# Interactive Graph Drawing

**A Drawing Software**

Bachelor Thesis
Faculty of Science, University of Bern

submitted by

**Dominik Fischli**

from Bern, Switzerland

Supervision:

PD Dr. Kaspar Riesen
Matthias Fuchs
Pattern Recognition Group
Institute of Computer Science (INF)
University of Bern, Switzerland

## Abstract

As graph datasets become larger and more complex, it becomes increasingly useful to visualise them. They often lack positional data attributes and require the researcher to decide how to lay out a graph. One solution to this problem is graph layout algorithms. Implementing such an algorithm and all the adjustments required for a graph take time away from the researcher, who may prefer to study the results of the graph layouts rather than how to create them. This work attempts to address this problem by providing an application that allows researchers to quickly load, adapt and lay out graphs. It implements several force-directed graph layout algorithms and focuses on making the application as intuitive as possible, taking into account design principles from human-computer interaction. It examines the results of the application in terms of various aesthetic criteria and provides ideas for further improvement of the application through the addition of other graph layout algorithms and functionality.

# Acknowledgements

I would like to express my gratitude to my supervisor PD. Dr. Kaspar Riesen for his support of this project and Matthias Fuchs, for his many inputs and helpful advice. Without their help and support, this project would not have been feasible.

# Contents

# Chapter 1

# Introduction

Graphs are relational models made up of edges and nodes. They are commonly used to describe networks where nodes can be connected by edges. Graphs can be used to describe a variety of systems and are used in a wide range of fields, including software engineering, information systems and security, social sciences, biology and geography. Examples of their use include describing networks of routers and their connections in computer science, and molecular structures with atoms and bonds in chemistry. In both cases, visualisation can help researchers understand their properties.

Graphs do not always come with positional data for their nodes. For large graphs, manually deciding on appropriate node positions becomes increasingly time-consuming. This motivates the development of automatic layout algorithms that generate positional data for a given graph according to some aesthetic criteria.

An early layout algorithm is Tutte's 1963 algorithm, which uses barycentric coordinate systems [1]. A more general technique for generating positional data is force-directed graph drawing. The core idea of force-directed graph drawing is to replace nodes and edges with pseudo-physical components and forces, e. g. nodes become metal rings and edges become springs. Typically, forces and physical constraints are approximated and simplified according to the speed and aesthetic constraints of an algorithm.

In 1984, Eades created the first force-directed graph drawing algorithm based on the simulation of spring forces [2], which in turn inspired other spring-based algorithms such as Kamada-Kawai's in 1989 [3] and Fruchterman-Reingold's in 1991 [4]. To ensure a pleasing layout, the algorithms attempt to optimise for aesthetic criteria such as even distribution of nodes within a frame, minimisation of edge crossing, uniform edge length, and representation of any inherent symmetry in the graph.

One problem researchers face when trying to take advantage of these algorithms is that the applications that implement them either lack the ability to customise the drawing, or require application-specific knowledge and often lack a graphical user interface. Researchers often need to become familiar with the specifics of the application before they can take advantage of the power of automatic layout algorithms.

This work aims to fill this gap with software that is easy to use and satisfies the need for customisation of graphs. It focuses on undirected graphs, which are supported by force-directed layout algorithms, and aims to allow a researcher to easily customise each node and edge.

To ensure that the software is easy to use, principles from the field of human-computer interaction are used to guide the development of the software. The principles cover aspects of user interface and user experience design. They advise decisions about the aesthetic look and feel of the graphical interface and the interaction flow.

The rest of the thesis is structured as follows. Chapter 2 introduces the definitions and algorithms used in this thesis. Chapter 3 explains the implementation, including the frameworks used and the widgets created for the software. Chapter 4 presents the performance of the implemented algorithm on a set of graphs and compares them according to aesthetic criteria defined in Chapter 2. Chapter 5 will give a brief overview of the work done and what further improvements could be made to the software.

# Chapter 2

# Theoretical Foundation

This chapter provides some technical definitions and describes the algorithm used by the product. It also provides some insight into the HCI principles used to design the product.

The following definitions of graphs closely follow K.Riesens book on Structural Pattern Recognition with Graph Edit Distance [5].

**Definition 2.0.1** (Graph)**.** Let $L_V$ and $L_E$ be finite or infinite label sets for nodes and edges, respectively. A **graph** is a four-tuple $g = (V, E, \mu, \vee)$, where

- $V$ is the finite **set of nodes**,
- $E \subseteq V \times V$ is the **set of edges**,
- $\mu : V \to L_V$ is the **node labelling function**,
- $\nu : E \to L_E$ is the **edge labelling function**.

The **size of g** is defined as the number of nodes in the graph. More formally

$$n = \sum_{v \in V} 1 = |V|$$

An **unlabelled** graph is a graph where all labels are equal to the **empty label** $\epsilon$:

$$\mu(v) = \epsilon, \quad \forall v \in V$$

$$\nu(e) = \epsilon, \quad \forall e \in E$$

An example of a unlabelled graph is shown in figure 2.1.

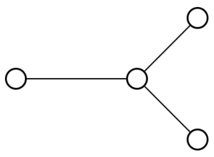


Figure 2.1: Unlabelled graph

## 2.1 Aesthetics

Aesthetics describe the criteria that graph layout algorithms optimize for. In this work, we will use the following generally-accepted criteria [6]:

- **Even distribution of nodes**. Nodes should use the available space as much as possible. A more even distribution will result in a less cluttered visual.

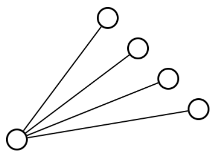


Figure 2.2: Bad distribution

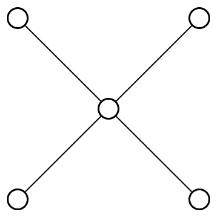


Figure 2.3: Good distribution

- **Minimisation of edge crossings.** Edge crossings should be avoided whenever possible because they usually confuse and add to clutter.

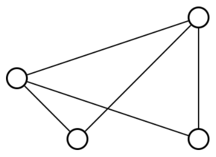


Figure 2.4: Edge crossings present

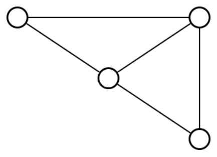


Figure 2.5: Edge crossings avoided

- **Uniform edge length.** Improves the readability of the graph.

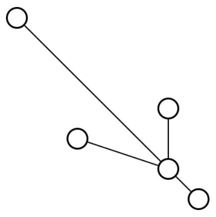


Figure 2.6: Non-uniform edge lengths

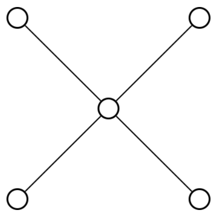


Figure 2.7: Uniform edge lengths

- **Maximise symmetry.** Symmetrical drawings are very recognisable. The inherent symmetry of a graph should therefore be represented whenever possible.

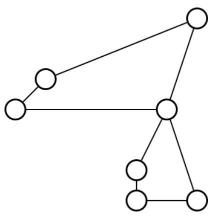


Figure 2.8: Non symmetric drawing

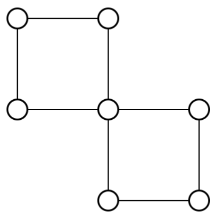


Figure 2.9: Symmetric drawing

## 2.2 Layout Algorithms

This section discusses the theory behind the layout algorithms implemented in the product. The aesthetic criteria for which the algorithms are optimised are explained. In total, four algorithms are implemented. The random and circle algorithms are basic algorithms that determine the layout of a graph in a single iteration. Fruchterman-Reingold is a force directed algorithm and simulates an attractive and a repulsive force to produce an intuitive layout of given graphs. Multiple iterations are required to achieve a good layout, and the forces acting on each node are recalculated with each iteration. Kamada-Kawai is similar to Fruchterman-Reingold, but simulates only one attractive force between each pair of nodes and tries to optimise the resulting system for total energy.

**Random layout.** The random layout (see figure 2.10) calculates a random position on the given topology, usually a rectangle, for each node. It can be useful to compute an initial layout or to reset all positions if the converged layout does not meet expectations. An example of such a situation is when a node doesn't manage to cross on the other side of an edge, which may correspond to a local minimum in the optimisation process.

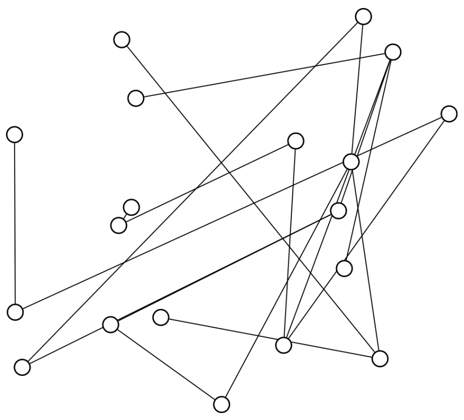


Figure 2.10: Random layout

**Circle layout.** The circle layout (see figure 2.11) assigns each node a corner position on an $n$-polygon with a given radius. Like the random algorithm, the circle algorithm can be used to generate an initial layout to pass to other layout algorithms.

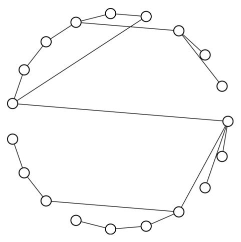


Figure 2.11: Circle layout

**Fruchterman-Reingold.** Fruchterman-Reingold's [4] algorithm (see figure 2.12) extends the ideas developed by Eades [2] and generates an iterative force-directed layout of nodes. Unlike real physical forces, the algorithm does not calculate the acceleration but the displacement of the nodes to avoid oscillation. The displacement is calculated based on an attractive and a repulsive force. For each pair of nodes with an edge between them, the attractive force is used. For all other pairs, the repulsive force is used. The forces on a node are then summed and used to calculate the new position of the node. The layouts tend to reflect the inherent symmetry of a graph and tend to distribute nodes evenly across a given frame. The algorithm is also fast and simple. Modifications can be made by changing the attractive force $f_a$ and the repulsive force $f_r$, or by influencing the edge length constant. $f_a$ and $f_r$ must accept a distance value. The edge length constant can be changed indirectly by setting the constant $C$, which is multiplied by the edge length constant.

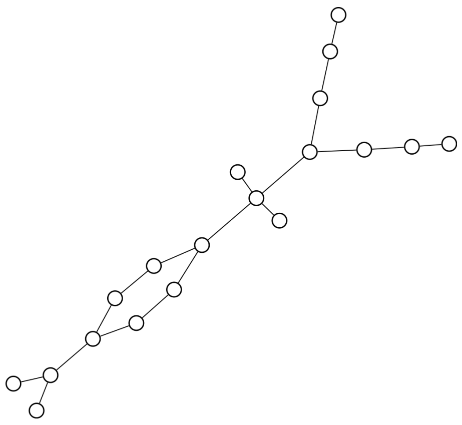


Figure 2.12: Fruchterman-Reingold layout

**Kamada-Kawai.** Kamada-Kawai's [3] algorithm (see figure 2.13) introduces the idea of a graph-theoretic distance, which states that the Euclidean distance between node positions should be similar to the number of edges on the shortest path between two nodes. Conceptually, the nodes are replaced by rings, and for each pair of nodes a spring is placed between their rings and given a length corresponding to their graph-theoretic distance. The optimal edge length is then the corresponding spring length for each edge. All the forces in this spring system are then summed and the resulting value is called the total energy. The goal of the algorithm is to minimise this total energy. The resulting algorithm is very good at generating graphs with uniform edge lengths and at generating layouts for symmetric graphs. In addition, the algorithm accepts edge weights that correspond directly to edge lengths. Larger weights increase the spring length corresponding to an edge.

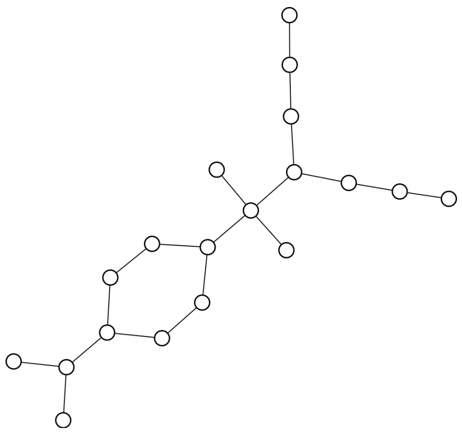


Figure 2.13: Kamada-Kawai layout

## 2.3 Human-Computer Interaction

A non-functional goal of the product is to be easy and intuitive to use. The theory of human-computer interaction (HCI) provides the basis for principled interaction design and the definitions that describe interfaces. The definitions and guidelines given in this chapter are based on K. Riesen's lecture script on HCI [7].

**WIMP and Widgets** WIMP stands for windows, icons, menus and pointers. It describes the kind of interfaces that rely on direct feedback given by the system to the user. WIMP software implementations often rely on widgets. Widgets are standardised parts of a user interface, e. g. a clickable button or a sliding window.

**Design Principles** The following design principles are applicable to all software. They have been consolidated by K. Riesen [7] from multiple sources.

- **Learnable.** New users should have an easy time learning how to use the product without a need for instructions in the form of tutorials. The less documentation and help an user needs to be able to use the full extent of features, the better.
- **Memorable.** Users who haven't used the product in some time should not have to relearn everything. This can be aided by allowing users to recognize the functionality of the interface by labelling and signifying relevant objects. The user should not have to try to remember how the processes work.
- **Efficient.** Advanced users will want to be fast and feel competent when working with the software. Frequent actions should therefore require as little input as possible and users should be able to customize their experience.
- **Flexible.** Adaptability: The user should be able to adapt the interface to his needs, for example by defining the position of graphical interfaces.
  Adaptivity: The system recognizes the needs of the user and adapts itself accordingly.
- **Consistent.** Similar situations in similar environments should require and allow similar actions. This can be internally similar or similar with other products.
- **Visible.** The current internal status should always be known to the user. There should be visual feedback indicating changes in the status and indicating background tasks. The bigger and rarer the action, the more important the feedback.

- **Controllable.** Users should be in control of the product and be able to define the state of the system as they wish. It should be possible to interrupt an action in order to work on a more urgent problem and to resume the action later.
- **Fault Tolerant.** Mistakes should be either impossible via constraints or easy to undo. Wrong inputs should not break the program but rather inform the user on what went wrong and how to correct it.
- **Uncomplicated.** It should not be expected of users to memorize multiple things at once in order to complete a task. The mental load expected on users should generally be minimized. To achieve this, users should be able to recognize based on the graphical interface how to process a task.

# Chapter 3

# Software Architecture

This chapter explains the design process for the software in section 3.1, the frameworks chosen for development in section 3.2, and the features and interfaces developed in section 3.3. The chosen programming language is `C++`. A detailed explanation of the code is omitted. The software and its source code are available on GitHub [8].

## 3.1 Design

To start the development process, mock-ups were created as shown in figure 3.1. The centre of the application can be used to draw graphs, while the right side can be used to set the appearance of the graph. Graph layout algorithms, here written as 'plotting algorithms', should be directly accessible. There is a menu bar at the top which provides access to additional operations such as loading and saving graphs. The second image shows how two graphs could be displayed side by side without the need for additional windows. Care should be taken to use signifiers such as colour boxes and other visual indicators to show the user that the appearance can be changed in the 'Appearance Settings' box.

These decisions have been influenced by the HCI principles defined in section 2.3. A key feature of the software is to allow the user to draw diagrams in a straightforward way and to learn how to use the software by experimenting.

Consistency is achieved by taking inspiration from other drawing applications. User interface design for buttons and labels is delegated to the Graphical User Interface (GUI) framework described in section 3.2.

From a software development perspective, the source code is divided into two distinct parts. The first part deals with the business logic and defines the internal representation of graphs, nodes, edges and more. The second part defines the graphical user interface, taking advantage of the GUI framework and depending on

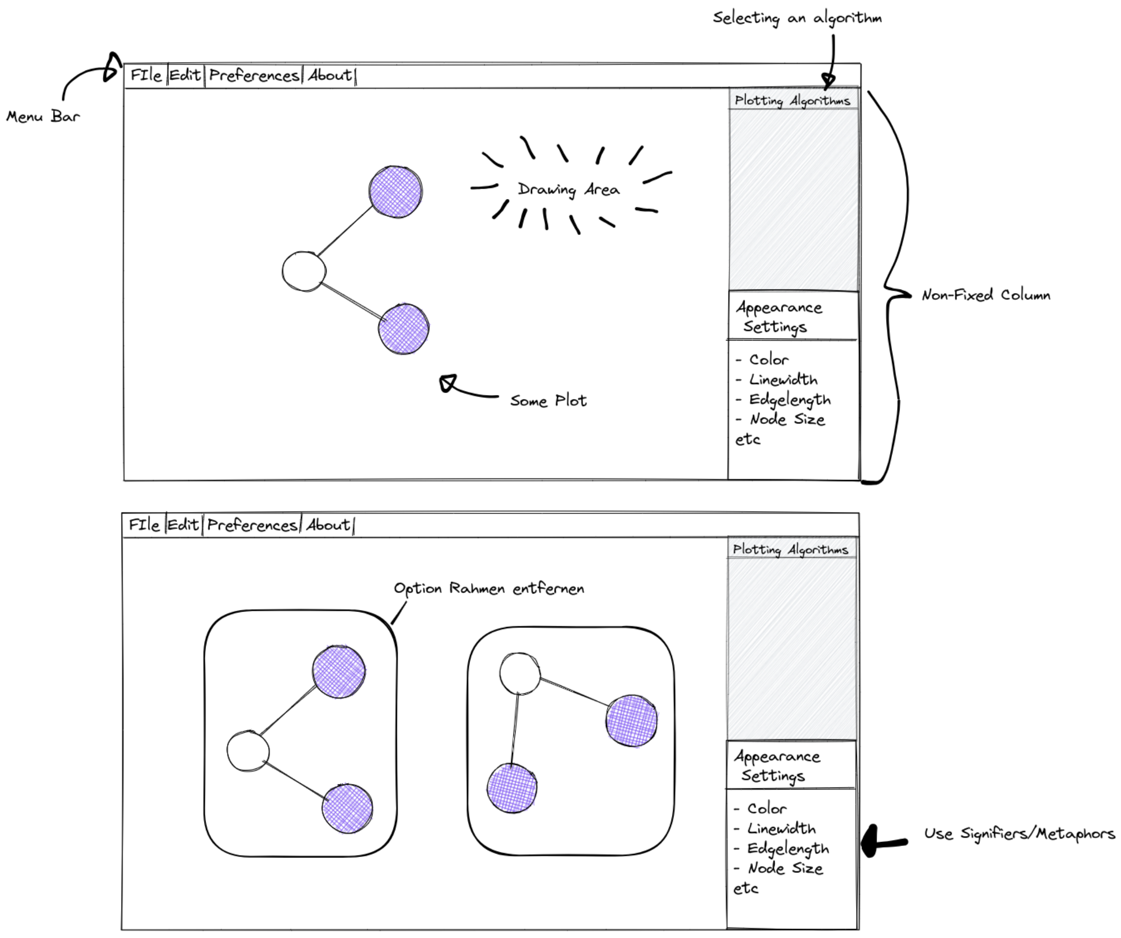


Figure 3.1: Mock-ups

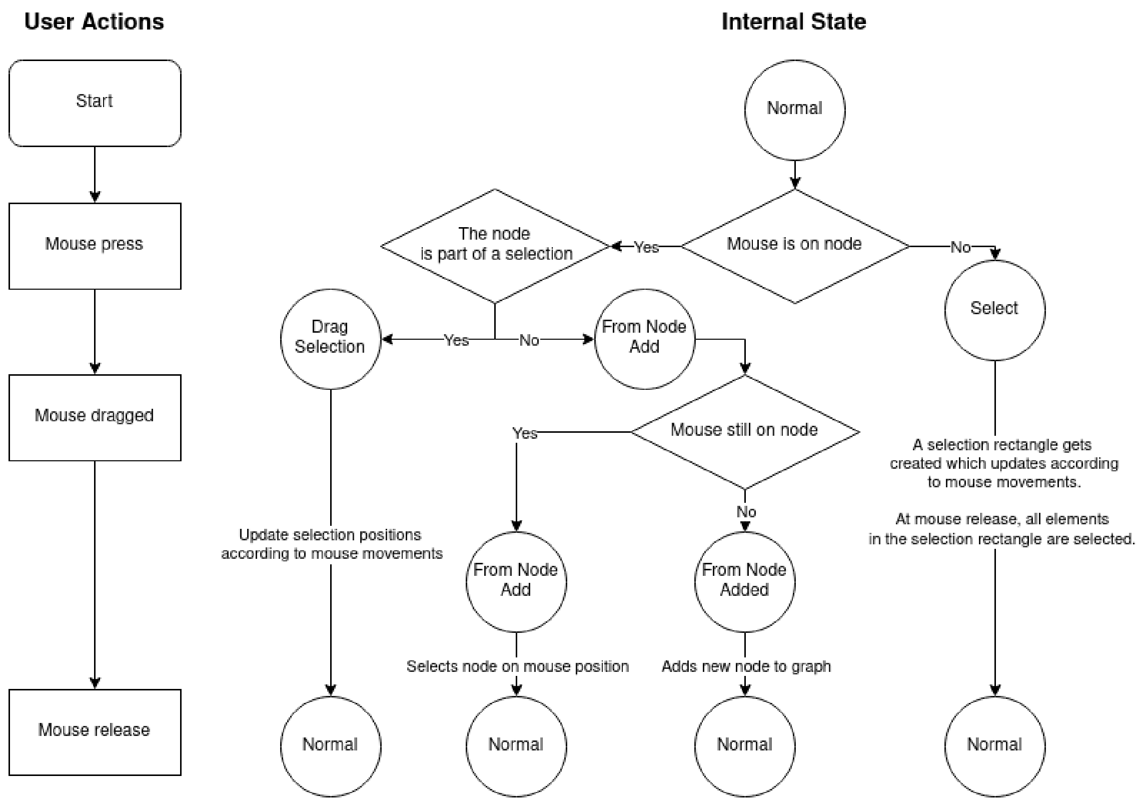

User Actions
Internal State
Start
Mouse press
Mouse dragged
Mouse release
Normal
Mouse is on node
Yes
No
The node is part of a selection
Select
Drag Selection
Yes
No
From Node Add
Mouse still on node
Yes
No
A selection rectangle gets created which updates according to mouse movements.
At mouse release, all elements in the selection rectangle are selected.
Update selection positions according to mouse movements
From Node Add
From Node Added
Selects node on mouse position
Adds new node to graph
Normal
Normal
Normal
Normal

Figure 3.2: Interaction flow with initial normal state

the first part. To ensure smooth interaction flows, the state design pattern was implemented. A total of six states have been implemented. The states handle the software flow during mouse actions meant to draw graphs.

Two such states are 'Normal' and 'Add'. They are two of the most used states, as 'Normal' is the state that all states fall back to when the user releases the mouse, except for 'Add'. When the user adds nodes using the 'Add' state, staying in the 'Add' state allows the user to add several consecutive nodes. As these two states are the most involved in the software, figure 3.2 and figure 3.3 show the reader how these states process the user's input.

## 3.2 Technologies

The programming language used is `C++`, as it is a mature and efficient object-oriented language with a large and active community maintaining it. The software uses two frameworks for development, boost [9] and gtkmm [10]. Boost is meant to be an extension to the standard library. It includes the boost graph library, which implements the layout algorithms used in the product and provides a flexible implementation of graphs. The second framework, gtkmm, is a free and open source widget framework based on GTK, which is the default widget library for GNOME.

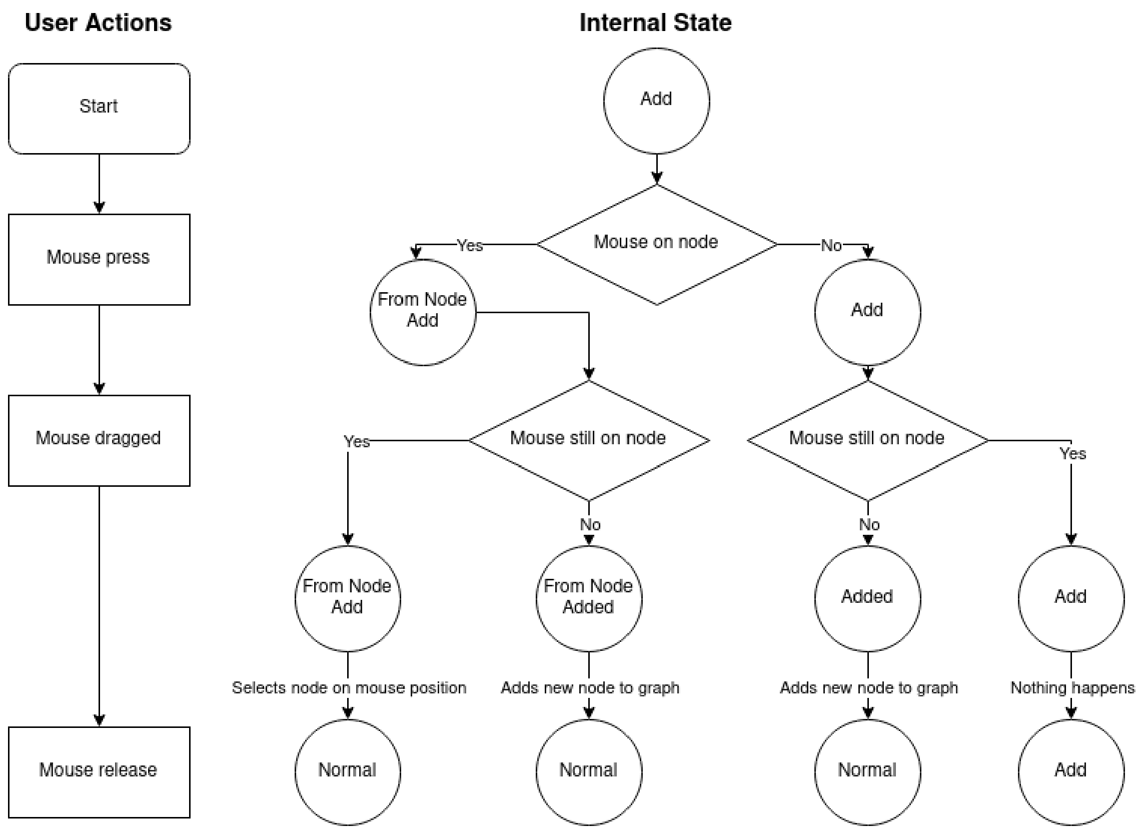


Figure 3.3: Interaction flow with initial add state

**Boost.** The aim of the boost framework is to create libraries that are standardisation-ready, meaning that they aim to include boost libraries in the standard `C++` libraries when they are mature enough. The libraries are well-tested and usable.

The Boost Graph Library is a subset of the Boost libraries that focuses on graph algorithms and concepts. The library is flexible in terms of data structures, i. e. it is possible to choose whether graphs are represented as adjacency lists or matrices. The algorithms can be extended by visitors.

**Gtkmm.** Gtkmm is the wrapper around the C widget library GTK. It contains a variety of widgets that are extensible through inheritance, a feature of the object orientation of `C++`. This made gtkmm a good choice to work with.

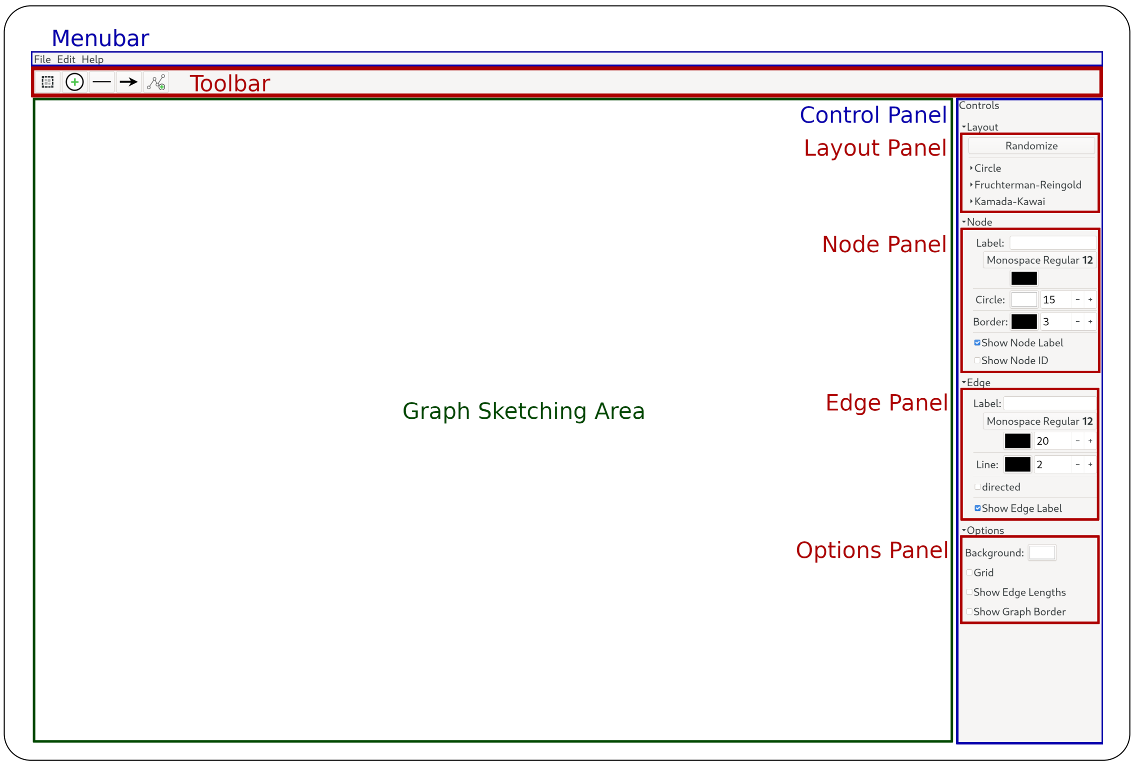
Because the base of gtkmm, GTK [11], is a widely used framework that adheres to industry standards and has an active community developing and maintaining it, the widgets provided by gtkmm are consistent with a variety of other products. Much of the interface design is left to gtkmm, with the notable exception of the graph drawings and interactions.

## 3.3 Features

This section describes the features implemented and the design choices made. In general, widgets from the gtkmm library have been used. Since gtkmm adheres to industry standards, this has the advantage of a clean and recognisable user interface. The widgets constrain the user to specific actions, so incorrect input is not possible. Advanced users will know how to navigate the widgets with as little input as possible, as gtkmm allows the user to tab between widgets. If a Linux desktop environment is based on GTK, the software will recognise the active theme and adapt the base widgets accordingly. All available functions can be applied by drawing and interacting with the menubar, toolbar and control panel.

**Main View.** This is the view the user is presented with after starting the software, as shown in figure 3.4. The user can start to create graphs by drawing on the Graph Sketching Area in the centre of the screen or load a graph via the menubar. The menubar also allows the user to save or export the graph.

Figure 3.4: Main View of the product



Label:
Monospace Regular 12
Circle: 15
Border: 3
☑ Show Node Label
☐ Show Node ID

Label:
Monospace Regular 12
20
Line: 2
☐ directed
☑ Show Edge Label

Background:
☐ Grid
☐ Show Edge Lengths
☐ Show Graph Border

To the right, the control panel is visible. The design choice here is to have a structure of toolbar on top, main area centre and controls on the right. This allows the software to use most of the available space for drawing while showing the user

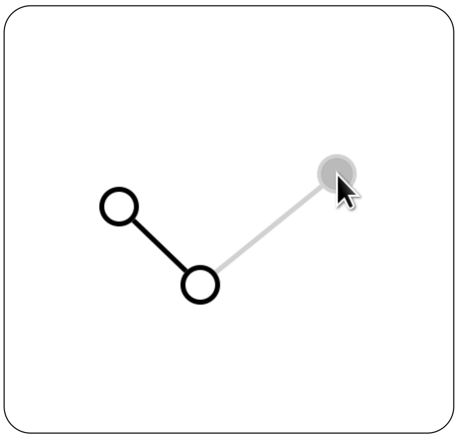


Figure 3.5: New nodes and their connecting edges are greyed out

the current properties on the right hand side. This structure can be seen in other drawing tools such as draw.io.

**Menubar.** This widget offers the user to load or save `graphml` files. Additionally, it offers to export the graph to one of the following file formats: `png`, `pdf`, `svg`.

**Graph Sketching Area.** This widget allows the user to interactively draw graphs by adding, moving and deleting nodes. An example of a user in the middle of adding a new node connected to another node is shown in figure 3.5. The following actions are supported by the widget:

- Click on a node and drag to create a new node connected to the node originally clicked on.
- Connect two nodes by dragging the mouse from one unselected node and releasing it on top of the other node.
- Click and drag over nodes and edges to create a selection. This action is consistent with the behaviour in other editors.
- Click and drag a selection to move nodes. This is consistent with clicking and dragging a selection in word or other editors.
- Zoom the area by pressing `ctrl` and scrolling horizontally.
- Pan the area by scrolling in vertical or horizontal direction.

In general, the actions are supposed to be intuitive to the user to ease the learning process and make working with the tool fun.

While there is an internal state of the application that changes with the users actions, the current state is made visible to the user by the natural constraints of an action. Clicking, keeping the mouse pressed and then releasing it constitutes as a single user action but can let the area run through multiple internal states which define whether a new node should be created or a selection moved and so on.

Newly created nodes are being greyed out until the user releases the mouse button as shown in figure 3.5. This informs the user of the temporary state of this node. Moving the mouse while pressed will show the user that he controls the positioning of this new node.

Creating a selection rectangle is simple. The user can select the selection icon from the toolbar and then drag on the sketching area, staying consistent with prior knowledge. Releasing the mouse selects all objects within the selection rectangle. Pre release, the objects within the selection rectangle are highlighted, such that the user knows, what his new selection will be before releasing the mouse, showing the user the state of the selection.

To enter add state and add multiple nodes to graph, the user can select the add icon from the toolbar. Nodes added in this state will not be connected to anything. This state is the starting point of the software but once the user leaves it, the only way to return to the Add state is to actively either press a hot key (default 'a') or select the state from the toolbar. Only this state has this functionality. This breaks some of the internal consistency, but enables the other actions to stay as intuitive as possible.

If users require the add state more often than the other states or users struggle to learn how to handle the program, changes ought to be considered.

**Toolbar.** This widget contains five icons. The first two icons allow the user to change the state of the Graph Sketching Window. The first icon sets the state to selection while the second icon sets the state to add. The third and fourth icons control whether new user drawn edges are drawn as undirected or directed. Clicking the third icon will ensure new edges to be drawn as undirected while clicking the fourth ensures new edges to be drawn as directed. The fifth icon allows the user to create a new graph. Multiple graphs are supported by the software, however they cannot be connected like nodes within graphs can. Figure 3.6 shows the widget.

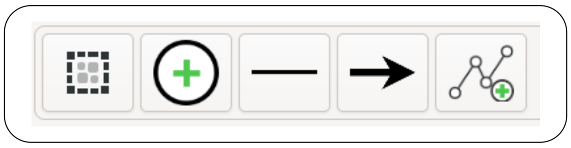


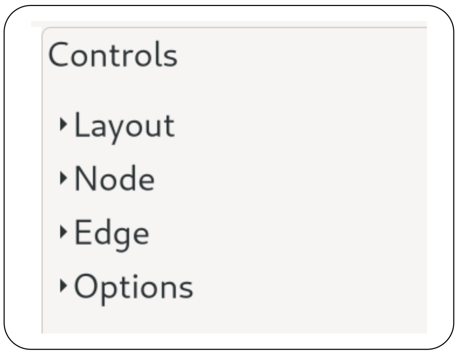
Figure 3.6: Toolbar

**Control Panel.** This widget controls how the user can access the widgets that enable customisation of graph properties.

By allowing the user to expand and hide the items of his interest, the product allows some limited adaption of the interface.

The control panel is divided into the following sections: Layout algorithms, customisation of node and edge properties and additional options. Figure 3.7 shows the widget with no expanded section.

Figure 3.7: Control panel with no expanded section



**Layout Panel.** This widget panel provides an interface to the layout algorithms implemented by boost. A theoretic introduction to the algorithms is available in 2.2. Figure 3.8 shows the widget.

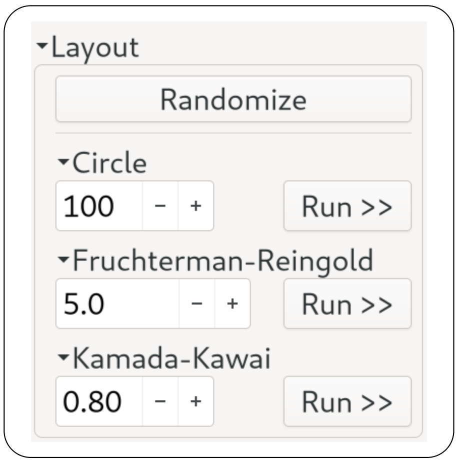


Figure 3.8: Layout Panel with expanded sections

The layout can be influenced by the available inputs. The radius of the circle layout can be defined. For Fruchterman-Reingold and Kamada-Kawai the input affects the maximum edge length.

The implementation requires further parameters to be set. A brief explanation of implicit and explicit parameters, as well as the time complexity of the algorithms, is given in table 3.1.

Table 3.1: Layout algorithms with variables

| **Variable** | **Description** | **Default** |
|---|---|---|
| $n$ | number of nodes | - |
| $E$ | edge set | - |
| $\|E\|$ | number of edges | - |
| k | spring force | 1 |
| T | number of iterations | - |
| **Algorithm** | **Time Complexity** | **Parameters** |
| Random | $O(n)$ | **Seed value**: current time |
| Circle | $O(n)$ | **Radius**: Manually set by user |
| Fruchterman-Reingold | $O(T * (n^2 + \|E\|))$[a] | **C**: Manually set by user[b] <br> $\mathbf{f_r}(\mathbf{d}) = -k^2/d$[c] <br> $\mathbf{f_a}(\mathbf{d}) = d^2/k$[d] |
| Kamada-Kawai | $O(T * n^3)$[e] | **Weight map**: 1 for all edges <br> **Side length**: Screen dependant |

[a] $T = 100$ as default
[b] **C**: proportionally influences the edge length
[c] $\mathbf{f_a}(\mathbf{d})$: repulsive force between every pair of nodes.
[d] $\mathbf{f_a}(\mathbf{d})$: attractive force between two connected nodes.
[e] Kamada-Kawai runs until the end condition is fulfilled. $T$ can therefore vary greatly.

**Node Panel.** This widget allows the user to customise node properties such as the node label, font, font colour, radius, inner and outer colours, border thickness, whether to display the node label and whether to display the node id. When properties are set, they are applied to all nodes selected in the graph sketching window or, if none are selected, to all nodes at once. When a single node is selected, the properties visible in the node panel are updated with the properties of that node. Figure 3.9 shows the widget.

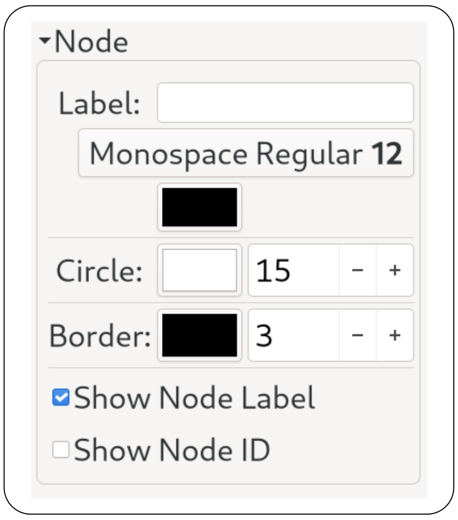


Figure 3.9: Node Panel

**Edge Panel.** This widget allows the user to customise edge properties such as label, font, font colour, line width and line colour. It is also possible to set the edge label distance, which is the distance from the edge label to the edge measured perpendicularly, whether the edge is directed and whether the edge label should be displayed. The side on which the edge label appears can be changed by changing the sign of the label distance value. When properties are set, they are applied to all edges selected in the graph sketching window, or, if none are selected, to all edges at once. When a single edge is selected, the properties visible in the edge panel are updated with the properties of that edge. Figure 3.10 shows the widget.

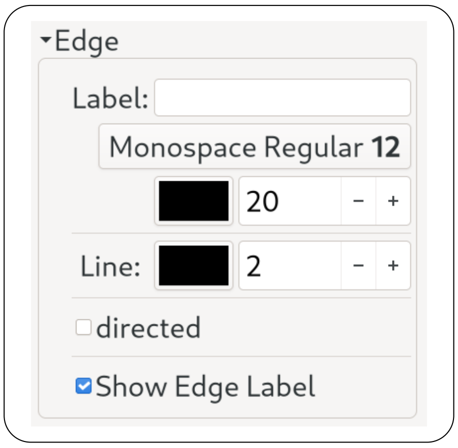
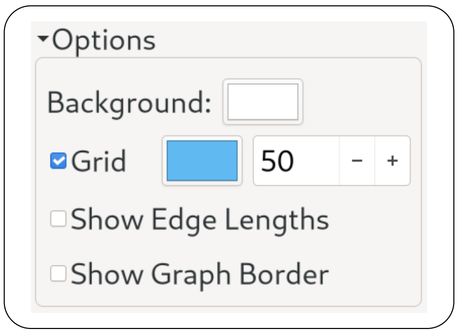


Figure 3.10: Edge Panel

**Options Panel.** This widget offers several options to influence the user experience. The background colour is customisable. The user can choose whether to use a grid, and if so, the colour and size of the grid. The grid snaps nodes moved by the user to the nearest grid point, making it easier to draw graphs. The user can choose to display edge lengths. This can help the user to stay consistent when drawing. The user can specify whether the software should draw a border around graphs. If there are several graphs, this helps to distinguish between them. Figure 3.11 shows the widget with an active grid set.


Options

Background:

✓ Grid 50

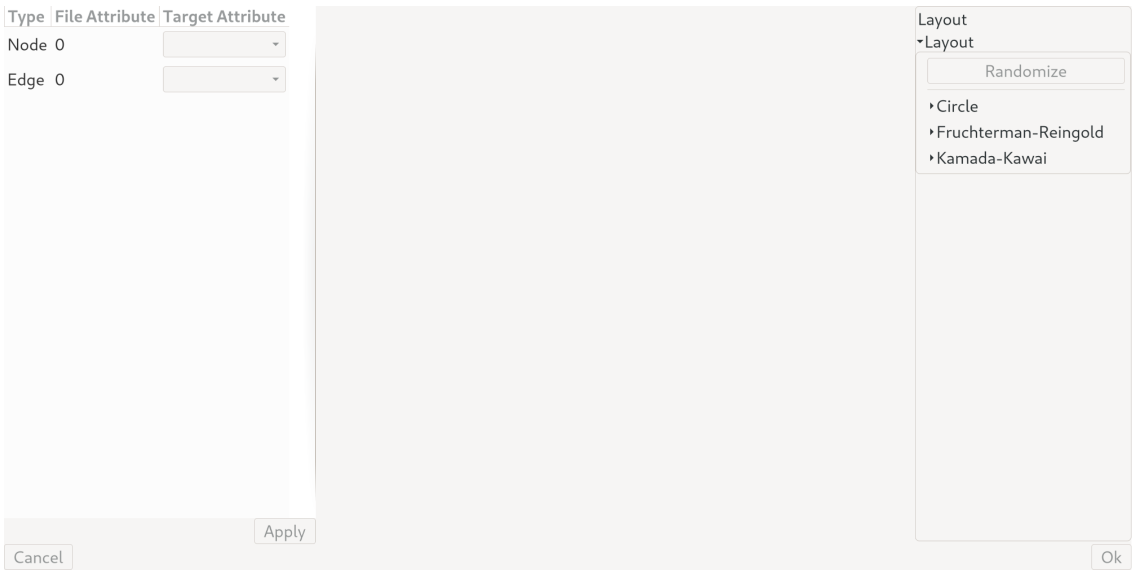
Show Edge Lengths

Show Graph Border

Figure 3.11: Node Panel

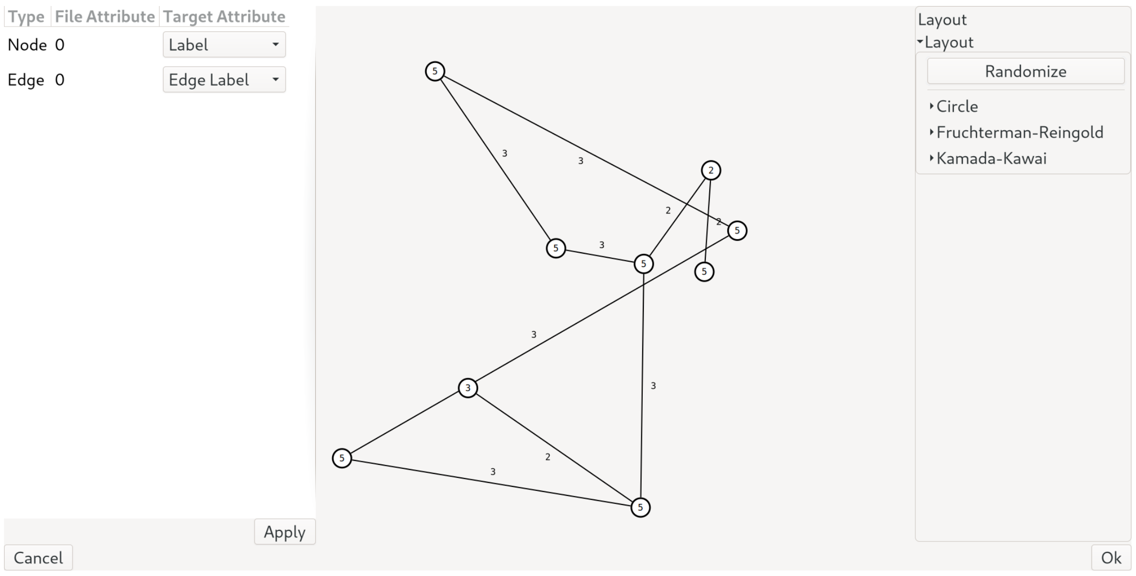
**Graph Load Window.** The graph load window loads a graph from a `graphml` file and allows the user to set loaded attributes if any are found. It consists of three parts: The attributes part, where the user can choose how to interpret the attributes found in the `graphml` file. A graph drawing area, where a preview of the graph is drawn after apply has been called, so that the changes made by setting the attributes are visible. A layout panel that looks and works the same as the layout panel of the main drawing area, thus maintaining consistency.

This setup allows the user to load and lay out graphs quickly and with little input. Figure 3.12 shows the interface and figure 3.13 shows the interface after the attributes have been applied and the layout randomised.



| Type | File Attribute | Target Attribute |
|---|---|---|
| Node | 0 | |
| Edge | 0 | |

Figure 3.12: Graph Load Window



| Type | File Attribute | Target Attribute |
|---|---|---|
| Node | 0 | Label |
| Edge | 0 | Edge Label |

Figure 3.13: Graph Load Window after applying target attributes and randomising the layout

# Chapter 4

# Evaluation

This chapter examines how the graph layout algorithms used by the software compare. It describes how the layouts are generated and what criteria are taken into account. It briefly explains the datasets. It shows the layouts and takes a closer look at some of the graph layouts. It briefly discusses the results.

A flowchart of the user actions performed to generate the graph layouts is shown in figure 4.1. When finished, the graph is exported to an image. Eight sample graphs are randomly selected and laid out once for each layout algorithm implemented. The results are compared in terms of the aesthetic criteria below. The section 2.1 defines the aesthetic criteria in more detail.

- **Minimise edge crossings.** Fewer crossings make a graph easier to read. The total number of crossings is counted and displayed.
- **Maximise symmetry.** Symmetry is usually pleasing to the eye. A statement is given describing the symmetry observed.
- **Uniform edge length.** Uniform edge lengths usually make graphs easier to interpret. The shortest, longest and average edge lengths are given. In addition, the magnitude of the deviation of the longest and shortest edge length from the average edge length is given as a percentage of the average edge length. This percentage is more useful than the absolute numbers because the absolute numbers depend on the screen size used to display the graphs.

**Datasets.** Two datasets are used, both provided by graphlearning.io [12]. The first contains unlabelled data from the IMDB [13]. The second contains labelled data for molecules [14] [15] with a single edge attribute and a single node attribute. When the graph is loaded, the attributes are assigned to node labels and edge labels respectively. From both datasets, four samples were randomly selected and laid out for each graph layout algorithm implemented. The results of this process can be

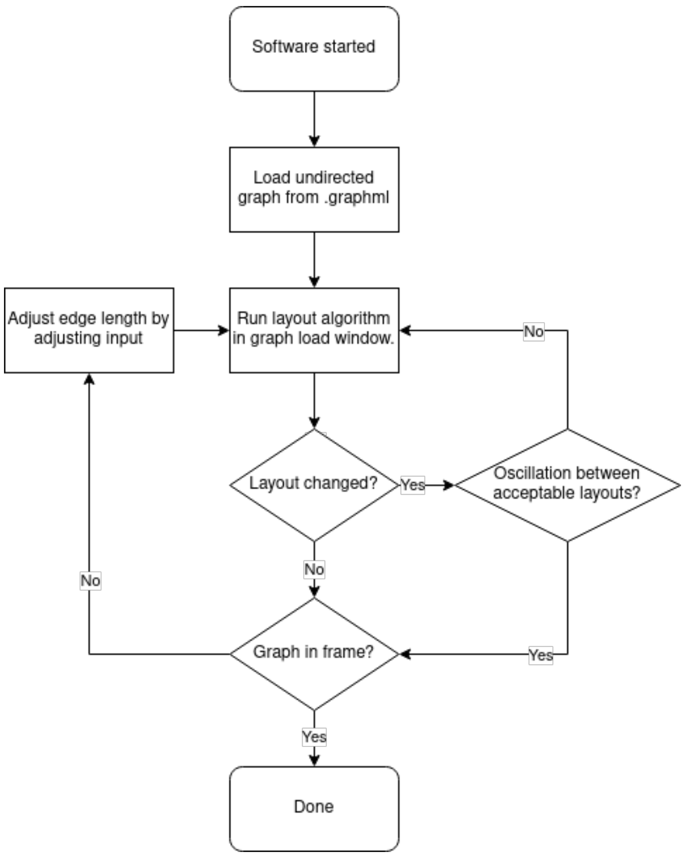

Software started
Load undirected graph from .graphml
Run layout algorithm in graph load window.
Adjust edge length by adjusting input
No
Layout changed?
Yes
Oscillation between acceptable layouts?
No
No
Graph in frame?
Yes
Yes
Done

Figure 4.1: User actions to generate graph layouts

found in tables 4.1 and 4.2 for the IMDB data samples and tables 4.3 and 4.4 for the molecule data samples.

**Results.** Fruchterman-Reingold minimises edge crossings better than Kamada-Kawai for tables 4.1 and 4.2, while Kamada-Kawai has more consistent edge lengths. Since the samples in these tables are more fully connected, the edge lengths do not have much effect on the aesthetics of the resulting layout. In tables 4.3 and 4.4, Kamada-Kawai's layout also manages to produce more uniform edge lengths, which has a greater impact on the aesthetics of the layout, as the more uniform edge lengths lead to better symmetry in these less connected graphs. The circle layout does not seem to offer any visible advantage, except for the fully connected samples in table 4.2, where the circle layout seems to be the most aesthetically pleasing layout overall. The random layout has no consistent characteristics.

These results suggest using the Kamada-Kawais algorithm for sparsely connected graphs and Fruchterman-Reingold for largely connected graphs. For fully connected graphs, the circle layout should be considered.

Table 4.1: IMDB data samples [13]

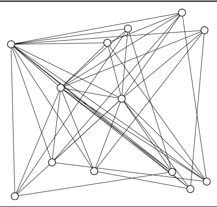
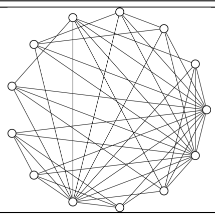
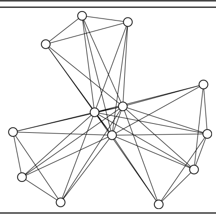
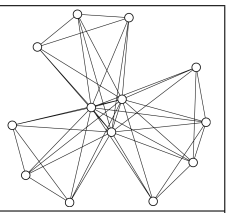
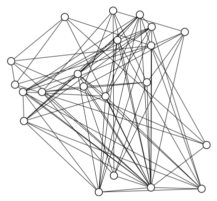
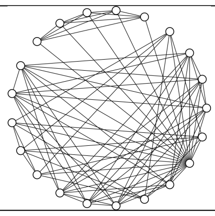
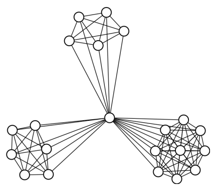
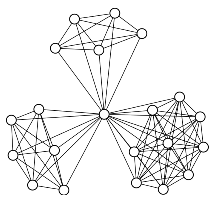
| Aesthetic | Random | Circle | Fruchterman-Reingold | Kamada-Kawai |
|---|---|---|---|---|
| |  |  |  |  |
| Crossings | 134 | 204 | 66 | 68 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>231 (−60%)<br>1071 (+84%)<br>580 | <br>155 (−68%)<br>758 (+51%)<br>499 | <br>72 (−74%)<br>427 (+49%)<br>286 | <br>87 (−71%)<br>437 (+45%)<br>300 |
| Symmetry | None | None | Emerging | Emerging |
| |  |  |  |  |
| Crossings | 632 | 724 | 175 | 190 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>40 (−91%)<br>923 (+88%)<br>489 | <br>83 (−81%)<br>761 (+65%)<br>461 | <br>30 (−75%)<br>319 (+150%)<br>127 | <br>56 (−66%)<br>314 (+86%)<br>167 |
| Symmetry | None | None | Emerging | Emerging |

Table 4.2: More IMDB data samples [13]

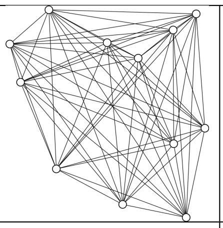
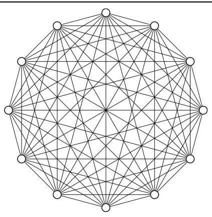
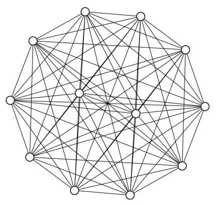
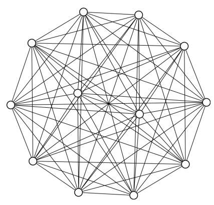
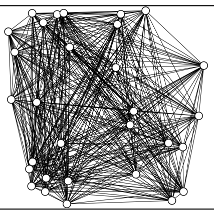
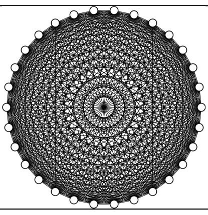
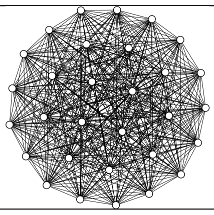
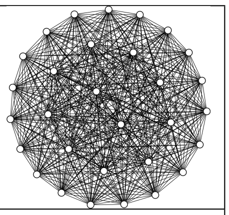
| Aesthetic | Random | Circle | Fruchterman-Reingold | Kamada-Kawai |
|---|---|---|---|---|
| |  |  |  |  |
| Crossings | 359 | 495 | 383 | 391 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>84 (−84%)<br>1026 (+91%)<br>536 | <br>171 (−66%)<br>764 (+47%)<br>516 | <br>186 (−58%)<br>742 (+64%)<br>450 | <br>197 (−59%)<br>791 (+63%)<br>482 |
| Symmetry | None | Symmetric | Emerging | Emerging |
| |  |  |  |  |
| Crossings | 18840 | 26475 | 18868 | 20065 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>2 (−99%)<br>986 (+105%)<br>479 | <br>47 (−90%)<br>764 (+55%)<br>490 | <br>122 (−73%)<br>876 (+87%)<br>466 | <br>110 (−78%)<br>918 (+80%)<br>509 |
| Symmetry | None | Symmetric | Some | Some |

Table 4.3: Molecule data samples [14] [15], part 1.

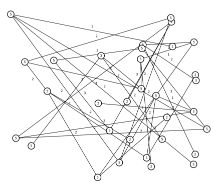
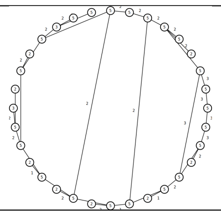
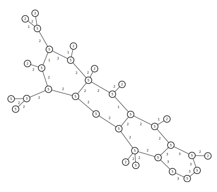
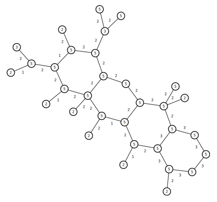
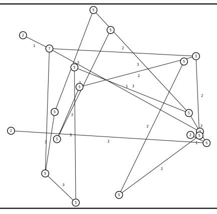
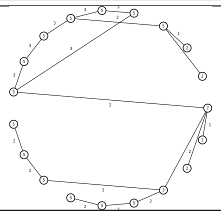
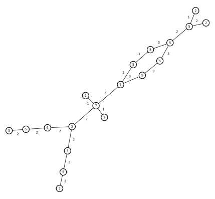
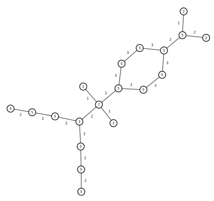
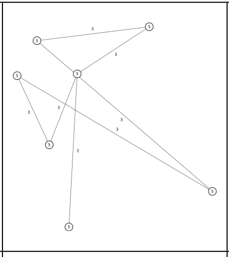
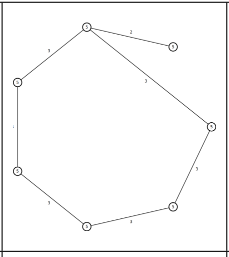
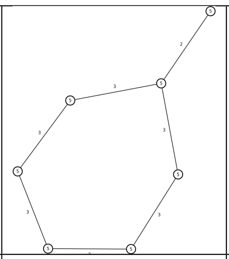
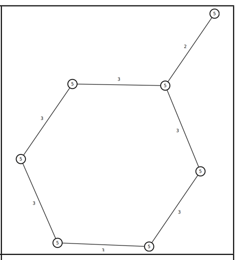
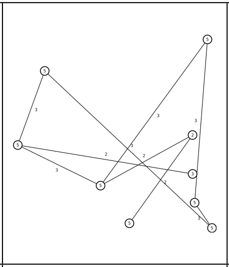
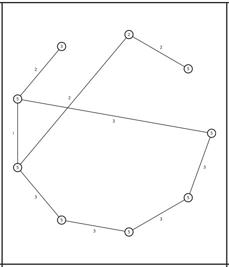
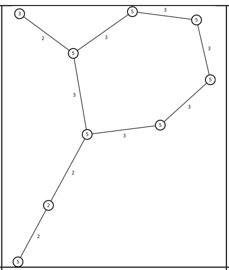
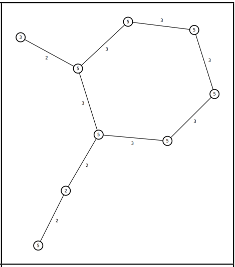
| Aesthetic | Random | Circle | Fruchterman-Reingold | Kamada-Kawai |
|---|---|---|---|---|
| |  |  |  |  |
| Crossings | 159 | 0 | 0 | 0 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>110 (−79%)<br>1136 (+115%)<br>528 | <br>42 (−67%)<br>748 (+475%)<br>130 | <br>27 (−56%)<br>103 (+60%)<br>64 | <br>58 (−11%)<br>73 (+10%)<br>66 |
| Symmetry | None | None | Emerging | Reproduced with some imprecisions |
| |  |  |  |  |
| Crossings | 23 | 1 | 0 | 0 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>46 (−89%)<br>860 (+97%)<br>436 | <br>95 (−55%)<br>761 (+256%)<br>213 | <br>36 (−50%)<br>126 (+72%)<br>73 | <br>65 (−13%)<br>84 (+11%)<br>75 |
| Symmetry | None | None | Symmetric | Symmetric |

Table 4.4: More molecule data samples [14] [15], part 2.

| Aesthetic | Random | Circle | Fruchterman-Reingold | Kamada-Kawai |
|---|---|---|---|---|
| |  |  |  |  |
| Crossings | 2 | 0 | 0 | 0 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>3 (−99%)<br>677 (+16%)<br>420 | <br>311 (−11%)<br>589 (+68%)<br>350 | <br>262 (−6%)<br>296 (+6%)<br>279 | <br>275 (−4%)<br>294 (+2%)<br>289 |
| Symmetry | None | None | Symmetric | Symmetric |
| |  |  |  |  |
| Crossings | 7 | 1 | 0 | 0 |
| *Edges*<br>Shortest:<br>Longest:<br>Average: | <br>80 (−82%)<br>836 (+83%)<br>457 | <br>237 (−30%)<br>751 (+120%)<br>341 | <br>169 (−15%)<br>239 (+19%)<br>200 | <br>185 (−9%)<br>214 (+4%)<br>204 |
| Symmetry | None | None | Emerging | Reproduced with some imprecisions |

# Chapter 5

# Conclusion and Future Work

This chapter contains the conclusion of this thesis and a section on future work. The conclusion will briefly cover the work done in this thesis and outline the results of the evaluation. The section on future work describes the features that were not the scope of the thesis and additional features that can be developed.

## 5.1 Conclusion

This thesis has developed an interactive application that facilitates the drawing and customisation of graphs. The unique feature of the application is that it provides graph layout algorithms capable of automatically arranging undirected graphs without positional data. This allows researchers working with large graph datasets to visualise their data efficiently.

Four layout algorithms are implemented: random, circle, Fruchterman-Reingold and Kamada-Kawai. The last two of these are force-directed layout algorithms. The design of the application follows human-computer interaction principles, resulting in a user-friendly interface that minimises the learning curve for users.

The core functionality of the application is based on two well-maintained frameworks, boost, which provides the graph algorithms and structures, and gtkmm, the basis of the user interface. The use of the gtkmm interface style ensures familiarity and meets user expectations. Furthermore, all graphical components rely primarily on widgets provided by gtkmm, while any additional functionality required beyond the widgets assumes user knowledge based on commonly used editors, such as Office products.

To evaluate the efficiency of the layout algorithms provided by the Boost framework, two large graph datasets were used for experimentation. Four samples were taken for each dataset. For each sample, each layout algorithm generated one layout. The resulting layouts were compared based on three aesthetic criteria: the number of edge crossings, edge lengths with values for average edge length, shortest edge and longest edge, and the ability to reproduce the inherent symmetry of the graphs.

Based on the results, the choice of the right layout algorithm depends on the given graph. For sparsely connected graphs, similar to molecular structures, the Kamada-Kawai layout algorithm gives good results due to its better enforcement of uniform edge lengths. For largely but not fully connected graphs, Fruchterman-Reingold achieves good results, which can be superior to Kamada-Kawai due to Fruchterman-Reingold's clustering of nodes. When a graph is fully connected, the circle layout should be considered, as it results in a perfectly symmetric polygon.

In conclusion, this thesis has successfully developed a user-friendly application that allows researchers to efficiently lay out large undirected graphs, enabling them to gain valuable insight and intuition from their graph datasets. By streamlining the visualisation process, the application improves the conditions for conducting future graph research with greater ease and effectiveness.

## 5.2 Future Work

An important feature that remains to be completed is the display and customisation of graph edit paths. These edit paths are essential for describing the operations that transform one graph into another, including insertions, deletions and substitutions.

While the software currently supports the drawing of graph edit paths and is capable of describing edit paths between two graphs as shown in figure 5.1, further development is required to allow users to define their own custom edit paths. This can be achieved by implementing an intuitive interface, either by enabling the software to read a text file that defines the graph edit operations, or by providing a user-friendly widget that allows the user to define the edit operations manually. A possible design for manual definition of graph edit operations by the user can be found in the appendix A.1. The implementation of this user-defined graph editing path feature significantly enhances the capabilities of the application, giving users greater flexibility and control over their graph transformations.

Future work for this bachelor thesis includes a number of possible features and improvements to enhance the functionality and user experience of the application:

1. Graph export window: Implement a dedicated window that gives users more control over the export of graphs. This feature could include options for selecting file formats, resolution and other relevant export settings.

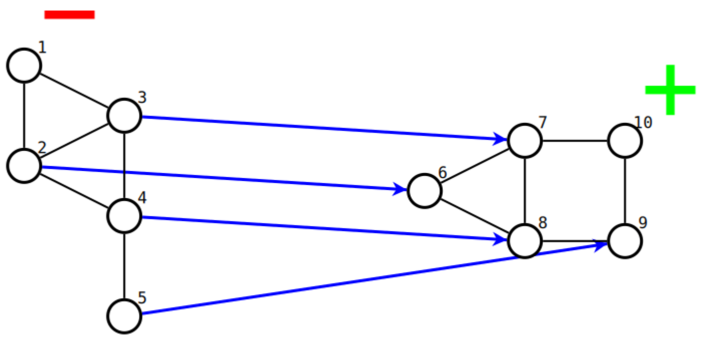


Figure 5.1: Graph edit path as drawn by the software

2. Additional graph layout algorithms: Explore and integrate new graph layout algorithms to give users a wider range of options for automatically arranging graphs. Different algorithms may be appropriate for different types of graphs or visualisation requirements.

3. Other graph algorithms: Consider incorporating various graph algorithms beyond layout, such as centrality measures, shortest path algorithms, and connectivity analysis, to enrich the application's graph analysis and exploration capabilities.

4. Improve test coverage: Enhance the application's testing procedures to ensure comprehensive coverage and validate the correctness and robustness of the software under different scenarios.

5. Support for additional data formats: Extend the application's compatibility by adding support for loading graphs from various data formats commonly used in graph-related research and applications.

6. Hotkey customisation: Allow users to customise the application's functionality by assigning hotkeys to frequently used actions, improving efficiency and ease of use.

7. Graph selection by clicking on the box surrounding the graph: Implement an intuitive feature that allows users to select graphs by clicking on the bounding box that surrounds the entire graph, simplifying the graph selection process.

8. Move graph by dragging box around graph: Allow users to move graphs by dragging the bounding box, providing a convenient and intuitive way to rearrange the visual representation.

9. Graph view window: Introduce a graph view window that displays the nodes and edges of the graph in a tree-like structure to help users understand complex graphs and improve navigation.

10. Other forms for nodes: Explore the inclusion of different node representations beyond the current forms, such as icons, images, or custom shapes, to accommodate different visualisation preferences.

11. User evaluations: Conduct user evaluations to assess and refine the application's user experience. Gather user feedback to identify areas for improvement and make informed enhancements to optimise the user experience.

# Appendix A

# Additional Figures

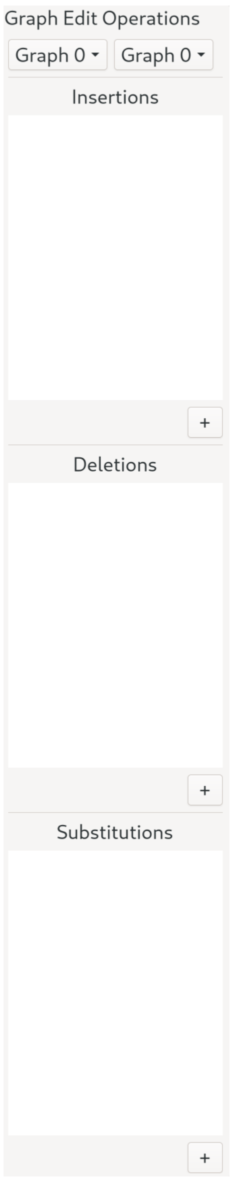


Figure A.1: A possible interface design to define graph edit operations manually

# Bibliography

[1] William T. Tutte. How to draw a graph. *Proceedings of The London Mathematical Society*, 13(1):743--767, 1963.

[2] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149--160, 1984.

[3] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7--15, 1989.

[4] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exp.*, 21(11):1129--1164, 1991.

[5] Kaspar Riesen. *Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications.* Advances in Computer Vision and Pattern Recognition. Springer, 2015.

[6] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom.*, 4:235--282, 1994.

[7] Kaspar Riesen. Mensch maschine schnittstelle, mms, September 2022.

[8] graphtool. https://github.com/DominikFischli/GraphTool. Accessed: 2023-07-20.

[9] boost.org. https://www.boost.org. Accessed: 2023-07-16.

[10] gtkmm.org. https://gtkmm.org/en/index.html. Accessed: 2023-07-16.

[11] gtk.org. https://gtk.org. Accessed: 2023-07-16.

[12] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020.

[13] Pinar Yanardag and S.V.N. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1365–1374, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Nils M. Kriege and Petra Mutzel. Subgraph matching kernels for attributed graphs. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress, 2012.

[15] C. Helma, R. D. King, S. Kramer, and A. Srinivasan. The Predictive Toxicology Challenge 2000–2001 . *Bioinformatics*, 17(1):107--108, 01 2001.

# Erklärung

gemäss Art. 30 RSL Phil.-nat.18

| | |
|---|---|
| Name/Vorname: | Fischli Dominik |
| Matrikelnummer: | 15-822-273 |
| Studiengang: | Informatik |
| | Bachelor ☑ Master ☐ Dissertation ☐ |
| Titel der Arbeit: | Interactive Graph Drawing, A Drawing Software |
| LeiterIn der Arbeit: | PD Dr. Kaspar Riesen, Mathias Fuchs |

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern/02.08.2023

Ort/Datum

Unterschrift D. Fischli