

DRAWING GRAPHS WITH GRAPHIC

by

Rachel L. Bood

Thesis submitted in partial fulfillment of the
requirements for the Degree of
Bachelor of Science with
Honours in Computer Science

Acadia University

January 2016

© Copyright by Rachel L. Bood, 2016

This thesis by Rachel L. Bood
is accepted in its present form by the
School of Computer Science
as satisfying the thesis requirements for the degree of
Bachelor of Science with Honours

Approved by the Thesis Supervisor

Dr. Jim Diamond

Date

Approved by the Director of the School

Dr. Darcy Benoit

Date

Approved by the Honours Committee

Dr. Anna Redden

Date

I, Rachel L. Bood, grant permission to the University Librarian at Acadia University to reproduce, loan, or distribute copies of my thesis in microform, paper or electronic formats on a non-profit basis. I, however, retain the copyright in my thesis.

Signature of Author

Date

Acknowledgements

I would like to thank my supervisor, Dr. Jim Diamond, for the patient guidance, encouragement and advice he has provided throughout my time as his student. He has been a great mentor and friend and I have learned so much from him during my years at Acadia. I would also like to thank all the members of staff at Jodrey School of Computer Science who have also helped me during the years of my Computer Science degree. Finally, I'd like to express my gratitude for my friends and family who experienced all of the ups and downs of my studies and helped me keep things in perspective.

Contents

Abstract	xvii
1 Introduction	1
1.1 Grapha	4
1.2 Graph Drawing Software	8
1.2.1 yED	10
1.2.2 Tulip	10
1.2.3 Meurs Challenger	11
1.2.4 Microsoft Research AGL	11
1.2.5 BioFabric	14
1.3 The Goal	14
2 Requirements	17
2.1 Grapha Walkthrough	17
2.1.1 Select a Graph	17
2.1.2 Edit a Graph	18
2.1.3 Saving and Loading Graphs	19
2.1.4 Combining Graphs	20
2.2 Improvements	22
2.2.1 Graphs	22
2.2.2 Attributes of a Graph	23
2.3 File Output Formats	23
2.4 Joining Graphs	23

2.5	Graphical User Interface	23
2.6	Additional Features	24
2.6.1	Freestyle Graph Creation	24
2.6.2	Graph Editing	25
2.6.3	Graph Component Deletion	25
3	Design	27
3.1	Language Choices	27
3.1.1	C++	28
3.1.2	Qt Creator	28
3.2	Review of Grapha's Design	29
3.2.1	Grapha.html	29
3.2.2	Graph.js	30
3.2.3	UserInteraction.js	30
3.2.4	SaveLoadDelete.js	30
3.2.5	RenderConnect.js	30
3.2.6	Outputs.js	31
3.2.7	Objects.js	31
3.2.8	HelperFunctions.js	31
3.2.9	GraphTypes.js	31
3.3	Redesign	31
3.3.1	User Interface	31
3.3.2	Pre-defined Graphs	32
3.3.3	Attributes of a Graph	32
3.3.4	User Modes	33
	Join Mode	33
	Delete Mode	34
	Edit Mode	34
	Freestyle Mode	34
	Drag Mode	34
3.3.5	Graphic Classes	34

	Node Class	34
	Edge Class	35
	Label Class	35
	Graph Class	36
	Preview Class	36
	CanvasView Class	36
4	Implementation	37
4.1	MainWindow Class	37
4.2	Graphic Classes	41
4.2.1	QGraphicsView Class	41
4.2.2	QGraphicsScene Class	41
4.2.3	QGraphicsItem Class	42
	Node Class	42
	Edge Class	42
	Label Class	43
	Graph Class	43
4.3	BasicGraphs Class	43
4.4	Features Added During Development	44
4.4.1	Undo Node Move Features	45
4.4.2	Snap-to-Grid Feature	45
4.4.3	Editing Individual Nodes and Edges on Canvas	45
4.5	Challenges During Development	45
4.5.1	Screen Resolution and Measurements	45
4.5.2	Widget Styles across Operating Systems	46
4.5.3	File Browser differences	46
5	Software	49
5.1	Overview of Graphic's User Interface	49
5.1.1	Graph Input Fields	51
5.1.2	Node and Edge Input Fields	51
5.2	Create and Customize Graphs	52

5.2.1	Freestyle Mode	52
5.2.2	Join Mode	52
5.2.3	Delete Mode	53
5.2.4	Edit Mode	53
5.2.5	Drag Mode	53
5.3	Save and Load Graphs	53
6	Conclusion and Further Work	55
6.1	Future Work	55
6.1.1	Fixes	55
6.1.2	Improvements	56
6.2	Conclusion	56
	Bibliography	59

List of Tables

1.1	A table representing friendships between individuals	2
1.2	Programs and their capabilities	9

List of Figures

1.1	Example of mock data displaying a correlation between years of education and additional income above the average salary.	1
1.2	A graph representing the data from Table 1.1	3
1.3	Two drawings of the same graph, one asymmetrical and one symmetrical	3
1.4	Two graphs, a left graph is drawn with bends in the edges while the graph on the right is drawn without bends.	4
1.5	A drawing of a bipartite graph (left) and a grid graph (right).	4
1.6	Make New Graphs tab	6
1.7	Edit Basic Graphs tab	6
1.8	Combine Saved Graphs tab	7
1.9	Edit Combined Graphs tab	7
1.10	Save Load and Delete tab	8
1.11	The interface of yED displaying an automatically generated binary tree graph (left) and a manually created Petersen graph (right).	10
1.13	Output of Meurs Challenger. Although difficult to interpret due to the density of the graph, the nodes represent actors while the edges (drawn in very fine coloured lines) represent movies actors have starred in together.	11
1.12	Tulip’s user interface and sample output	12
1.14	Output of AGL.	13
1.15	A example of how the program Biofabric models a graph’s nodes and edges.	14

2.1	In the Make New Graphs tab the Petersen graph has been selected to be drawn with “small” nodes.	18
2.2	In the Edit Basic Graphs tab the Petersen Graph is drawn with a rotation of 10 degrees. The nodes and edges have been assigned labels and weights respectively.	19
2.3	The Save, Load, and Delete Graphs tab displaying the saving, loading and deleting features of Grapha	20
2.4	In the Combine Saved Graphs tab the Petersen graph constructed in Figure 2.2 is to be combined with a Cycle graph at vertices V_6 and V_4	21
2.5	In the Edit Combined Graphs tab the graphs in Figure 2.4 have been combined.	22
2.6	An example of a basic graph (left) and a compound graph (right). The compound graph is made up of two basic graphs: a star and a bipartite graph.	24
3.1	Examples of various Qt Widgets [7]	29
3.2	Additional graphs to be included in Graphic . From left to right: Grid (G_3), Helm (H_3), Crown (R_3), Prism (Y_3), Anti-prism (A_3), Gear (G_3), Dutch Windmill (D_3^6)	33
4.1	A cycle graph with five nodes (C_5)	38
5.1	Graphic user interface.	50

Abstract

In the field of Graph Theory, graphs are abstract representations used to show relationships (or connections) between objects. The objects are represented as *nodes* and the relationships between the entities are known as *edges*. Although graphs are abstract mathematical objects, it is common to use graph drawings to visualize the information they contain; in graph drawings lines symbolize edges while circles or ellipses illustrate nodes. Graphs are used frequently in documents and presentations to provide a visual aid when describing correlations and relationships between objects. There are few tools available that generate graph drawings. These tools are either too simple and provide few or no features to output a graph, or are too powerful and complex for the average user. Hence a graph drawing tool called **Grapha** was created to fill this niche. **Grapha** is a simple, easy to use program that provides users with professional looking graphs. Initially, the goal of this thesis was the further development of **Grapha** by improving and adding features to the program. After some review, it was decided that **Grapha** would be rewritten in C++ using the Qt Framework and Qt Creator. **Grapha**'s successor program, named **Graphic**, has a redesigned and simplified interface, additional file outputs and an expanded graph library. Numerous features were also added including a “Freestyle” mode, to create custom graphs, a node editing feature, that allows the user to move nodes around within a graph, and a deletion feature.

Chapter 1

Introduction

In many disciplines people have numerical data or data representing relationships between entities. The understanding of both these types of data can often be improved by providing visual depictions or representations of the data, as the example in Figure 1.1 shows.

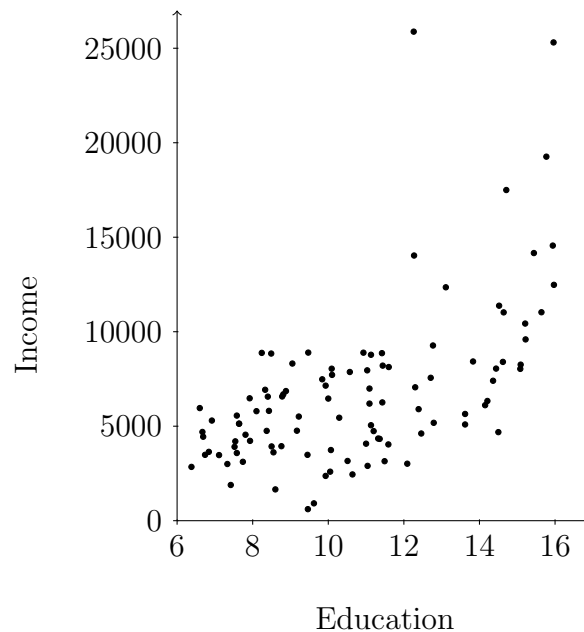


Figure 1.1: Example of mock data displaying a correlation between years of education and additional income above the average salary.

In other areas of study, the data of interest is a collection of “entities” and the relationships between the entities. For example, a study may require data as shown in Table 1.1 that indicates who knows whom in a group of several individuals.

Table 1.1: A table representing friendships between individuals

Person	Erin	Eric	Robert	Trevor	Jill	Maria	Edward
Erin		×	×	×	✓	✓	✓
Eric	×		✓	✓	✓	✓	×
Robert	×	✓		✓	✓	✓	✓
Trevor	×	✓	✓		×	×	×
Jill	✓	✓	✓	×		✓	✓
Maria	✓	✓	✓	×	✓		✓
Edward	✓	×	✓	×	✓	✓	

A *graph* $G = (V, E)$ is a set of *vertices* (also known as nodes or points) and a set of *edges* (also known as lines) [12]. The data in Table 1.1 could be modeled as a graph where the individuals are represented as nodes and their relationships are represented as edges. Although a graph is an abstract idea, it can be visualized with a graph drawing. For example, a drawing of the graph representing the information in Table 1.1 is shown in Figure 1.2; this graph has seven nodes and 14 edges. These drawings can be crucial aids when conveying information.

Graphs are used extensively in various branches of mathematics and computer science. Consequently, graph drawings are of considerable interest in both of these fields. The arrangement of nodes and edges affects the understandability, usability and aesthetics of the graph and what it represents. There are several ways to measure these qualities [10]; one such measurement is graph symmetry. Symmetry is defined as the arrangement of balanced proportions (such as size, shape, and position). Figure 1.3 shows two drawings of the same graph. The symmetric drawing on the right makes it easy to see various “regular” aspects of the graph, such as the fact that there is a cycle comprising all of the nodes but one. This is much less obvious in the asymmetrical drawing on the left. Further, one might hypothesize that most people would find the symmetrical drawing more aesthetically pleasing. Another measurement used to create aesthetically pleasing graphs is known as “minimal edge crossing”.

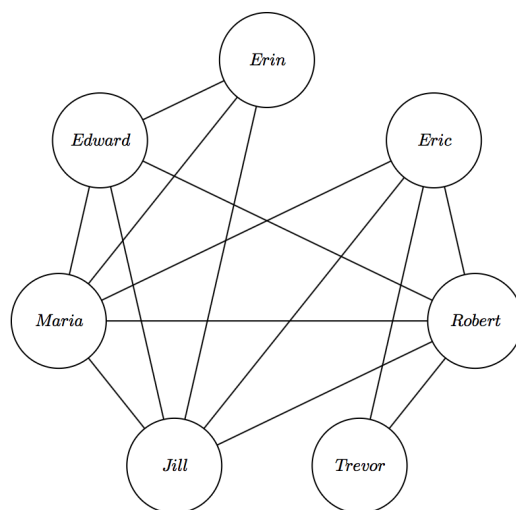


Figure 1.2: A graph representing the data from Table 1.1

The edge crossing number, denoted $cr(G)$, is the number of edges that overlap each other [19]. A graph that has an edge crossing number of zero indicates that the graph is planar; that is, it can be embedded on a plane. The number of bends in edges is another graph drawing measurement. As Figure 1.4 demonstrates, a graph drawing with bends in the edges can be considered more difficult to interpret than a graph drawing without bends [24].

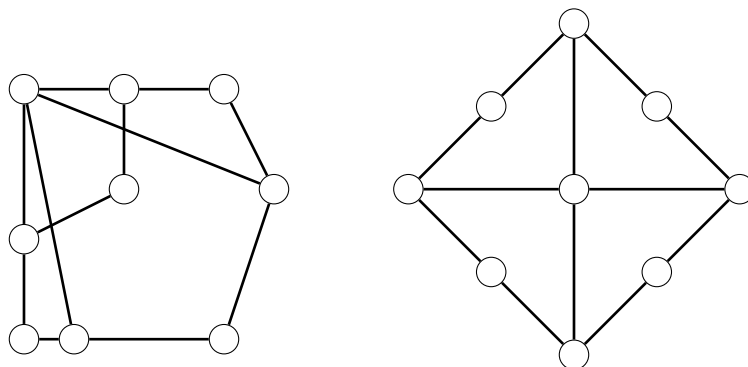


Figure 1.3: Two drawings of the same graph, one asymmetrical and one symmetrical

Creating a graph drawing with aspects of symmetry or regularity is difficult to do free-hand, even with “snap to grid” and “align” tools provided by some computer drawing tools. While such tools allow the creation of graphs (such as bipartite or

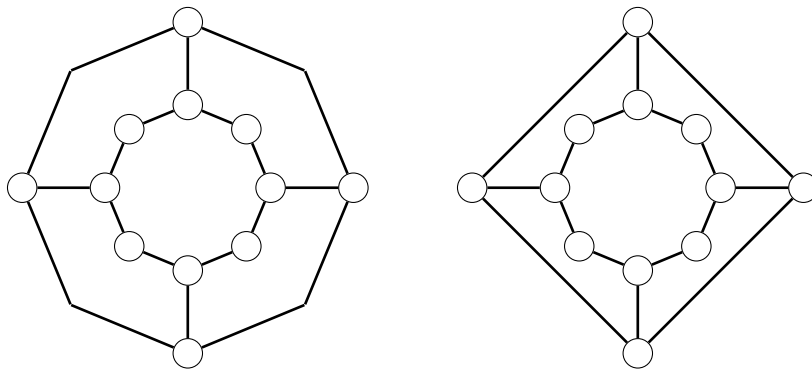


Figure 1.4: Two graphs, a left graph is drawn with bends in the edges while the graph on the right is drawn without bends.

grids) shown in Figure 1.5 without much difficulty, other graph drawings based on circles (such as cycles and Petersen graphs) are exceedingly difficult to lay out by hand so they look aesthetically appealing. There is a need to provide a program that can quickly generate quality graph drawings to visualize data. One tool whose purpose is to allow people to interactively create high-quality graph drawings is known as **Grapha**.

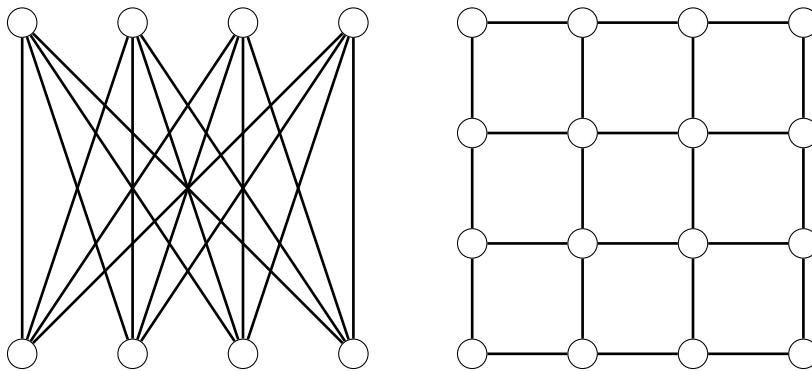


Figure 1.5: A drawing of a bipartite graph (left) and a grid graph (right).

1.1 Grapha

Grapha was started during the fall of 2013 by Nicolas Wetmore under the supervision of Dr. Jim Diamond. The program was completed in the spring of 2014 [28]. The

goal was to design and implement an easy-to-use program that could quickly generate visual representations of graphs, with an easy-to-use graphical user interface (also referred to as a GUI). The user's graphs could be saved in different file formats. The program was written using JavaScript, CSS and HTML and runs in a web browser but does not require an internet connection. **Grapha**, as Wetmore [28] describes it, has a "five-tabbed single-page user interface. . . Every tab contains a different set of functions which can be performed. The flow of the user interface is from left to right". As shown in Figures 1.6, 1.7, 1.8, 1.9 and 1.10, the user begins at the leftmost tab titled **Make New Graphs** where he or she selects a graph they would like drawn. The graph drawing is rendered using a html tag titled "canvas" that allows raster-based image creation [28]. Once the graph is created the user is automatically directed to the **Edit Basic Graphs** tab where the user can edit the attributes of a graph drawing, such as graph height, graph width, and node sizes. The user also has the option to select the output format of their graph from this tab. The **Save, Load, and Delete Graphs** tab has features that allow the user to save or delete their graph drawings and the **Combine Saved Graphs** and **Edit Combined Graphs** tabs allow them to combine graphs and edit the combined graphs respectively.

Make New Graphs	Edit Basic Graphs	Save, Load, and Delete Graphs	Combine Saved Graphs	Edit Combined Graphs
Click below to choose a graph, or change your decision to another graph.				
Wheel ▾				
Node Size Medium ▾	Text Size (pt) 12	Size (Pixels) 500	Generate Graph	
For each partition below, input the number of vertices that are contained in the partition.				
Partition 1				
5				
<input checked="" type="checkbox"/> Label the vertices?				
For each partition below, input the label name of the partition's vertices and the number at which that section begins. Partitions that you give the same label name to will continue their numbering scheme.				
Partition 1				
Label		Begin at		
v		0		

Figure 1.6: Make New Graphs tab

Make New Graphs	Edit Basic Graphs	Save, Load, and Delete Graphs	Combine Saved Graphs	Edit Combined Graphs
Load Graph ▾	Graph Name: Cycle,t,5,v,0,500,500	Save Graph To Cache		
Weight Options Right ▾ Default Weight		Node Size Medium ▾	Text Size (pt) 12	Update Output
Partition 1 Label v	Begin at 0	Label the vertices? <input checked="" type="checkbox"/>	Size (Pixels) 500	Rotation (Degrees) 0
To edit weights, double click on an edge. To move weights, drag a weight to a different location.				

Figure 1.7: Edit Basic Graphs tab

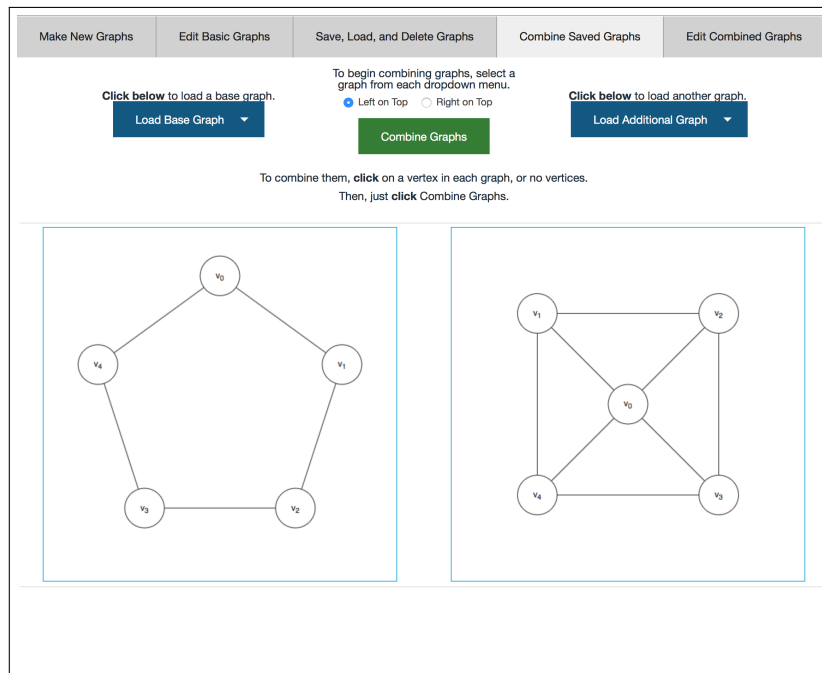


Figure 1.8: Combine Saved Graphs tab

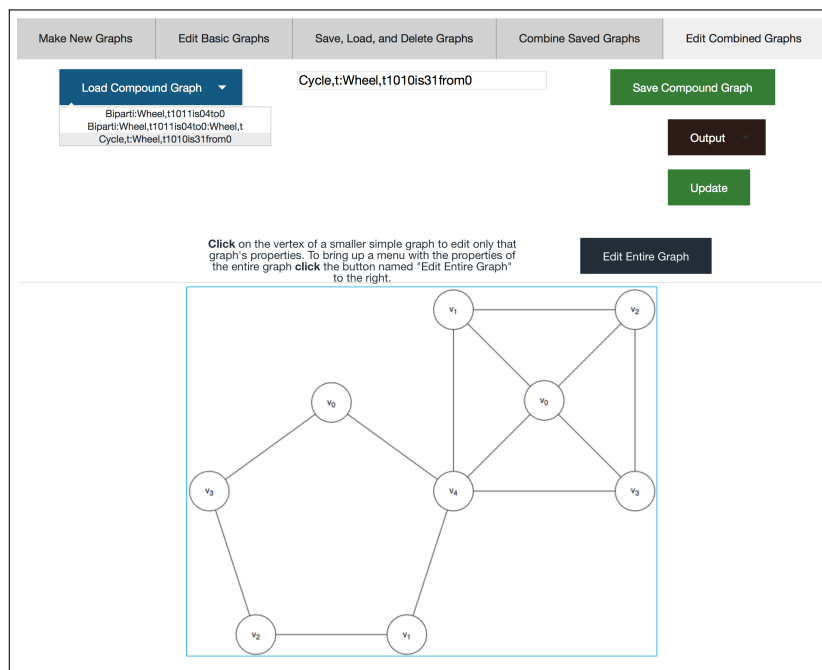


Figure 1.9: Edit Combined Graphs tab

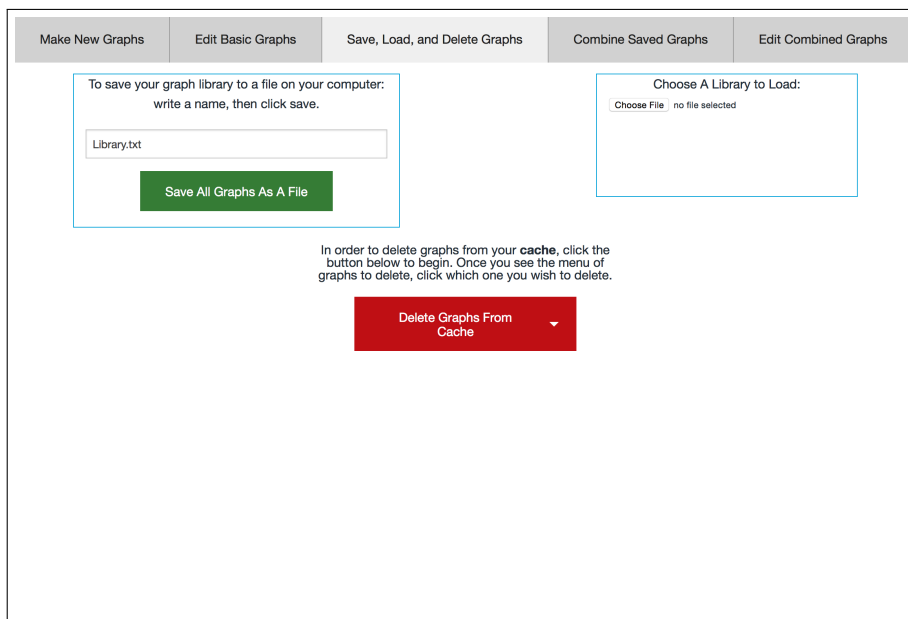


Figure 1.10: Save Load and Delete tab

1.2 Graph Drawing Software

All of the graph drawing programs detailed in Wetmore's thesis are summarized in Table 1.2. For clarification of Table 1.2a, Wetmore noted if each program could be run on a mobile device and whether it requires installation under the columns "Mobile" and "Installed" respectively. The programs listed in Wetmore's thesis are still available today. In addition to Wetmore's findings, there are other graph drawing programs available.

Table 1.2: Programs and their capabilities

(a) Supported platforms and portability

Software	Desktop	Mobile	Installed	On-line	Off-line
Graph Creator	✓	×	×	×	✓
Creately	✓	×	✓	✓	✓
GraphTea	✓	×	✓	×	✓
GraphViz	✓	×	✓	×	✓
Gephi	✓	×	✓	×	✓
Grapha	✓	✓	×	✓	✓

(b) Outputs

Software	Raster-Based Output	Latex Output	SVG Output
Graph Creator	×	×	×
Creately	✓	×	✓
GraphTea	✓	✓	×
GraphViz	✓	×	✓
Gephi	✓	×	✓
Grapha	✓	✓ TikZ	✓

(c) Ease and speed of use

Software	Graph Generation	Specific Editing	Learning Curve	Time to Use
Graph Creator	×	✓	minimal	average
Creately	×	✓	average	average
GraphTea	✓	✓	average	minimal
GraphViz	×	With Work	steep	maximal
Gephi	×	✓	steep	average
Grapha	✓	×	minimal	minimal

1.2.1 yED

Written in Java and thus runnable on platforms that support a Java Virtual Machine, yED is considered more of a diagramming software than a graph drawing software. This program can draw various types of diagrams such as flowcharts, network diagrams, UML diagrams, and Entity Relationship diagrams. It does, however, have a small subset of graph drawing algorithms such as grid and tree layouts. As shown in Figure 1.11 the program can generate a tree graph automatically, but should the user want to draw a Petersen graph they would need to generate it manually. Drawing a Petersen graph manually can be time consuming and may not produce quality results. The program has various output formats for the user to save their graph drawings [29].

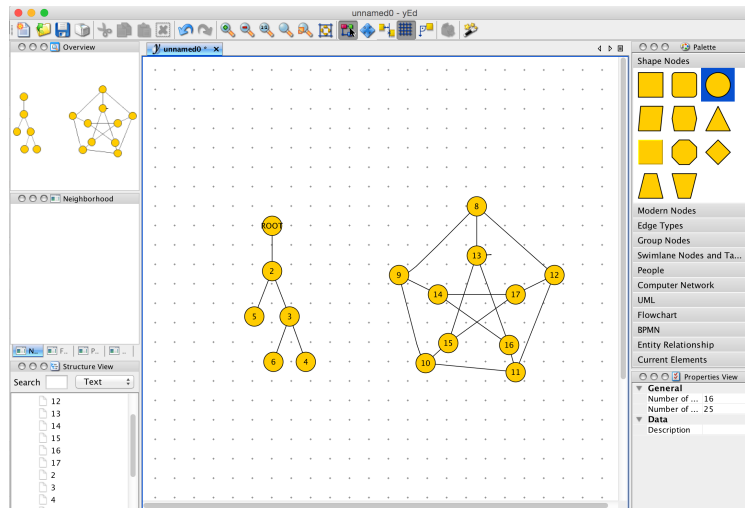


Figure 1.11: The interface of yED displaying an automatically generated binary tree graph (left) and a manually created Petersen graph (right).

1.2.2 Tulip

Tulip is a data modeling program; two views of its user interface are shown in Figure 1.12a. As its website explains, “Tulip is an information visualization framework dedicated to the analysis and visualization of relational data” [26]. The program is data driven and creates graphs based on abstract data. Figure 1.12b demonstrates

Tulip generating a graph displaying an air traffic map from 1990 data. Although Tulip also has a small subset of graph drawing algorithms, the program only allows the user's work to be saved its native file format.

1.2.3 Meurs Challenger

Similar to Tulip, Meurs Challenger is a data visualization software with integrated interactive data analysis and browsing features [16]. As presented in Figure 1.13, the main goal of Meurs Challenger is to visualize a large quantity of data to make it readable and understandable for the user.

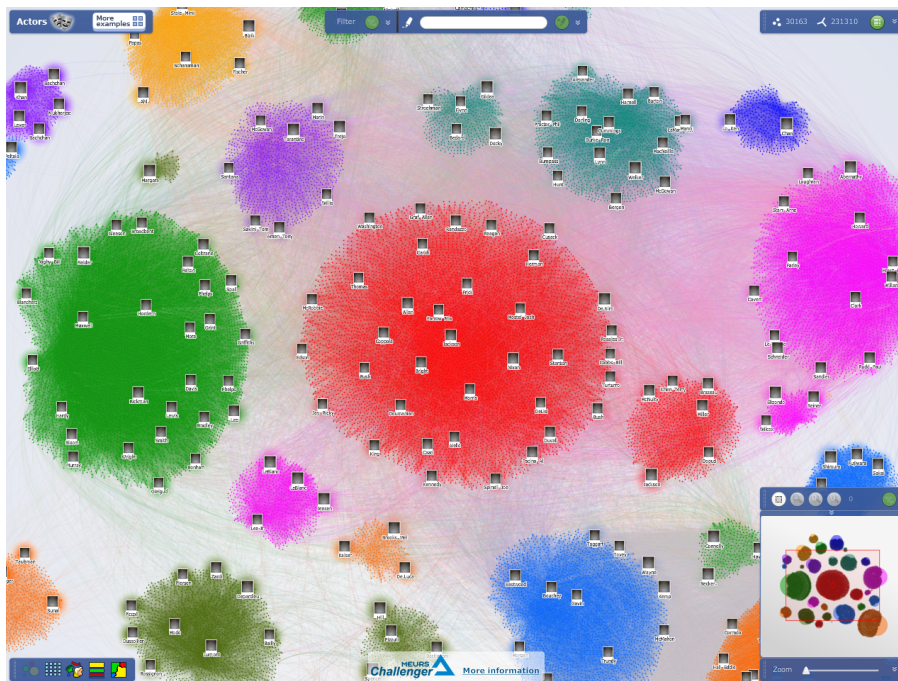
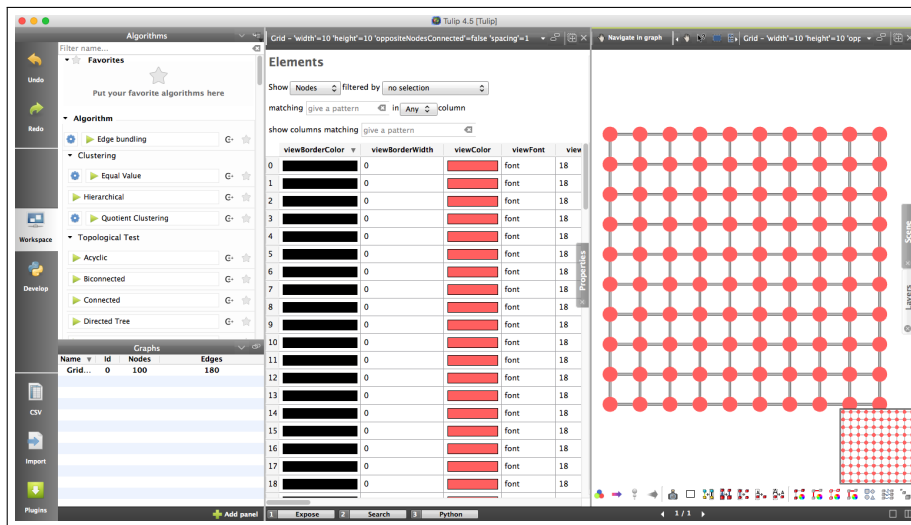


Figure 1.13: Output of Meurs Challenger. Although difficult to interpret due to the density of the graph, the nodes represent actors while the edges (drawn in very fine coloured lines) represent movies actors have starred in together.

1.2.4 Microsoft Research AGL

Microsoft Research AGL (Automatic Graph Layout): AGL can be executed from inside a web browser or installed on a computer. While it allows graph customization,



(a) An example of Tulip's user interface with a grid graph.



(b) Output of Tulip generating a graph displaying an air traffic map from 1990 data.

Figure 1.12: Tulip's user interface and sample output

AGL requires the user to type in the attributes of a graph using a somewhat verbose and tedious input format. Therefore, this program lacks ease of use and requires user knowledge of the input format [25]. Listing 1.1 and Figure 1.14 provide input and output examples of AGL.

Listing 1.1: A sample of input the user is required to write to generate the graph shown in Figure 1.14.

```

1 digraph "Honda-Tokoro" {
2   rankdir="LR" ranksep="0.2" edge[labelfontsize="8"
      fontsize="8" labeldistance="0.8" arrowsize="0.9"
      labelangle="-30" dir="none"] nodesep="0.2" node[width
      ="0" height="0" fontsize="10"]
3   n000 [label="z"]
4   n001->n000 [headlabel=":s:" arrowhead="invdot"]
5   n001 [label="m"]
6   n002->n001 [samehead="m002" headlabel=":r:"
      samearrowhead="1" arrowhead="invdot" arrowtail="inv"]
7   n002 [label="p1"]
8   n003->n002 [headlabel=":s:" arrowhead="dot"]
9   n003 [label="b"]

```

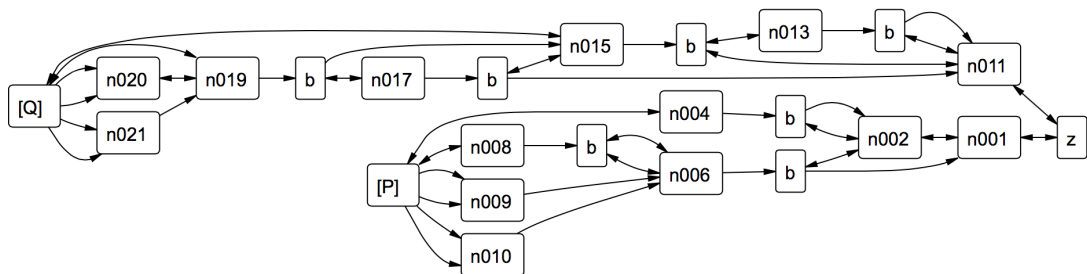


Figure 1.14: Output of AGL.

1.2.5 BioFabric

BioFabric is another graph drawing software package which uses an alternative modeling method to draw graphs (Figure 1.15). In BioFabric, nodes are depicted as horizontal line segments while edges are represented as vertical line segments; each edge has its own unique column. This specific layout scheme can be useful for biologists to display large networks in a organized display [13], [20].

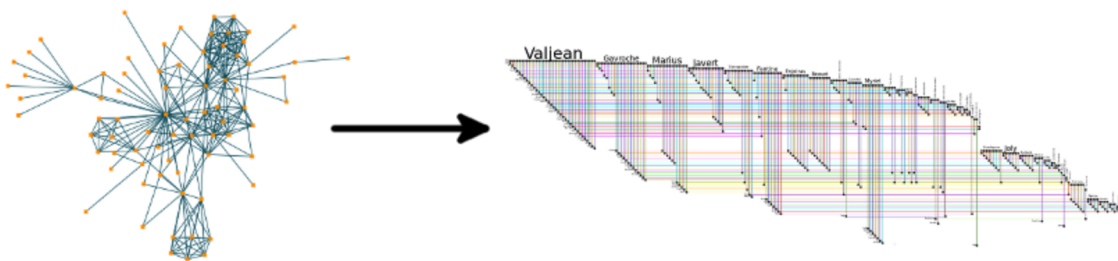


Figure 1.15: A example of how the program Biofabric models a graph’s nodes and edges.

1.3 The Goal

The majority of graph drawing programs discussed above are being developed to model and analyze large and complex systems such as social networks. Amongst all these graph drawing programs, **Grapha** is useful because it allows interactive construction of a graph drawing; that is, **Grapha** fills a need which none of these other programs address. Wetmore outlined five goals for **Grapha** to help define an “easy to use” program [28]:

- **Quick Graph Generation:** **Grapha** will need a way in which to generate entire graphs at once without much interaction from the user. This is opposed to generating graphs in a way, which requires the user to specify each detail along the way.
- **Quick to Use:** For a user to use **Grapha** and obtain output, he should not have to spend much time. Another way of describing this is in terms of the

number of clicks. There should be a minimal number of clicks required in order to generate and output a graph.

- **Easy to Learn:** The path that the user needs to follow in order to generate a graph should be apparent and intuitive.
- **Different Outputs:** The user should be able to output a graph from **Grapha** in many different formats.
- **Portable:** **Grapha** should run on mobile and desktop operating systems as well as run both on-line and off-line in order to be accessible to a large number of users in many different locations.

Obtaining quantitative measures on Wetmore's goals **Easy to Learn** and **Quick to Use** would be difficult, which renders them subjective. While designing rigorous experiments to measure these would be an interesting investigation, it is outside the main thrust of this thesis.

Therefore this thesis will take a different approach. There is research available that provides paradigms and models to create effective and intuitive graphical user interfaces [17], [27], [22], [18], [15]. These models allow UI designers to communicate concepts and relationships that exist in the application. Communicating these concepts well enough allows the user to quickly understand how to use the program.

The goal and focus of this thesis is to maintain Wetmore's goals of **Grapha** while improving the functionality that Wetmore established. Each feature of **Grapha** that will be refined, as well as the features that will be added, will have the same underlying concept: the ease of use. **Grapha**'s successor program, **Graphic**, will be designed with these goals and features in mind.

The following five chapters of this thesis describe **Graphic** and its features in detail. Chapter 2 addresses the features of **Grapha** that will be improved upon as well as the features to be added. Chapter 3 explains the design of the software. Chapter 4 details the implementation of the new software. Chapter 5 discusses the use of the program. Chapter 6 concludes with possible future work and features to further improve **Graphic**.

Chapter 2

Requirements

After reviewing Wetmore's thesis and program, a number of additions and areas for improvement were identified. As stated previously, defining and implementing these improvements is the focus of this thesis.

2.1 Grapha Walkthrough

Before expanding upon improvements to **Grapha**, it would be beneficial to explain the steps one would take to generate a graph in this program.

2.1.1 Select a Graph

As the user opens **Grapha** in a browser the program starts at the first tab, **Make New Graphs**, as shown in Figure 2.1. The user must select from a range a graph types. Then the user will be prompted to change attributes of the graph drawing, such as adjusting the size of the nodes, the height and width of the graph and the choice of whether to label the nodes. Once the user clicks on **Generate Graph** the **Edit Basic Graphs** tab will be selected and the user's graph will be displayed.

Make New Graphs Edit Basic Graphs Save, Load, and Delete Graphs Combine Saved Graphs Edit Combined Graphs

Click below to choose a graph, or change your decision to another graph.

Petersen

Node Size: Small Text Size (pt): 12 Size (Pixels): 500 Generate Graph

For each partition below, input the number of vertices that are contained in the partition.

Label the vertices?

For each partition below, input the label name of the partition's vertices and the number at which that section begins. Partitions that you give the same label name to will continue their numbering scheme.

Partition 1

Label: v Begin at: 0

Figure 2.1: In the **Make New Graphs** tab the Petersen graph has been selected to be drawn with “small” nodes.

2.1.2 Edit a Graph

In the **Edit Basic Graphs** the user can select various options to customize their graph. As displayed in Figure 2.2 the options from the previous tab are carried over, in addition to adding weights to the edges. Since **Grapha** has a graph saving feature, a saved graph can be loaded into this tab for editing. Once the editing is completed there is the option to save the graph to cache memory or output the graph in either `.png`, `.jpeg`, `.bmp`, `.webp`, `.TikZ` and `.svg` format.

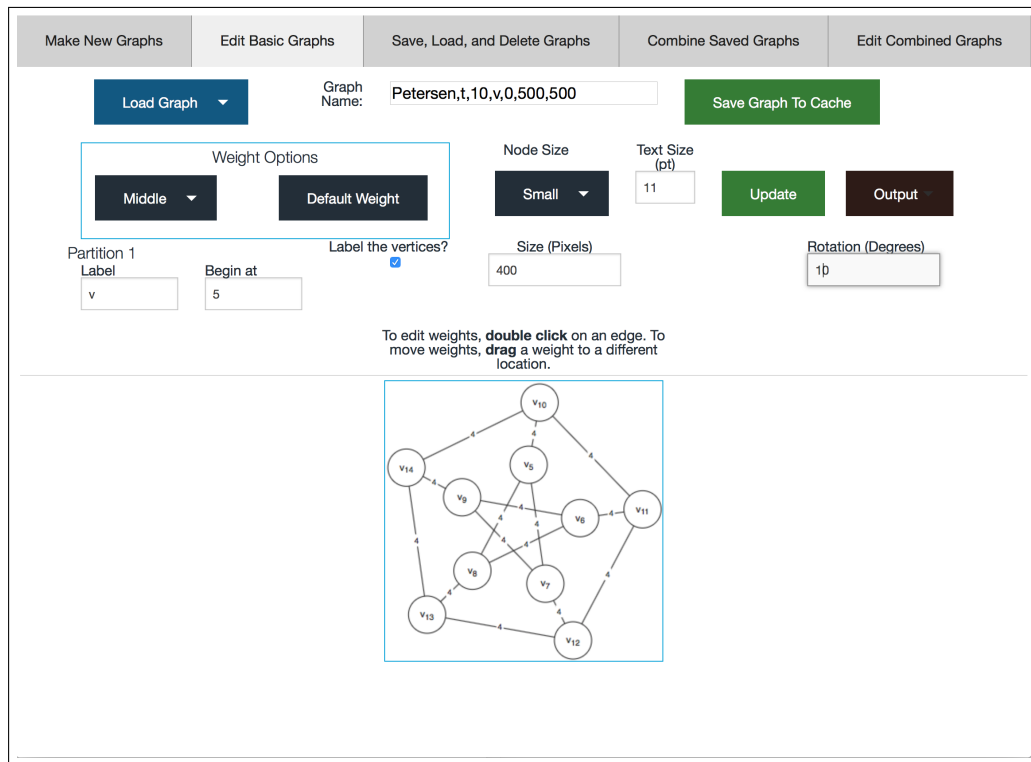


Figure 2.2: In the **Edit Basic Graphs** tab the Petersen Graph is drawn with a rotation of 10 degrees. The nodes and edges have been assigned labels and weights respectively.

2.1.3 Saving and Loading Graphs

All the graphs saved to cache can be deleted or saved to the local file system on the **Save, Load, and Delete Graphs** tab shown in Figure 2.3. All the graphs would be saved into a `.txt` file. This library of graphs can be loaded into **Grapha** in the current or future **Grapha** sessions.

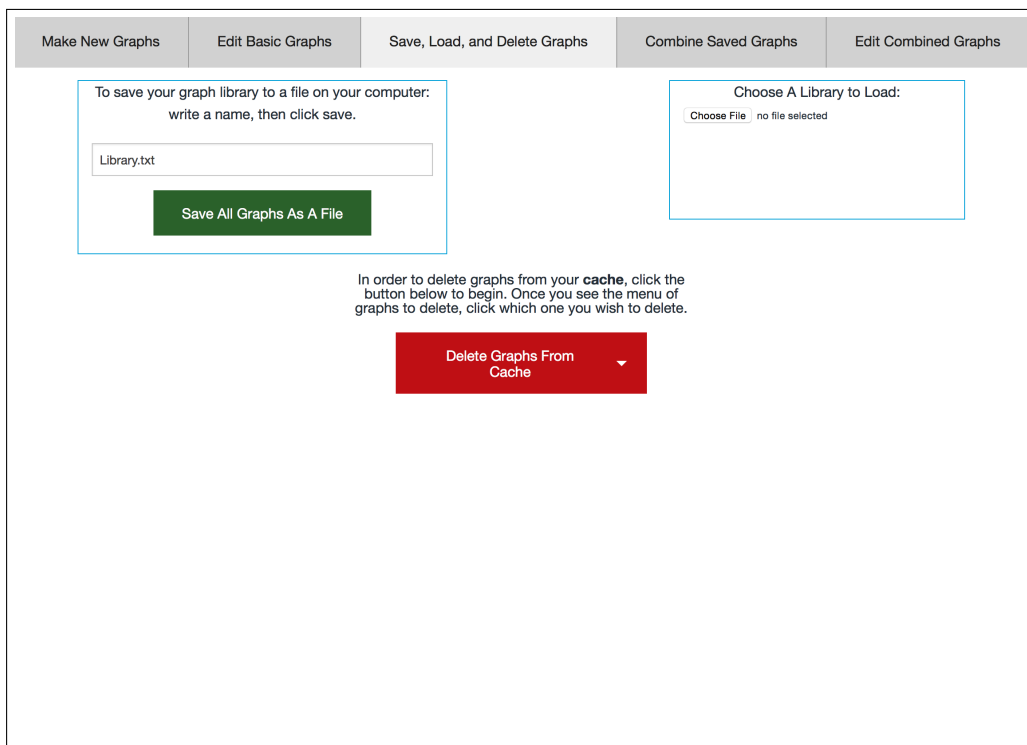


Figure 2.3: The Save, Load, and Delete Graphs tab displaying the saving, loading and deleting features of Grapha

2.1.4 Combining Graphs

When at least one graph has been saved to the cache, graphs from the cache can be combined together in the **Combine Saved Graphs** tab to form a so-called “compound graph” (This idea is explained further below). First, two graphs are copied from the cache (or, one graph from the cache can be copied twice). Then a node from each graph must be selected to indicate where the graphs will be joined as shown in Figure 2.4. By clicking on “Combine Graphs” the **Edit Combined Graphs** tab is selected and the new complex graph is drawn. Much like the **Edit Basic Graphs** tab, the **Edit Combined Graphs** tab, shown in Figure 2.5, allows the user to customize the graph in a similar manner by rotating the graph and relabeling or resizing the nodes. Once the user is satisfied with their compound graph they are able to save it to cache or to a file format mentioned previously.

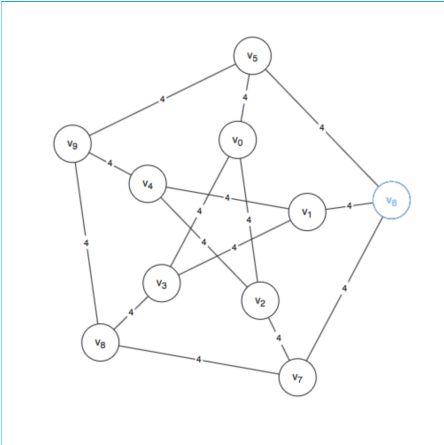
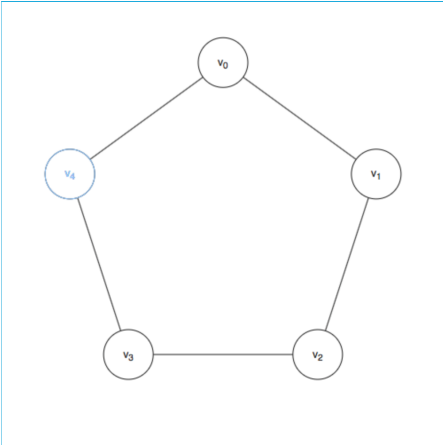
Make New Graphs	Edit Basic Graphs	Save, Load, and Delete Graphs	Combine Saved Graphs	Edit Combined Graphs
<p>To begin combining graphs, select a graph from each dropdown menu.</p> <p><input checked="" type="radio"/> Left on Top <input type="radio"/> Right on Top</p> <p>Combine Graphs</p> <p>To combine them, click on a vertex in each graph, or no vertices. Then, just click Combine Graphs.</p>				
<p>Click below to load a base graph.</p> <p>Load Base Graph ▾</p>		<p>Click below to load another graph.</p> <p>Load Additional Graph ▾</p>		
				

Figure 2.4: In the **Combine Saved Graphs** tab the Petersen graph constructed in Figure 2.2 is to be combined with a Cycle graph at vertices V_6 and V_4 .

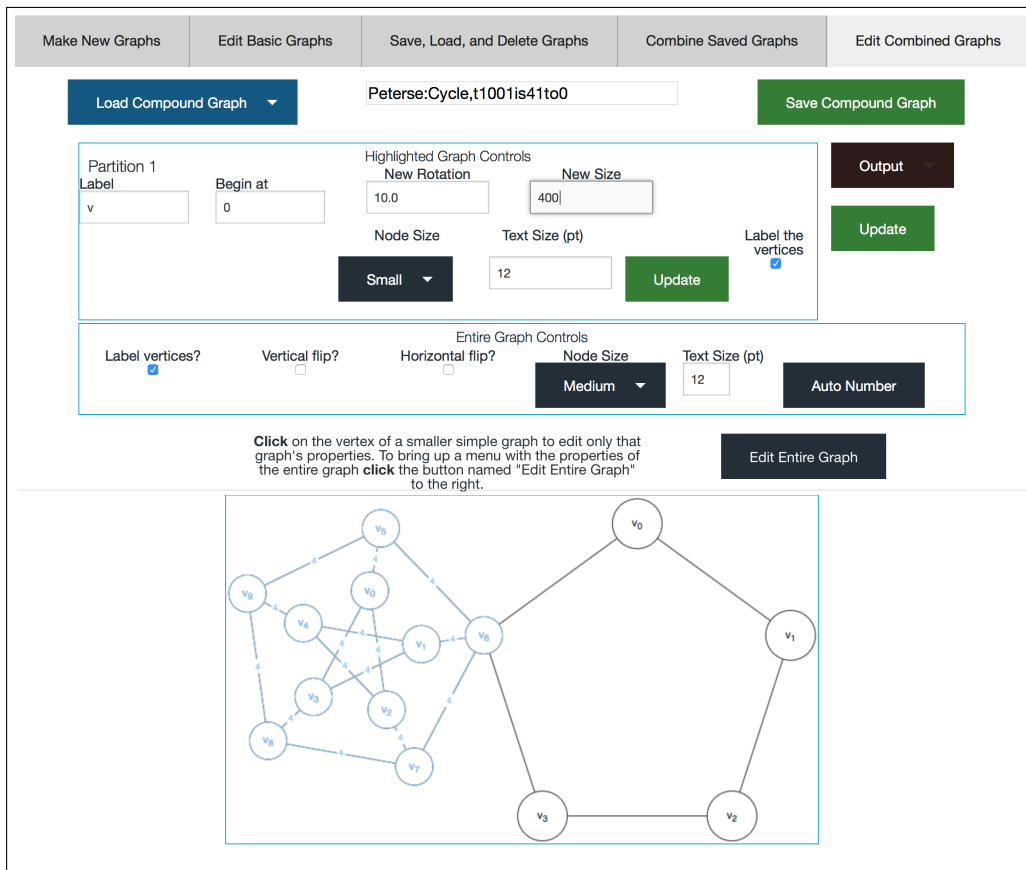


Figure 2.5: In the **Edit Combined Graphs** tab the graphs in Figure 2.4 have been combined.

2.2 Improvements

The improvements of **Grapha**'s features shall be explained in the following sections:

2.2.1 Graphs

Currently, **Grapha** allows the user to choose between eight different pre-defined graph types. **Graphic** will have an expanded library of graphs for the user to select from.

2.2.2 Attributes of a Graph

Currently in **Grapha**, the user is able to adjust the dimensions of the graph, the number of nodes per graph and the rotation. The user can further customize the graph by adding labels and weights to nodes and edges respectively. **Graphic** will contain the same input fields and provide additional ones to allow more detailed customization.

2.3 File Output Formats

Currently, **Grapha** only has six output file formats. More file formats will be added to **Graphic**'s output repertoire to allow flexibility for the user. Another output file format, with the file format `.edges`, will be created, by special request. This file format provides a listing of edges of a graph and may be useful for people developing graph theory algorithms. Finally, a more user friendly **Save file** dialog window will be added so the user can easily select where they want to save their graph and in which format.

2.4 Joining Graphs

In his thesis, Wetmore describes a compound graph as “a graph in which there are multiple basic or compound graphs joined together. These graphs form a larger more complex graph when joined”. The same terminology is also used in this thesis. Figure 2.6 provides an example of a basic graph and a compound graph.

2.5 Graphical User Interface

Although **Grapha** incorporated an easy to use interface, several improvements can be made to it. The features in tabs **Make New Graphs**, **Edit Basic Graphs**, **Combine Saved Graphs** and **Edit Combined Graphs** could be combined into one tab. **Grapha** does not have an established layout or organizational pattern amongst the tabs. That

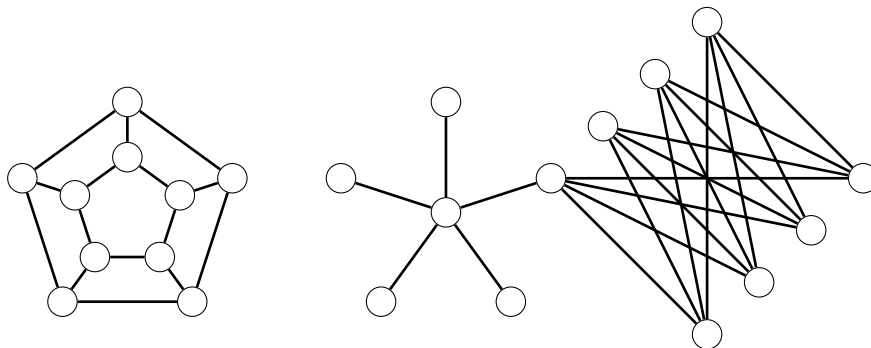


Figure 2.6: An example of a basic graph (left) and a compound graph (right). The compound graph is made up of two basic graphs: a star and a bipartite graph.

is, the layout of the buttons and text fields change from tab to tab. The **Edit Basic Graphs** tab, for example, has a blue border that groups the buttons that edit edge attributes. However, the buttons and text input fields that edit node attributes are not contained within a border. These inconsistencies in the graphical user interface can confuse the user. The Gestalt Principles help explain how humans perceive, organize and make sense of visual information [21]. The similarity principle explains that people often perceive similar looking objects as a group or pattern. The continuation principle illustrates how the eye is compelled to move through one object and continue to another object. The proximity principle describes how elements that are placed close together are usually perceived as a group. These principles can be utilized to create an improved user interface for **Graphic**.

2.6 Additional Features

The following sections describe the new features of **Graphic**.

2.6.1 Freestyle Graph Creation

This new feature will allow the user to generate their own custom graph. With simple mouse clicks the user can create and position nodes and edges to create graphs completely “by hand” or to add nodes and edges to already-defined graphs.

2.6.2 Graph Editing

When the user generates a graph they may wish to rearrange the nodes. An edit feature in `Graphic` will allow the user to do this. Careful consideration must be made for this feature since the edges incident to the moved node must be adjusted and rendered appropriately.

2.6.3 Graph Component Deletion

The user may want to delete certain components from their graph. The user, for example, may want to generate a graph which is almost complete (a *complete* graph is a graph in which every pair of distinct vertices is connected by a unique edge) [9]. A deletion feature was implemented in `Graphic` that allows users to delete edges, nodes or entire graphs. Some care must be put into node deletion, however, since by definition, an edge must have two nodes. Therefore when a node is deleted, all edges incident to that node must also be deleted.

Chapter 3

Design

After some careful consideration, it was decided that `Graphic` would be implemented using C++ and a program called Qt Creator instead of appending JavaScript functions to `Grapha`'s code. This choice will be explained in more detail below.

3.1 Language Choices

As Wetmore [28] explains in his thesis, “JavaScript is not really a single language which is developed by a single entity. Instead, JavaScript is a term used to refer to a group of languages which are all used for similar purposes. . . . There is no single standard for exactly how to interpret or process JavaScript. When a browser encounters JavaScript code on a web page, the browser hands the JavaScript off to an appropriate engine which handles the code for the browser. One problem with this is that almost every major browser uses a different JavaScript engine to interpret and execute the commands of the language”. Since different browsers use different forms of JavaScript there can be dissimilar results for the same code. In `Grapha`, for example, Wetmore had to create separate functionality to allow graph saving if the user ran `Grapha` in the `Internet Explorer` browser. For a program whose goal is to be easy-to-use it would appear that JavaScript could be an unreliable language, since the program could behave differently on various browsers that use different engines to interpret JavaScript. Knowing these drawbacks of JavaScript, it was decided that

`Graphic` would be programmed in a different language.

3.1.1 C++

C++ is a standardized cross-platform language [11]. Bjarne Stroustrup, the creator of C++, describes it as a language that “... allows for high level abstraction and efficiency” [8]. It has the ability to manipulate memory at a low level while enforcing object orienting programming structures with the speed of the C programming language [8]. This language is predominantly used for desktop applications [11]. The nature of `Graphic`’s proposed implementation would be best suited to use an object oriented approach. `Graphic` would contain classes to represent nodes, edges and graph objects. Combining this programming model with the efficiency of a C language, C++ would be a sound choice of language to use for `Graphic`. When paired with an integrated development environment (IDE) that uses C++, such as Qt Creator, `Graphic` would be a robust piece of cross-platform software.

3.1.2 Qt Creator

Qt Creator is a cross-platform IDE designed to facilitate the development of programs using Qt. [5]. The user is provided with a large set of components and widgets to design and develop their own graphical user interface. Widgets, as Qt documentation explains, are “... elements for creating user interfaces in Qt. Widgets can display data and status information, receive user input, and provide a container for other widgets that should be grouped together.” [7]. A few examples of widgets include, but are not limited to, spin boxes, labels and text edit fields; see Figure 3.1 for examples of these widgets. The developer is able to preview their GUI as they design it without the need to compile, which allows them to quickly iterate through the design process. Other IDEs, such as Eclipse, do not provide a GUI builder so another program such as Glade must be used to create the user interface [14], [23]. Qt Creator’s design features, cross-platform capability and large framework make it a suitable IDE to create `Graphic`.

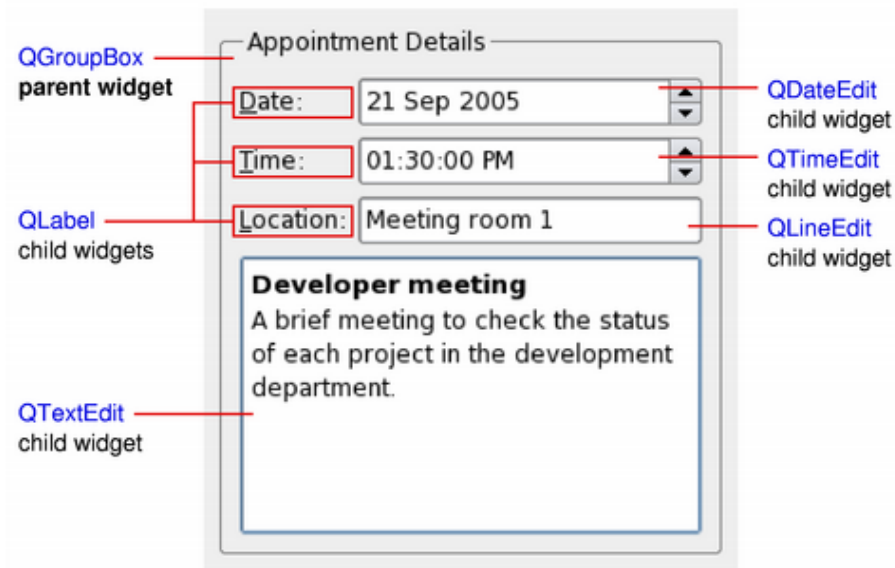


Figure 3.1: Examples of various Qt Widgets [7]

3.2 Review of Grapha's Design

It is important to review Grapha's code to understand the design choices. Ideas used in the development of Grapha were used or improved during the design and implementation of Graphic. Since Wetmore wrote Grapha using JavaScript for use in browsers these ideas must be translated into C++ for Graphic.

3.2.1 Grapha.html

This HTML file contains the code that creates the user interface. It arranges the five different tabs as well as the placement of the input fields. The majority of the input fields are already populated with default values. Once the user selects the graph they want to draw and clicks the button Generate Graph, the values from all the fields are used to render the graph in Edit Basic Graphs of Grapha.

3.2.2 Graph.js

This JavaScript file initializes the default values of the user interface at run time in `Grapha.html`. It also collects the values from the input fields and generates a graph once the user clicks the `Generate Graph` button. `Graph.js` contains other listener functions for the `Edit Basic Graphs` tab of the interface. There are functions that will change all the weights or individual weights of edges based on user interaction. This script also generates and formats a name for the graph that is later used for saving that graph.

3.2.3 UserInteraction.js

This script connects event listeners, such as a listener for mouse drags, with functions found in `Grapha`'s various scripts. Listener functions are created for each tab to handle the events differently. The `Edit Basic Graphs` tab, for example, requires mouse click event listeners to monitor if the user is clicking on an edge (to adjust the value of its weight). The `Edit Combined Graphs` tab, on the other hand, monitors user clicks to check if they have selected nodes that will be used to combine two graphs.

3.2.4 SaveLoadDelete.js

As the name implies, this script handles the loading and saving of graph drawings. This script also handles the storing and deleting of graphs in the browser cache. There are functions that handle the saving and loading depending on the particular browser `Grapha` is running in.

3.2.5 RenderConnect.js

This script contains functions that return lists of edges and nodes that are used to render the different types of graphs onto the `canvas`. There are separate functions for drawing nodes and edges, respectively.

3.2.6 Outputs.js

This script contains the various functions used to generate .png, .jpeg, .bmp, .webp, .tikz and .svg files. This script also contains functions that draw nodes and edges onto the canvas.

3.2.7 Objects.js

This script contains various functions that initialize objects to be used in other scripts. The functions to create node and edge objects are written in this script.

3.2.8 HelperFunctions.js

This script file contains additional functions to assist the drawing of the graphs. There are several translation and rotations functions to alter the orientation of the graph.

3.2.9 GraphTypes.js

This script holds the different graph types `Grapha` has to offer. There is an initializer function for each graph that the user can select. These functions initialize the variables needed to draw the graph on the `canvas`.

3.3 Redesign

The following sections details the ideas selected for implemented in `Graphic`.

3.3.1 User Interface

In `Graphic` the number of tabs used in the interface was reduced to provide the user with all `Grapha`'s features on one window. `Graphic` also provides a more interactive drawing area that will be called the “`canvas`” that allows the user to add, edit and arrange as many graph drawings as they wish. `Graphic` would use a top-down design methodology and group similar UI elements together to create an organized layout

for the GUI. This approach to organizing is known as the Gestalt principle [17]. At the top of the tab widget, the input fields that modify the characteristics of the entire graph, such as the graph type, width, height and rotation, will be grouped together. Below, the input fields that control edge and node characteristics will also be organized in a similar way. Edge and node characteristics can be grouped in two ways: the properties can be grouped by what they are controlling and how they are controlling it. The edge and node colour characteristics, for example, can be grouped together because they perform the same function (they change the colour of edges or nodes) and they can be grouped with characteristics that change the same object (the edge colour can be grouped with the edge weight input field and the edge size input field). Therefore a table layout should be implemented. Each row could specify a characteristic (colour or label) and columns indicate what the characteristic affects (node or edge).

3.3.2 Pre-defined Graphs

A total of seven graphs are added to **Graphic**'s library in addition to those found in **Grapha**'s library. These graphs are grid, helm, crown, prism, anti-prism and gear as shown in Figure 3.2. The Dutch Windmill, also known as the friendship graph, was also added to the library to provide an example that graphs with greater complexity can also be included in **Graphic**. Forethought was required when designing this library such that supplementary pre-defined graphs can be easily added at a later date.

3.3.3 Attributes of a Graph

In **Graphic**, the user will be able to adjust the font size of edge weights and node labels. The user will also be able to resize the edges and nodes, as well as be able to colour edges and nodes. The height and width attributes will be adjusted to allow the user to scale the graph. For example, a cycle graph would always be drawn as a circle in **Grapha**. In **Graphic**, however, a cycle graph could be drawn as an ellipse.

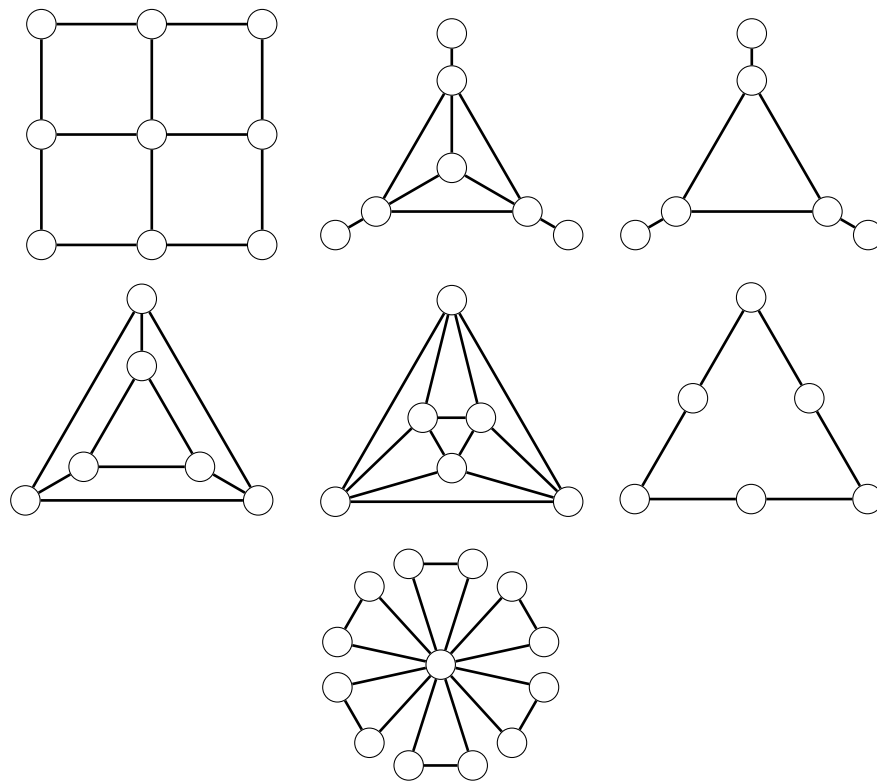


Figure 3.2: Additional graphs to be included in **Graphic**. From left to right: Grid (G_3), Helm (H_3), Crown (R_3), Prism (Y_3), Anti-prism (A_3), Gear (G_3), Dutch Windmill (D_3^6)

3.3.4 User Modes

Graphic has different modes that allow the user to interact with graph drawings in different ways:

Join Mode

This mode allows the user to join two graphs together, forming one larger graph. The user can join two graphs together by choosing either one or two nodes from each graph. If the graphs are joined by one node, the graphs join together by translating one graph to the other while maintaining uniformity in edge lengths of the graph. If the user joins graphs together by two nodes, then the second graph will rotate to align the joining nodes and translate to join the nodes together.

Delete Mode

Users are able to select and delete nodes and edges from a graph. By utilizing a straightforward user input, such as a key stroke or a mouse click, the delete can entire graph or graph attributes with ease.

Edit Mode

Nodes can be freely moved around within each graph. Edges incident to the moving node are updated to reflect its new position.

Freestyle Mode

The user can create their own arbitrary graphs in this mode. It allows the user to add nodes and edges. `Graphic` will also allow the user to add nodes and edges to graphs that are already on the `canvas`.

Drag Mode

The user is able to move the graphs around on the `canvas` to arrange them to his or her liking.

3.3.5 Graphic Classes

The following sections are proposed class ideas for `Graphic`.

Node Class

The `node` class would store the following node attributes:

- Diameter
- Fill colour
- Outline colour
- List of edges incident with the node

- Label
- Label's font size

The user would be able to change visual aspects such as size and colour and the number of edges that connect to the node. The user would also be able to label the node via the label object within the node.

Edge Class

The `edge` class stores similar attributes as the `node` class. The following is a list of edge attributes:

- Source node
- Destination node
- Weight
- Line width
- Colour
- Weight's font size

The edge will be drawn as a solid line. The thickness of the line would be determined by the user. The user would also be able to define its colour and weight.

Label Class

The `label` class would store a string. It is essentially a text field that would be in the center of the object. This class would be used to store labels in node and edge objects. Any value can be entered into the node and edge labels.

Graph Class

The **Graph** class would store **node** and **edge** objects. **Graph** objects would also store other **graph** objects. Establishing this recursive structure allows future programmers of **Graphic** to build upon the current version. For example, a developer may want to create an algorithm or feature that checks how many simple graphs went into the final graph.

Preview Class

As the name suggests, the **Preview** class would generate a preview of a graph. The graph would be generated and updated as the user changes the input fields for the graph. Once the user is satisfied with their graph they could drag it from the **Preview** drawing onto the **CanvasView** Object.

CanvasView Class

The **CanvasView** object will contain and display all the edge, node and graph objects the user creates. The user could further customize their graphs on the **canvas** by moving around the graphs, deleting edges, nodes or entire graphs, adding nodes and edges or even moving around nodes within a graph by way of the new features. The different modes used affect how the user interacts with the **canvas** object.

The next chapter discusses the implementation of these ideas.

Chapter 4

Implementation

Agile development was applied to the creation of `Graphic`, just as it was used in the development of `Grapha` [28]. This development model provided crucial user feedback that allowed `Graphic` to become a powerful tool while still remaining user friendly.

4.1 MainWindow Class

The `MainWindow` class maintains the connections between the different widgets and classes of `Graphic` via Qt's signals and slots mechanism [6]. A signal is emitted when a particular event occurs. A slot, usually a normal function, is connected to a signal and will fire when the signal is emitted. This class contains the methods related to saving image, `grphc`, `edges` and `TikZ` files and loading custom user graphs. Qt contains its own set of classes to read and write images and SVG files. By utilizing these classes, saving graphs as images was a straightforward feature to implement.

The additional file formats such as `.tikz`, `.grphc` and `.edges` require the parsing of information into text format to be read by other programs. Refer to Listings 4.1, 4.2 and 4.3 as output files for the cycle graph in Figure 4.1. The `.grphc` files are the native file format for this program. The information of every node and edge had be stored so the user could reload the same graph at a later time. The `.edges` file format is a text file that contain a list of edges, where each edge is represented as a pair of node indices. This file type provides an abstract representation of a graph that can

be useful for graph theorists as explained earlier in this thesis.

Generating TikZ code was a moderate challenge that required the user's graph information to be modeled in the TikZ language. The same format that Wetmore used to generate TikZ code was used in `Graphic` with some modifications: the colours of the edges and nodes must be stored in the file as well. The user can save their files anywhere on the computer but `Graphic` has a special directory for `.grphc` files in which the user is encouraged to save their files.

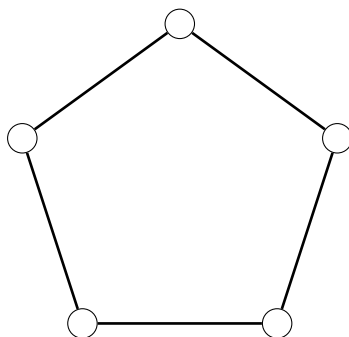


Figure 4.1: A cycle graph with five nodes (C_5)

Listing 4.1: `.edges` output file for a cycle graph with five nodes.

```
1 5
2 0, 1
3 0, 4
4 1, 2
5 2, 3
6 3, 4
```

Listing 4.2: `.TikZ` output file for a cycle graph with five nodes.

```
1 \begin{tikzpicture} [x=1.0in, y=1.0in, xscale=1, yscale
   =1]
2 \definecolor{node0fillColor} {RGB} {255,255,255}
3 \definecolor{node0lineColor}{RGB}{0,0,0}
```



```
4 \node (v0) at (0.165937,1.84444) [scale=1.00,font=\
    fontsize{1}{1}\selectfont, fill=node0fillColor, shape
    =circle,minimum size=0.00277778in,draw=node0lineColor
    ]{$$};
5 \definecolor{node1fillColor} {RGB} {255,255,255}
6 \definecolor{node1lineColor}{RGB}{0,0,0}
7 \node (v1) at (1.25965,1.04981) [scale=1.00,font=\
    fontsize{1}{1}\selectfont, fill=node1fillColor, shape
    =circle,minimum size=0.00277778in,draw=node1lineColor
    ]{$$};
8 \definecolor{node2fillColor} {RGB} {255,255,255}
9 \definecolor{node2lineColor}{RGB}{0,0,0}
10 \node (v2) at (0.84189,-0.235925) [scale=1.00,font=\
    fontsize{1}{1}\selectfont, fill=node2fillColor, shape
    =circle,minimum size=0.00277778in,draw=node2lineColor
    ]{$$};
11 \definecolor{node3fillColor} {RGB} {255,255,255}
12 \definecolor{node3lineColor}{RGB}{0,0,0}
13 \node (v3) at (-0.510016,-0.235925) [scale=1.00,font=\
    fontsize{1}{1}\selectfont, fill=node3fillColor, shape
    =circle,minimum size=0.00277778in,draw=node3lineColor
    ]{$$};
14 \definecolor{node4fillColor} {RGB} {255,255,255}
15 \definecolor{node4lineColor}{RGB}{0,0,0}
16 \node (v4) at (-0.927778,1.04981) [scale=1.00,font=\
    fontsize{1}{1}\selectfont, fill=node4fillColor, shape
    =circle,minimum size=0.00277778in,draw=node4lineColor
    ]{$$};
17 \definecolor{edge0edgeColor}{RGB}{0,0,0}
18 \definecolor{edge1edgeColor}{RGB}{0,0,0}
19 \definecolor{edge2edgeColor}{RGB}{0,0,0}
```

```

20 \definecolor{edge3edgeColor}{RGB}{0,0,0}
21 \definecolor{edge4edgeColor}{RGB}{0,0,0}
22 \definecolor{edge5edgeColor}{RGB}{0,0,0}
23 \path (v0) edge[draw=edge0edgeColor,line width=0.0138889
    in] node [font=\fontsize{12}{1}\selectfont]{$$} (v1);
24 \path (v4) edge[draw=edge1edgeColor,line width=0.0138889
    in] node [font=\fontsize{12}{1}\selectfont]{$$} (v0)
    ;
25 \path (v1) edge[draw=edge2edgeColor,line width=0.0138889
    in] node [font=\fontsize{12}{1}\selectfont]{$$} (v2);
26 \path (v2) edge[draw=edge3edgeColor,line width=0.0138889
    in] node [font=\fontsize{12}{1}\selectfont]{$$} (v3);
27 \path (v3) edge[draw=edge4edgeColor,line width=0.0138889
    in] node [font=\fontsize{12}{1}\selectfont]{$$} (v4);
28 \end{tikzpicture}

```

Listing 4.3: .grphc output file for a cycle graph with five nodes.

```

1 5
2 11.9475,-132.8,0.2,0,1,1,1,0,0,0
3 90.695,-75.5866,0.2,0,1,1,1,0,0,0
4 60.6161,16.9866,0.2,0,1,1,1,0,0,0
5 -36.7211,16.9866,0.2,0,1,1,1,0,0,0
6 -66.8,-75.5866,0.2,0,1,1,1,0,0,0
7 0,1,0.1,0.00694444,0,1,0,0,0
8 0,4,0.1,0.00694444,0,1,0,0,0
9 1,2,0.1,0.00694444,0,1,0,0,0
10 2,3,0.1,0.00694444,0,1,0,0,0
11 3,4,0.1,0.00694444,0,1,0,0,0

```

The user can load a specific graph that has been previously saved to their computer. The `loadGraphicFile` method will open up the `.grphc` file the user selected

and parse the information to generate the graph onto the preview window. A `Graphic` directory (`graph-ic`) is checked every time `Graphic` is executed; any `.grphc` files saved in this directory will be loaded into `Graphic`. This directory is always created and checked in the same directory as `Graphic`

4.2 Graphic Classes

The following sections describe the program layout of `Graphic`. The majority of the classes created for `Graphic` will be subclassed from classes provided from the Qt framework.

4.2.1 QGraphicsView Class

As Qt documentation describes, the `QGraphicsView` is a widget that displays the contents of a `QGraphicsScene` (which will be discussed in further detail later in this chapter). In `Graphic`, `QGraphicsView` has been subclassed into two classes to handle different view widgets with different functionality; the `Preview` widget and the `canvas` widget [4].

The `Preview` object, as the name suggests, generates a preview of a graph the user created from their inputs. When the user is satisfied with the graph they drag and drop the graph over to the `CanvasView`. The `Preview` object provides the user with a quick visual feedback to their inputs.

The `CanvasView` displays the user's graphs and allows the user to interact with their graphs. This view has different behaviours depending on which mode the user selected. An overview of the modes was given in the previous chapter.

4.2.2 QGraphicsScene Class

This Qt class is a container for visualizing `QGraphicItems` [2]. The `QGraphicsScene` class uses an indexing algorithm to efficiently organize and determine the location of `QGraphicItems` that are rendered. The Qt documentation states that a `QGraphicsScene` can manage millions of items on one scene [2]. This feature is crucial to

Graphic. This capability allows the user to create very large graphs without negatively impacting the performance of the program. The different modes of **Graphic** require the quick and efficient identification of nodes and edges via mouse clicks. **QGraphicsScene** has various functions that filter the different key press and mouse click events. The **canvas** is a subclass of **QGraphicsScene**.

4.2.3 QGraphicsItem Class

The **QGraphicsItem** class is the base class for all graphical items in a **QGraphicsScene**. When creating a custom **QGraphicsItem** class the developer must implement two virtual public functions, the **boundingRect()** method, which returns an estimate of the area painted by the item, and the **paint()** method, which implements the painting of the object. It is also advised to implement the **Type()** method; this method returns a unique integer that is used to help identify a **QGraphicsItem** [1]. This latter method has proven to be very useful when casting a **QGraphicsItem** object to a **node** or **edge** object. The **node** and **edge** classes are subclasses of the **QGraphicsItem** class.

Node Class

The **node** class stores the necessary information to be rendered. As mentioned in the previous chapter, this class stores a diameter (in pixels), a list of edges that are incident to it, and a label should the user want to include one in their graph. Its **paint()** method will render the **node** object based on the diameter, the outline and fill colour the user has chosen. The **label** is also rendered in the paint method and placed in the center of the node.

Edge Class

The **edge** object represents the abstract edge in graph theory. An edge requires two nodes, one at each end, for it to be rendered. Like the **node** object, the **edge** object stores the necessary information for rendering. The **edge** class currently stores a colour, the two nodes it connects, the thickness of the edge (measured in pixels), the weight and the font size of the weight. As these attributes are updated the paint

method will be called to update the edge object. In addition to “getter” and “setter” methods the edge class also has an `adjust()` method that gets called when either nodes are moved on the `canvas`. The coordinates are updated and the edge will get drawn so it remains connected to its nodes.

Label Class

The `label` class is a subclass of a `QGraphicsTextItem`, which is a subclass of a `QGraphicsItem` [3]. It has a unique flag, `TextEditorInteract`, that allows the user to click on the text item and modify it. The `setTextInteraction()` method in the label class can set the text interaction flag so the labels can be enabled and disabled depending on the mode selected.

Graph Class

This class can delegate whether an event can be handled by an individual child item, such as a `node` or `edge` object, or if the event should be handled by the parent `graph` object. For example, the user can move an entire graph around on the `canvas` or just a single node, depending on the mode the program is in.

4.3 BasicGraphs Class

The `BasicGraphs` class is a library of graph drawing algorithms. The location of each node in a graph is calculated based on the size of the nodes and the graph the user specified. A graph can be drawn with or without edges depending on user input. Since edge objects have the `adjust()` method mentioned earlier there is no need to calculate edge positions. A developer can easily add a new graph to this library by implementing a new drawing algorithm in this class. Listing 4.4 provides an example of a method in the `BasicGraphs` class that calculates the node positions of a Petersen graph.

Listing 4.4: A method from the `BasicGraphs` class that calculates the node positions in a Petersen graph.

```

1 void BasicGraphs::generate_petersen(Graph *item, qreal width,
2     qreal height, int numOfNodes,
3     int starSkip, bool complete)
4 {
5     item->nodes.double_cycle.append(create_cycle(item, width, height,
6     numOfNodes));
7     item->nodes.double_cycle.append(create_cycle(item, width/2, height/2,
8     numOfNodes));
9     if (complete)
10    {
11        for (int i = 0; i < numOfNodes; i++)
12        {
13            Edge * edge = new Edge(item->nodes.double_cycle.at(0).at(i),
14                item->nodes.double_cycle.at(0).at((i + 1)
15                % item->nodes.double_cycle.at(0).count()));
16            edge->setParentItem(item);
17
18            if (starSkip % numOfNodes != 0)
19            {
20                edge = new Edge(item->nodes.double_cycle.at(1).at(i),
21                    item->nodes.double_cycle.at(1).at((i + starSkip)
22                    % numOfNodes));
23                edge->setParentItem(item);
24            }
25            Edge * connectEdge = new Edge(item->nodes.double_cycle.at(0).at(i),
26                item->nodes.double_cycle.at(1).at(i));
27            connectEdge->setParentItem(item);
28        }
29    }
30 }

```

4.4 Features Added During Development

Additional unplanned features were added to `Graphic` during the implementation of the program.

4.4.1 Undo Node Move Features

When the edit mode was implemented it became apparent that an undo feature should be implemented. A rudimentary `undo-move` feature was created to store the position history of nodes. Should the user be unhappy with a moved node they can press the `esc` key and the node will revert back to its previous position.

4.4.2 Snap-to-Grid Feature

A basic snap-to-grid feature was included on the `canvas` that creates a grid as a visual aid for more precise alignment for graph drawings by “snapping” the graphs in place. This feature can be deactivated.

4.4.3 Editing Individual Nodes and Edges on Canvas

The tab `Edit Graphs` provides the user the ability to edit nodes and edges individually. When the `Edit Graph` tab is selected it will generate widgets for each node and edge that is on the `canvas`. The user will have a similar layout as the `Create Graph` tab, such as the same button to change colour and a text field to input a label.

4.5 Challenges During Development

4.5.1 Screen Resolution and Measurements

In the initial development of `Graphic`, pixels were used as units of measurement for a graphs width and height, in the same way `Grapha` was designed. However, this created an inconsistency of graph sizes on computers with different screen resolutions. If `Graphic` was run on a computer with a high screen resolution, it would output a smaller graph than `Graphic` running on a computer with a low resolution screen. It is important for the size measurements of a graph to be uniform across all computers of varying screen resolutions. In place of pixels, inches were used as the unit of measurement for the graph drawings to provide consistency across all computers running `Graphic`. However, the Qt render methods required pixels to draw

`QGraphicsItems`. Therefore a conversion from inches to pixels was necessary. The screen resolution of the computer `Graphic` is running on is required for the calculations. In Qt's documentation it describes the `QScreen` class that provides queries to screen properties. By using this class and the methods `logicalDotsPerInchX` and `logicalDotsPerInchY` the conversion from inches to pixels could be calculated for any computer screen `Graphic` is displayed on.

4.5.2 Widget Styles across Operating Systems

In the early development of `Graphic` it became apparent that Qt widgets were styled differently depending on the operating system on which they were running on. The inconsistency among widget styles became an issue when organizing the layout of the widgets. One major inconsistency, in particular, was the difference of widget font sizes. When `Graphic` was executed on a Linux system the font sizes of all the widgets would be 14 points instead of 12 points as seen on Mac OS systems running `Graphic`. `Graphic` requires its main window to have a small size in order to display correctly on all screens with varying resolution. After considerable research without a clear explanation from Qt documentation on how to resolve this issue a different solution was designed and implemented. A private function was created and called when `Graphic` is executed to manually set the font sizes of the widgets. Although not an elegant solution, this function was able to resolve this particular issue.

4.5.3 File Browser differences

Another OS-dependent issue appeared during the final development stages of `Graphic`. Files were not being saved on Linux systems despite the fact the same files could be saved on Mac OS and Windows operating systems. After some debugging, it was discovered that the Qt file browser behaves differently on different operating systems. On Mac OS systems, for example, the file browser requires the user to select a file type and include a file name. On Linux systems, however, the file extension must be appended to the file name, otherwise the file won't be saved. After acquiring this information, a compiler flag `#ifndef` was included to check if `Graphic` is running on

a Linux system.

Chapter 5

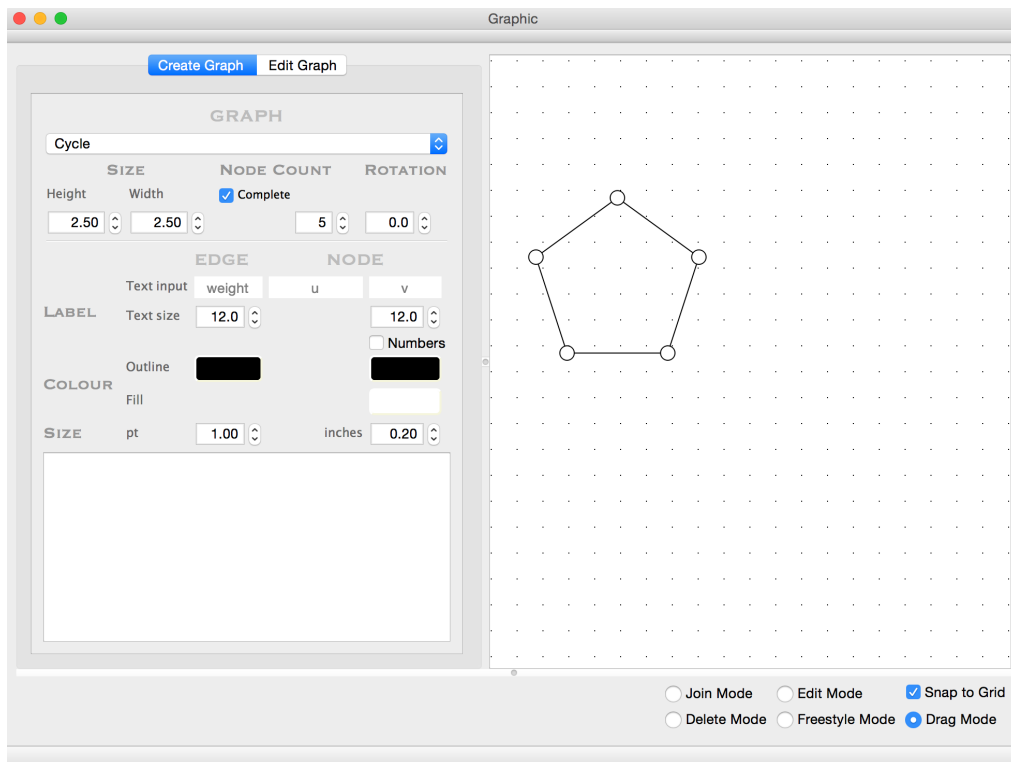
Software

This chapter will provide detailed description of the program `Graphic`.

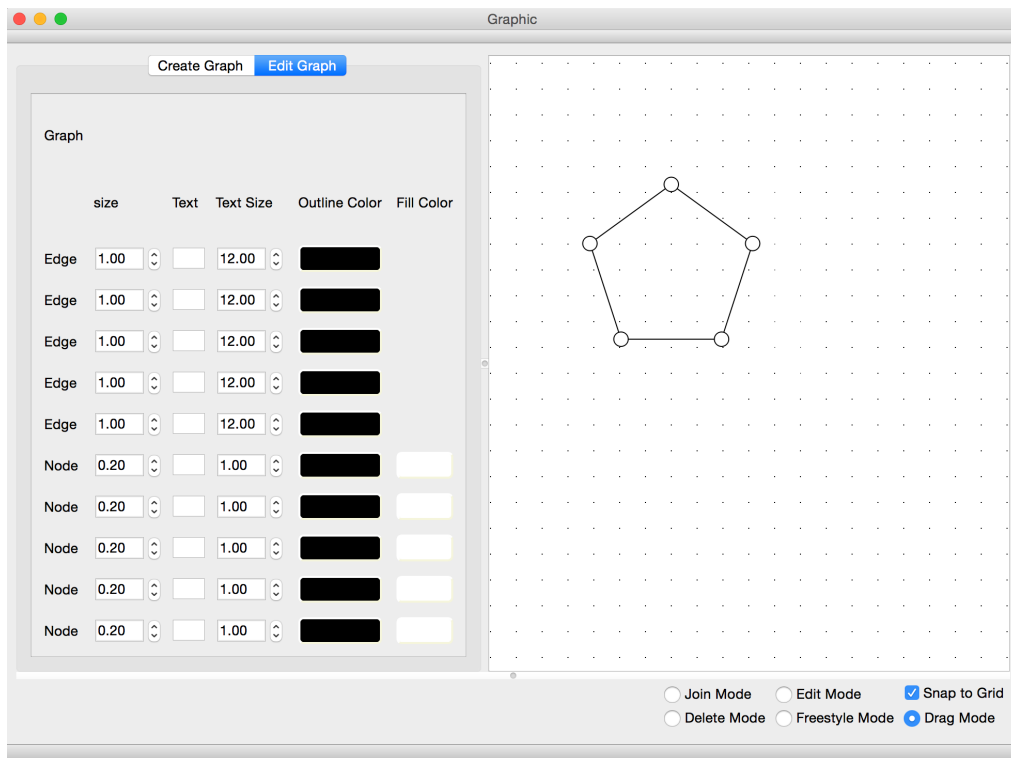
5.1 Overview of `Graphic`'s User Interface

As Figures 5.1a and 5.1b illustrate, `Graphic`'s interface received a complete redesign from `Grapha`'s original UI. As mentioned in the design chapter, `Graphic` has reduced the number of tabs used. `Graphic` has two tabs, the `Create Graph` tab that has all the options available to create a graph, and the `Edit Graphs` tab that allow the user to edit all the nodes and edges individually.

On the left-hand side of the interface there is a tab widget labeled `Create Graph`. This widget provides the user with a variety of input options to customize a graph. The graph is drawn on the `preview` object below the input fields and is redrawn automatically when the user changes any input field. Once the user is satisfied with their customized graph they can drag and drop the graph onto the `canvas` object. Once dropped onto `canvas`, the user can further edit their graph via the modes that were discussed in Chapter 3. Another feature added as a proof of concept was a snap-to-grid feature, which allows the user to align graphs easily. This is an optional feature and can be deactivated by the user.



(a) Graphic user interface with Create Graph tab selected



(b) Graphic user interface with Edit Graph tab selected

Figure 5.1: Graphic user interface.

5.1.1 Graph Input Fields

The input fields under the “Graph” label can manipulate the entire graph by altering the height (in inches), width (in inches), rotation (in degrees) and number of nodes in the graph. Adjusting the height and width of the graph is done by clicking the arrows next to spin boxes associated with those dimensions under the “Size” label. The spin boxes change by 0.5 inches per click but the user can enter a specific value. The user can adjust the number of nodes using the spin boxes. The number of spin boxes will change depending on the type of simple graph the user has selected from the Graph Type dropdown box. A bipartite graph type, for example will have two spin boxes, one for the user to select the number of nodes for top partition and the other for the bottom partition. A cycle graph, on the other hand, will only have one spin box. Finally, the user can also rotate the graph via the rotation spin box; clicking on an arrow increments or decrements the value by 1 degree.

5.1.2 Node and Edge Input Fields

The input fields below the “Edge” and “Node” labels are used to style the edges and nodes in the user’s graph, respectively. The labels below on the left hand side titled “Label”, “Colour” and “Size” are used to break down the input fields to better organize the layout. In the “Label” section, the user can enter a weight for all the edges in the graph as well as label all the nodes in a graph. Below the “Text Input” row the user can adjust the font size of the edge weights and node labels in points. In the “colour” subsection the user can select the colour for the edges by clicking on the colour button. A colour dialog window will pop up, allowing the user to pick a colour. When the user clicks “OK” the colour will change on the graph and on the button to represent what colour they chose. Node colours can be adjusted in the same way but in this case the user can change both the outline colour and the fill colour of the node. The last subsection configures the sizing of the edges and nodes. The thickness of the edges can be adjusted (in pixels) and the diameter of the nodes can be adjusted (in inches).

5.2 Create and Customize Graphs

To create a graph the user must first select from the dropdown labeled “Select Graph Type”. There is a standard list of known graphs; however the dropdown will also list graphs previously saved in `.grphc` files. Once a simple graph is selected the user can adjust the various aspects of the graph. The user may then drag and drop their graph onto the `canvas` for further editing and saving. The user can interact with the graph by using one of the several radio buttons on the bottom panel. This will change the mode in the `canvas`.

5.2.1 Freestyle Mode

The Freestyle mode allows the user to create nodes and edges directly on the `canvas`. The nodes are generated by double clicking on the `canvas`. The attributes of the node and edges (colour, size, etc.) are what were in the “Create Graph” tab before the graph was dragged to the `canvas`. A user can add edges between two nodes by right clicking on the two nodes to be joined. Should the user create an edge between two different graphs the program will combine these graphs into one. With this mode the user has the ability to create any graph. If a graph is not easily constructible from the built-in simple types then the user will be able to create and save a new graph. Thus expanding their own library.

5.2.2 Join Mode

If the user wishes to join two simple graphs they may select the “Join Mode” radio button. In this mode, the user has the option to join graphs by one node or two nodes. The joining of the two graphs are handled differently based on how many nodes the user has selected to join. If the user has selected one node from each of the graphs, upon pressing the ‘J’ key, the second node the user selected will move to the graph that contains the first node, and the two nodes are *identified*; that is, they have become one single node. The second selected node will be removed and the first selected node will collect all the edges associated with that second node. If the user

has selected two nodes from each of the graphs, the second graph will rotate and then move to the first graph. The two nodes from the second graph will be deleted, and the first two nodes will collect the edges from the second two nodes, respectively. In both cases, a new graph object will be created to be the parent of the two graphs that were joined.

5.2.3 Delete Mode

The user can click on a node to have the node and the edges that are incident to that node removed from the graph. An edge can be removed with a single mouse click. If the user double clicks on the graph the entire graph will be removed from the `canvas`.

5.2.4 Edit Mode

Edit mode allows the user to move nodes around in a graph via a mouse drag. Edges incident to the moving node will update their positions accordingly.

5.2.5 Drag Mode

Drag mode, as the name implies, allows the user to just move the entire graph around the `canvas` via mouse drags.

5.3 Save and Load Graphs

The user can press `Ctrl-S` to bring up the save dialog window. The user has a large selection of file formats in which the graph can be saved. The majority of these are image file formats such as `.png`, `.jpeg`, `.tiff` and `.svg` to name a few.

If the user saves their graph into a `.grphc` file they can use this graph later in `Graphic`. Loading graphs is done automatically by `Graphic` when the program is launched or when a new `.grphc` file is created. `Graphic` checks a specific directory, named `graph-ic`, to look for `.grphc` files. The program will add these custom graphs to the dropdown list of graphs for the user to select. To delete a custom graph the

user must locate the specific directory and delete the particular `.grphc` file or re-name it so that it doesn't have a `.grphc` extension. `Graphic` has several image file formats as well.

Chapter 6

Conclusion and Further Work

The goal of this thesis was to improve upon and add features to **Grapha**. After evaluating the possible ways of doing this, **Grapha** was entirely rewritten using a different language and framework and renamed to **Graphic**. The additional features of **Graphic** allow the user to further customize their graphs and provide additional output formats in which their graphs can be saved. The user is now able to delete individual nodes and edges, as well as entire graphs. The user can also create their own custom graphs by positioning nodes by hand and adding the edges as needed. The user can also move nodes around within a graph and join graphs. **Graphic** provides more options for the user while still providing an easy-to-use interface.

6.1 Future Work

Even with the new features added to **Grapha** there are still other improvements that could be made and additions that could be included.

6.1.1 Fixes

While most of the bugs in the program have been removed, there are still a few minor ones left. One such bug exists in the “freestyle” mode. Under certain circumstances, when a user adds an edge the edge won’t be drawn between the nodes. Instead the

edge will be drawn by itself away from the graph. The user is able to fix this by moving one of the edge's nodes around. The `adjust` method in the `Edge` class will redraw the edge in the correct position.

6.1.2 Improvements

There are several improvements that were suggested but were not added to `Graphic` due to time constraints. Fully implementing the Qt undo/redo framework would allow the user to undo every deletion, addition or move on the canvas. The user would not have to carefully monitor their actions. More options could be included in the `Create Graph` tab to further customize nodes and edges. Different fonts and font colours, for example, could be available for the node labels and edge weights. Nodes could be drawn in different shapes, such as squares and triangles, while edges could be directed. Adding curved edges would open up the option to create even more complex graphs which could also increase `Graphic`'s library. Other file options could be added to `Graphic`, depending on requests from users. The Edit mode could be enhanced to allow the graph to be rotated and resized on the canvas. The `Edit Graphs` tab could be improved upon by providing a visual aid to help identify which node or edge the user is editing. Editing graph attributes on this tab could also be added so the user could adjust the size and rotation of sub-graphs or entire graphs. Finally, the user could select what measuring system to use in the program, whether it is points, centimeters or inches.

6.2 Conclusion

The goal of this thesis was to improve upon `Grapha` by improving existing or adding features and to provide the user with a more powerful graph drawing program. To accomplish this the program was rewritten with new features. `Graphic` still maintains `Grapha`'s goals while also being a robust and powerful program. In conclusion, while `Graphic` does not implement every conceivable graph drawing tool, it successfully demonstrates that such a tool can be implemented using Qt, and it provides a tool

which can meet many needs of people who wish to create aesthetically-pleasing graph drawings.

Bibliography

- [1] Qt Documentation: QGraphicsItem Class, 2015. URL <http://doc.qt.io/qt-5/qgraphicsitem.html>.
- [2] Qt Documentation: QGraphicsScene Class, 2015. URL <http://doc.qt.io/qt-5/qgraphicsscene.html>.
- [3] Qt Documentation: QGraphicsTextItem Class, 2015. URL <http://doc.qt.io/qt-5/qgraphicsitem.html>.
- [4] Qt Documentation QGraphicsView Class, 2015. URL <http://doc.qt.io/qt-5/qgraphicsview.html>.
- [5] Qt Creator, 2015. URL <http://www.qt.io/ide/>.
- [6] Qt Documentation: Signals and Slots, 2015. URL <http://doc.qt.io/qt-4.8/signalsandslots.html/>.
- [7] Qt Documentation: Qt Widgets, 2015. URL <http://doc.qt.io/qt-5/qtwidgets-index.html>.
- [8] big think: Bjarne Stroustrup “Why I Created C++”, 2016. URL <http://bigthink.com/experts/bjarnestroustrup>.
- [9] Wolfram Alpha. Complete Graph, 2016. URL <http://mathworld.wolfram.com/CompleteGraph.html>.

- [10] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for Drawing Graphs: An Annotated Bibliography. *Computational Geometry*, 4:235–282, 1994.
- [11] cplusplus.com. C++: A Brief Description, 2015. URL <http://www.cplusplus.com/info/description/>.
- [12] Reinhard Diestel. Graph Theory. Chapter 1. pages 1–33. Springer, 2010.
- [13] Institute for Systems Biology. Biofabric, 2014. URL <http://www.biofabric.org>.
- [14] The Eclipse Foundation. Eclipse, 2016. URL <https://eclipse.org/home/index.php>.
- [15] Wilbert O. Galitz. *The Essential Guide to User Interface Design*. John Wiley & Sons, Inc., Canada, 2008.
- [16] Meurs HRM. Meurs Challenger, 2015. URL <http://www.q1000.ro/challenger/>.
- [17] B.J. Jansen. The Graphical User Interface: An Introduction. *SIGCHI Bulletin*, 30(2):22–26, 1998.
- [18] Jeff Johnson. *GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos*. Elsevier Inc, Burlington, 2008.
- [19] Michael Junger and Petra Mutzel. Chapter 1: Introduction. *Graph Drawing Software*, pages 1–3. Springer, 2004.
- [20] W.J.R. Longabaugh. Combing the Hairball with Biofabric: A new approach for visualization of large networks. *BMC Bioinformatics*, 13(275), 2012.
- [21] Kevin Matz. The Gestalt Laws of Perception and how to use them in UI Design. <http://architectingusability.com/2011/05/26/using-the-gestalt-laws-of-perception-in-ui-design/>, 2015.

- [22] Kevin Mullet and Darrell Sano. *Designing Visual Interfaces*. Sun Microsystems, Inc, Mountain View, 1995.
- [23] The Glade Project. Glade — a User Interface Designer, 2014. URL <https://glade.gnome.org>.
- [24] Helen C. Purchase, Robert F. Cohen, and Murray I. James. An Experimental Study of the Basis for Graph Drawing Algorithms. *Journal of Experimental Alorithmics*, 2(4), 1997.
- [25] Microsoft Research. Microsoft Automatic Graph Layout, 2015. URL <http://research.microsoft.com/en-us/projects/msagl/>.
- [26] Tulip. Tulip: Better Visualization Through Research, 2015. URL <http://tulip.labri.fr/TulipDrupal/>.
- [27] Lingfeng Wang and Key Chen Tan. Modern Industrial Automation Software Design. Chapter 5. pages 53–58. John Wiley & Sons, Inc., 2006.
- [28] Nicolas J. Wetmore. Grapha: Graph Generating Software. Honours dissertation, Acadia University, 2014.
- [29] yWorks. yed graph editor: High Quality Diagrams Made Easy, 2015. URL <http://www.yworks.com/en/products/yfiles/yed/>.