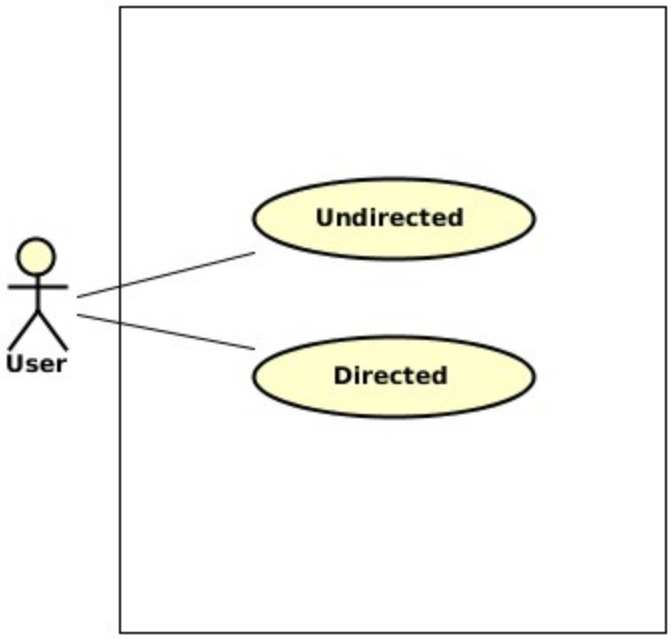
2.3.1 Use case “Choose Algorithm”



2.3.2 Use case “Set Parameters”



2.3.3 Use case “Choose Graph Type”



2.4 Usecase: Visualize

2.4.1. Draw Graph

To draw the graph, you can click on empty space on the board to create a vertex. Add an edge by choosing two vertices. When two vertices are chosen, a text box will pop up to ask you for the edge weight. If you accidentally choose an unwanted vertex, unselect it by right clicking on the board.

2.4.2 Clear Graph

To quickly discard the current graph and create a new one, you can click on the “Clear Graph” button. This will keep the option you chose concerning graph type and algorithm.

2.4.3 Back to menu

Clicking the button with a triangle on the upper-left corner of the screen will put you back to the input menu. In here, you can try experimenting with new algorithm and graph type.

2.4.4 Run one step

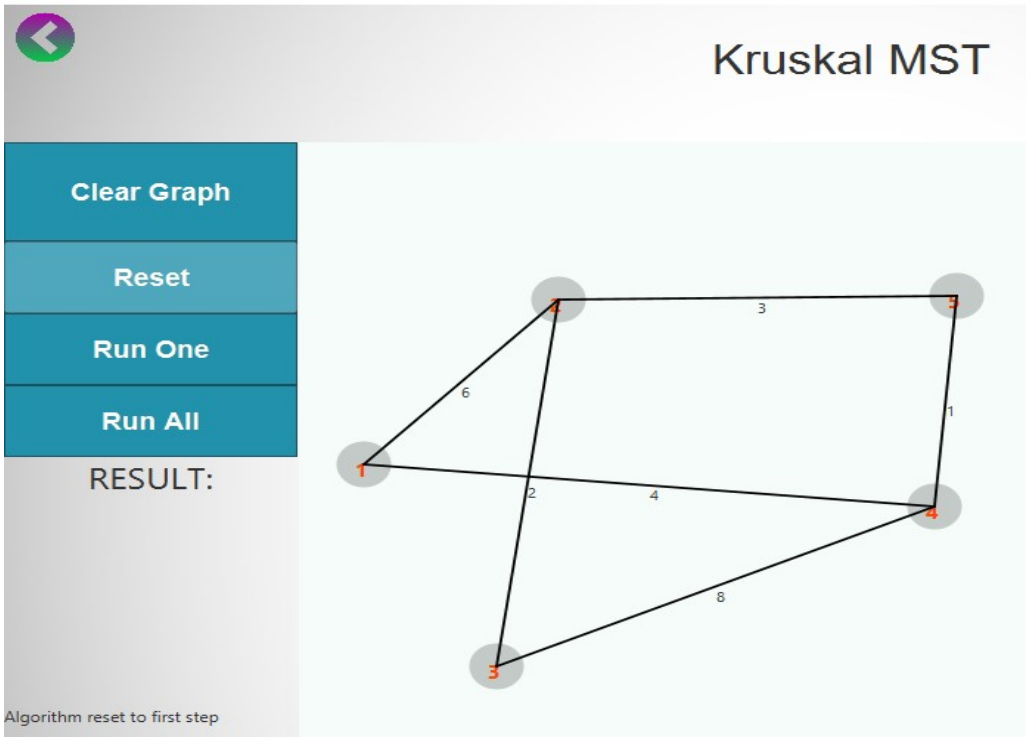
Clicking the “Run one” button will run one step of the algorithm. The graph is locked – You cannot make any change to the graph at this point. For simplicity, some algorithms will randomly choose the root and destination vertex for you. Edges and Vertices are highlighted, indicating the step taken in the algorithm.

2.4.5 Run the whole

Clicking the “Run all” button will run the algorithm until very end without stopping. The graph is locked, edges and vertices are highlighted, indicating the steps taken in the algorithm.

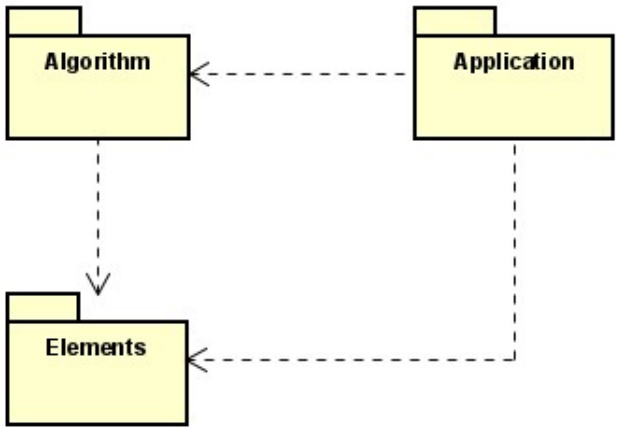
2.4.6 Reset Algorithm to step 0

The “Reset” button will refresh the graph back to the initial state when no algorithm step was taken. It also unlocks the graph – if it was previously locked. You can modify the graph as you see fit, and then run again when you choose to.



2.5 Package diagram

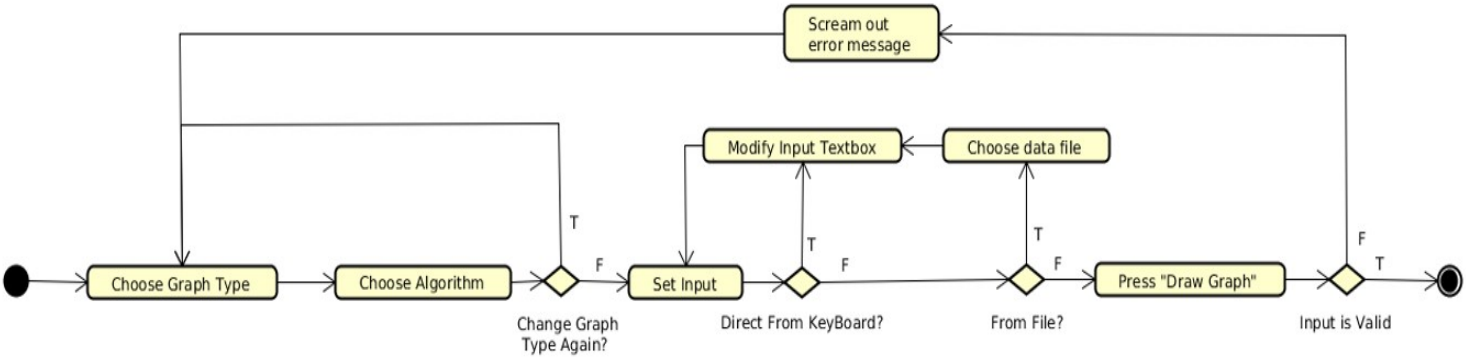
The package diagram showing the arrangement and organization of model elements in our project:



Two packages “Algorithm” and “Application” both depends on package “Elements” that contains basis elements of a graph. The package “Application” also depends on “Algorithm” to get how the algorithms work.

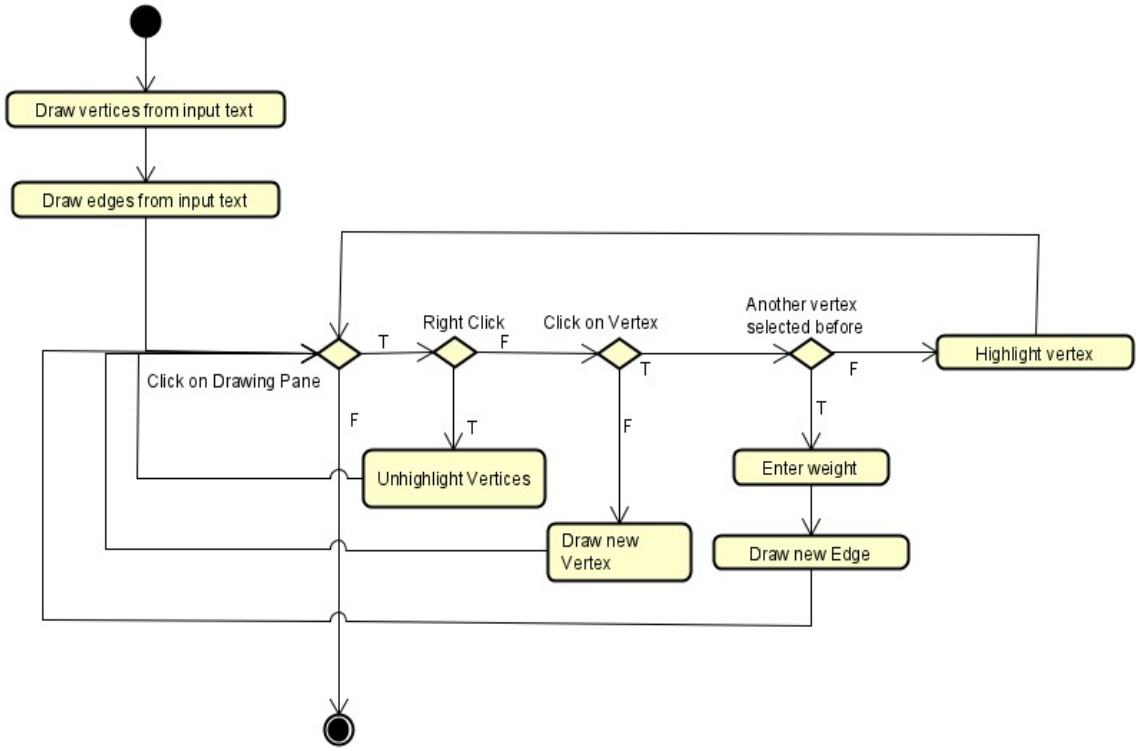
2.6 Activity Flow Diagram

2.6.1 Configuration



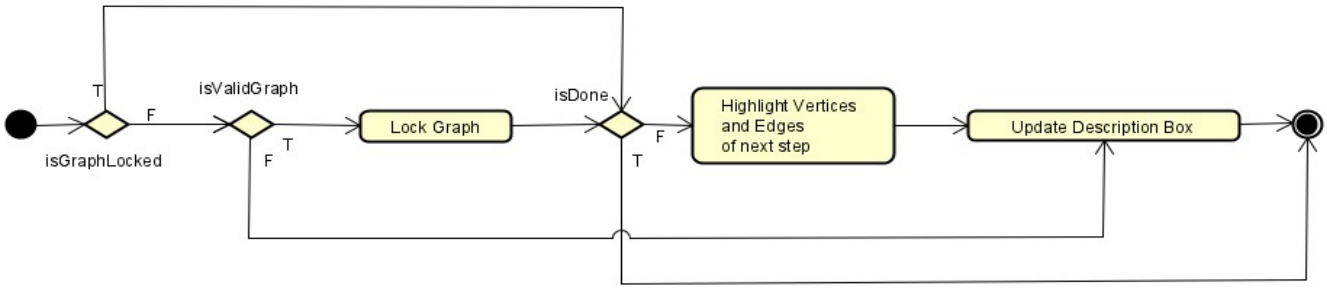
This is the initializatiton of our application. Here user will choose the type of graph and algorithm they want to visualize. They can also provide an input text (manual input or get from file) for the visualization.

2.6.2 Visualization: Drawing Graph



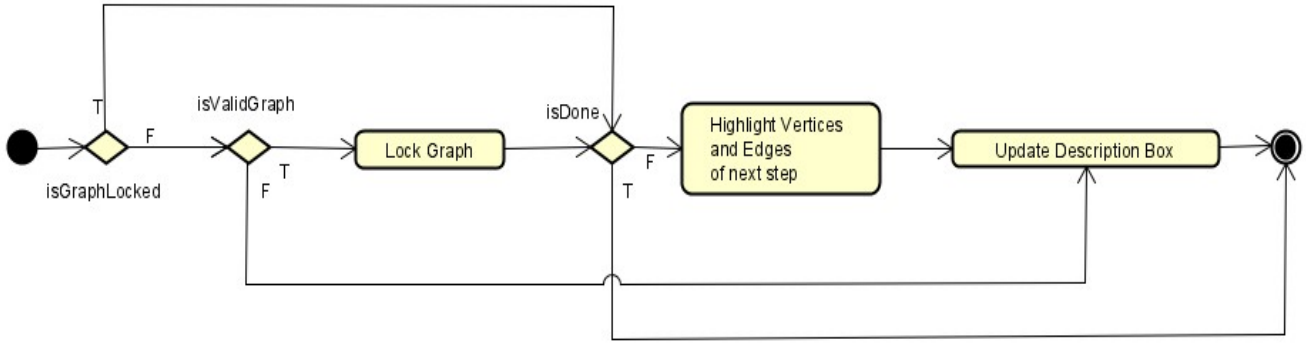
User can start drawing vertices, connecting them and set the weight to form edges.

2.6.3 Visualization: Run one algorithm step



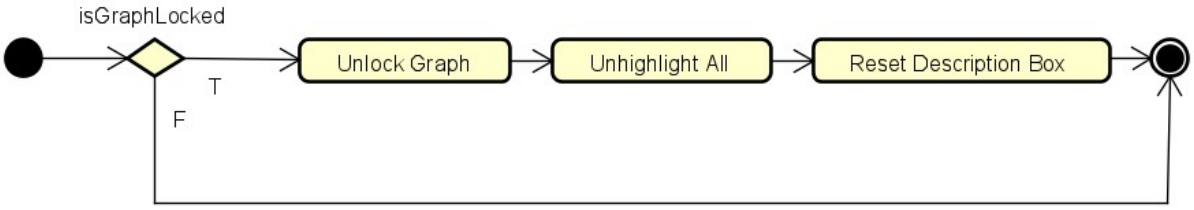
If the graph is valid, it will be locked from modification. Then the vertices and edges at that steps will be highlighted, and the box for algorithm description will be updated correspondingly.

2.6.4 Visualization: Run all algorithm steps



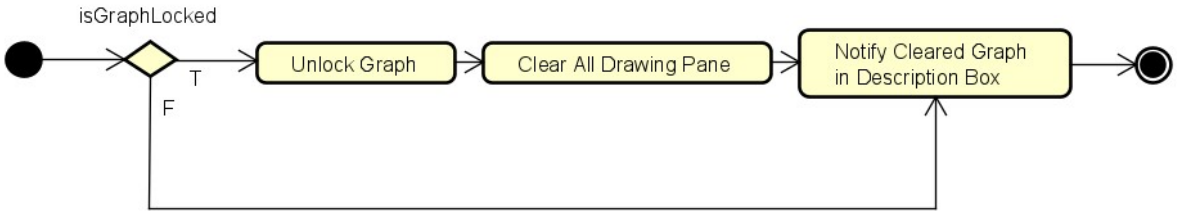
With the same representation as run one step, the run all algorithm will repeat the steps till the algorithm finishes.

2.6.5 Visualization: Reset algorithms steps



For the reset, every simulation will be set again to the time before the algorithm first run. The graph is also unlocked for other modification

2.6.6 Visualization: Clear graph



If users are willing to initiate with a completely new graph, they can choose the clear. Graph will be unlocked for drawing and the pane is set to a white blank board.

3 Specification of functions

3.1 Specification of use case UC001 "Configure"

<table>
<tr><td>Use case code</td><td>UC001</td><td>Use case name</td><td>Configure</td></tr>
<tr><td>Agent</td><td colspan="3">User</td></tr>
<tr><td>Prerequisite</td><td colspan="3">None</td></tr>
<tr><td>Default</td><td colspan="3">Graph Type: Undirected</td></tr>
<tr><td>Main flow of event (Success)</td><td colspan="3">
<table>
<tr><th>No.</th><th>Performed by</th><th>Action</th></tr>
<tr><td>1.</td><td>User</td><td>Tick the target graph type</td></tr>
<tr><td>2.</td><td>System</td><td>Change the applicable algorithms list.</td></tr>
<tr><td>3.</td><td>User</td><td>Choose an algorithm.</td></tr>
<tr><td>4.</td><td>System</td><td>Enable the user to continue to the next step</td></tr>
<tr><td>5.</td><td>User</td><td>Go the the next step: Graph drawing</td></tr>
</table>
</td></tr>
<tr><td>Alternate flow of event</td><td colspan="3">
<table>
<tr><th>No.</th><th>Performed by</th><th>Action</th></tr>
<tr><td>4a</td><td>User</td><td>Type input into the text box</td></tr>
<tr><td>4b</td><td>User</td><td>Choose a file to get data from</td></tr>
<tr><td>4c</td><td>User</td><td>Notify the user about incorrect graph</td></tr>
</table>
</td></tr>
<tr><td>Postrequisite</td><td colspan="3">Both graph type and Algorithm chosen</td></tr>
</table>

3.2 Specification of use case UC002 "Visualize"

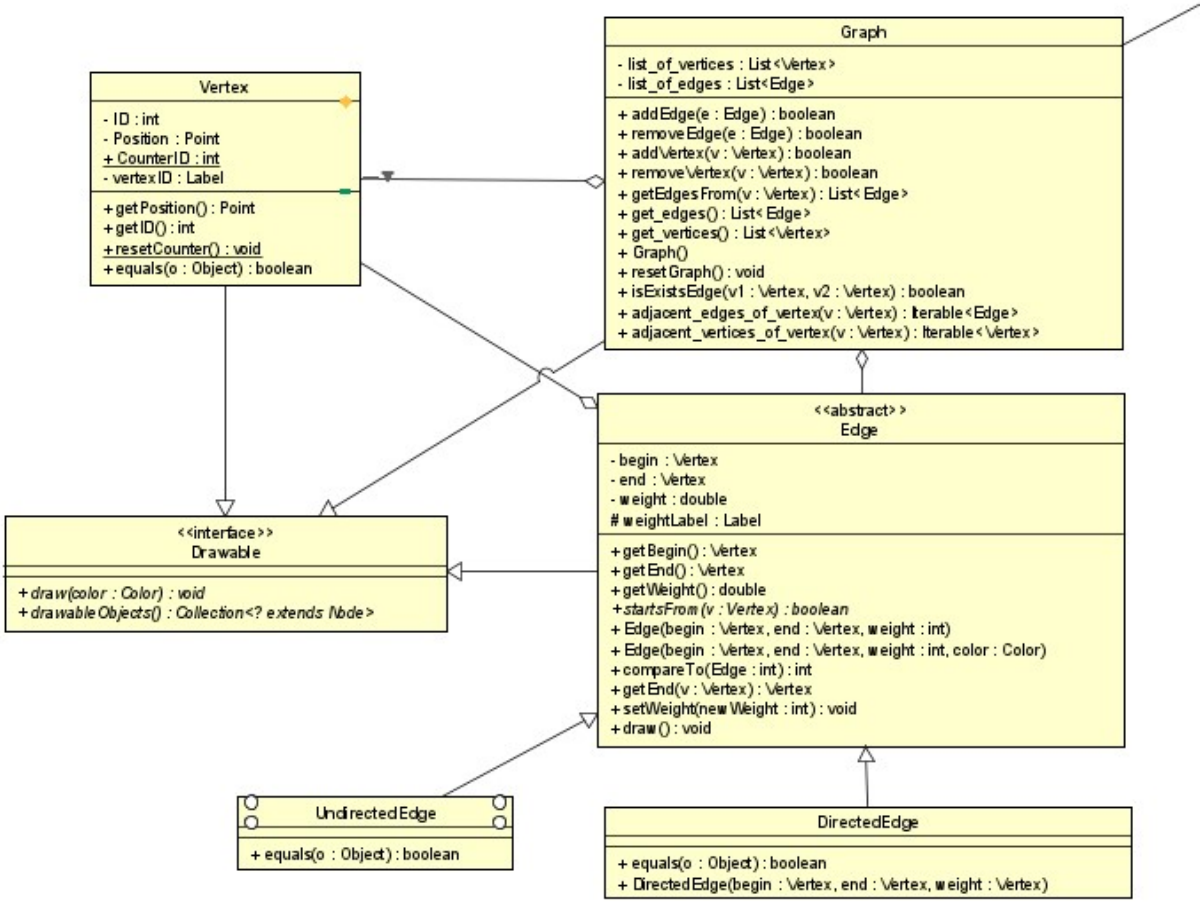
<table>
<tr><td>Use case code</td><td>UC002</td><td>Use case name</td><td>Visualize</td></tr>
<tr><td>Agent</td><td colspan="3">User</td></tr>
<tr><td>Prerequisite</td><td colspan="3">Input stage successfully completed</td></tr>
<tr><td>Main flow of event
(Success)</td><td colspan="3">
<table>
<tr><th>No.</th><th>Performed by</th><th>Action</th></tr>
<tr><td>1.</td><td>User</td><td>Draw the graph</td></tr>
<tr><td>2.</td><td>User</td><td>Run the algorithm</td></tr>
<tr><td>3.</td><td>System</td><td>Test for graph validity</td></tr>
<tr><td>4.</td><td>User</td><td>Keep running</td></tr>
<tr><td>5.</td><td>User</td><td>Reset the algorithm</td></tr>
<tr><td>6.</td><td>User</td><td>Run Again</td></tr>
<tr><td>7.</td><td>User</td><td>Go back to input menu because they are bored with the current algorithm</td></tr>
<tr><td>8.</td><td>User</td><td>Run the new algorithm</td></tr>
<tr><td>9.</td><td>User</td><td>Shutdown the application</td></tr>
</table>
</td></tr>
<tr><td>Alternate flow of event</td><td colspan="3">
<table>
<tr><th>No.</th><th>Performed by</th><th>Action</th></tr>
<tr><td>3a</td><td>System</td><td>Notify error: The algorithm might not be applicable for this graph (it may has many disconnected components, for example)</td></tr>
</table>
</td></tr>
<tr><td>Postrequisite</td><td colspan="3">No</td></tr>
</table>

4 Application Construction

The application is composed of 3 main groups of classes:

- Graphical User Interface (GUI): Defines how the user will interact with the application, what will be shown on the screen for the user to see.
- Graph: Defines a set of interfaces to draw a graph the screen, store the data about the graph.
- Algorithm: Defines a set of interfaces to render the graph and illustrate the algorithm.

4.1 Graph



To get started, because we want to draw something, we create an interface named Drawable. It's simple as that. The method draw() will tell an Object to draw itself, however it wants to. And drawableObjects() asks the object for all the "drawable" things it's responsible for. This is necessary, the reason why will be explained in the upcoming classes.

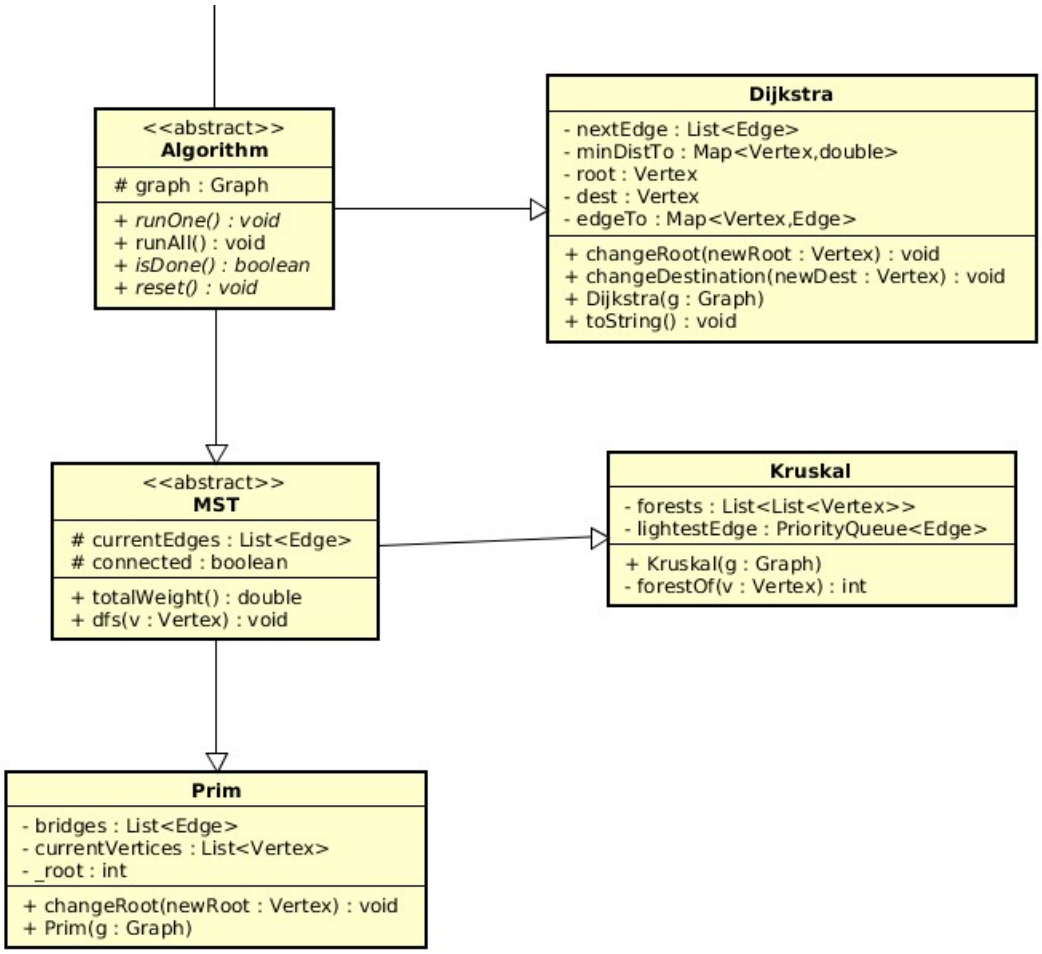
All the Drawables in this application are Graph, Vertex, Edge. A Vertex is a coloured circle. An Edge is a line that connects two "coloured circles". Vertex and Edge each also has a label to tag their information. For Vertex, it's tagged with its ID (a number), and also a number for Edge, but to represent weight of the Edge. Because we want to stick the tag to the edges and vertices, it's the job of the developers (us) to notify the GUI that the vertices and edges also have tags. DrawableObjects() does this.

The graph is a composition of Edges and Vertices. Because the graph is supposed to work with Algorithm group, we define a (bit silly) interface for the Algorithm to work with. The target of this application is to illustrate the way an algorithm work, not solving a problem, and thus, we do not litter our Graph class with many extra-information-that-has-nothing-to-do-with-OOP (like a 2-D matrix representing the neighbours of a vertex). Performance sacrifices were made, and we gain simple objects construction in turn.

There are two types of Edges: Directed and Un-. This leaves the ground for parameterizing Graph class into a generic. But no, we did not do that. Although graph type distinction is important for an Algorithm to run, we leave it to Algorithm and GUI to handle this complication.

4.2 Algorithm

The source of many troubles, Algorithm remembers by heart the graph which is responsible for rendering. Algorithm defines a set of interface to render a graph, like runOne() to show the user what's it like after one algorithm step. Or if you're impatient, runAll() of them (although this is not encouraged, our graph visualizer is designed for visualization, not problem solving). An Algorithm should knows when to stop, which is tested by isDone(). Redoing the algorithm might come in handy, so reset() does just that. We do not support tracing back one steps at a time, because this is a mini-project, not a milk cow.



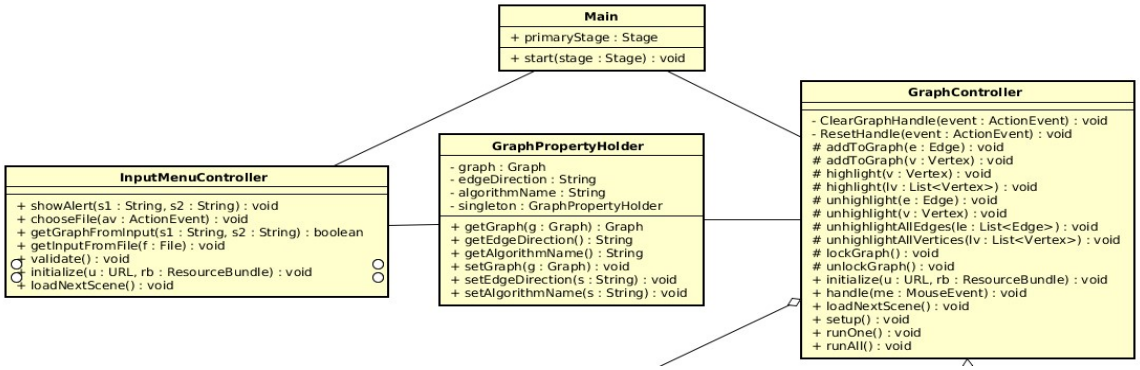
Further up the inheritance tree are concrete Algorithms with specified steps to run. We extend Prim and Kruskal from MST class (Minimum Spanning Tree) because they solve the same class of problems. Actually, that looks OOP, but no interesting optimization takes place. Dijkstra solves another class of problems, and thus it makes sense to not extend it from MST.

Each of them needs to store different information, so they have different member variables. But looking from outside, you will not see much difference between them. Who cares, just runOne() or runAll(), all other details are minor.

The algorithm is supposed to work with different graph types, so we have tried to not duplicate classes by implementing them as compatible to both graph as possible. It comes with a downside: We don’t see the bugs. Not immediately.

4.3 GUI

The Main class's responsible for Scenes transition. Though named Main, it's the one and only minor actor in the whole program.



Basically, there are 2 scenes in our applications:

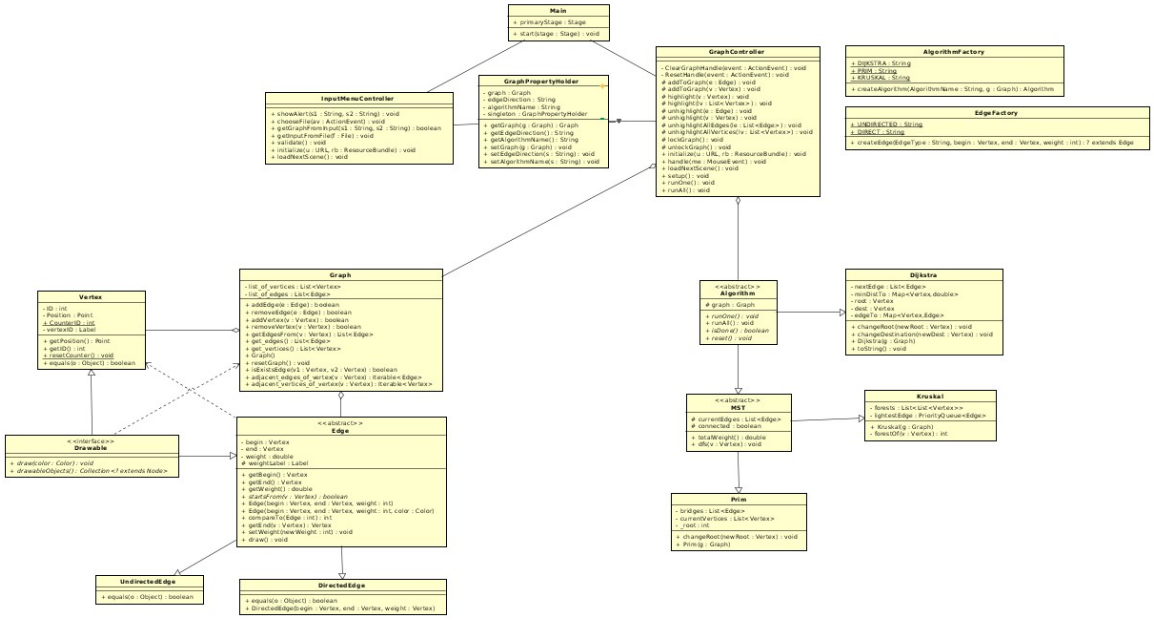
- The Menu Scene: You give your demand about algorithm, graph type and, optionally, a stored text file saving a graph. What happens here is monitored by InputMenuController class.

- The Artist Scene: You draw your graph as you like. Another vertex here, a sharp edge there,… Doing art has never felt so free! Jokes aside, you also run the algorithm in this scene, with some less artistic-looking buttons. This Scene is controlled by GraphControllerClass

Because what information is taken at InputMenuController must be forwarded to GraphController, we need a mechanism to transfer them. I personally do Not like the idea of public static variables to let the Controllers to throw objects back and forth, the GraphPropertyHolder is born for this purpose. It acts as the intermediate ground for two classes, preventing them from obnoxiously harming each other without anyone knowing. Also, it's safer. The GraphController can copy the data from Holder and then hide under its coat, hissing at anyone who dares try to modify its data.

4.4 Conclude

Above is a snapshot of our full diagram:



Don’t worry if you can read nothing clearly, the diagram should be attached with the application. There’re like two more classes. Factories. Used to create Edges and Algorithms. They are not important. Because they are not important, neither useful they are. Good for shortening some lines of codes, but not interesting enough to put in this document. This ends the Software Requirement Specification document. Thank you for reading.