GRAPHA: GRAPH GENERATING SOFTWARE

by

Nicholas J. Wetmore

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

Acadia University
April 2014

This thesis by Nicholas J. Wetmore
is accepted in its present form by the
Department of Computer Science
as satisfying the thesis requirements for the degree of
Bachelor of Computer 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. Matthew Lukeman	Date

I, Nicholas J. Wetmore, grant permission to the University Librarian at Acadia University to reproduce, loan, or distrubute 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

Contents

List of Tables

List of Figures

Abstract

Many people who generate graphs for their documents are faced with a small number of tools from which to choose. The problem is that these tools seem to either be very powerful and take a long time to produce professional graphs or they are very simple and offer little or no way in which to output the graph. This makes graph generating currently very time consuming. This thesis describes the development of a piece of software named `Grapha` which solves some of these problems.

`Grapha` is a piece of software that was designed and developed to facilitate the generation of many different graphs. The user will first generate some "basic" graphs. Once he was done this, the basic graphs can then be combined together to create more complex graphs. Any generated graph is made available by the software to be output in many formats. These formats can be images, both raster-based and vector-based, or programmatic representations. In addition, `Grapha` allows a user to edit, save or load any of his graphs.

This thesis is focused on exactly what the `Grapha` graph-generating software is and what was involved in developing it. It goes over the various design challenges that were overcome while creating the software. In addition, there is a summary of what the final piece of software looks like, and how users will utilize all of its features. This thesis covers how the program interacts with the user and what type of results can be achieved from using the software. The techniques and code that were used to make the design a reality are also discussed within this thesis.

Acknowledgments

I would like to thank Dr. Jim Diamond for supervising this thesis, providing plenty of useful and helpful feedback, and being an Alpha and Beta tester.

I would like to thank Kate-Lynn MacPhail for testing the software and making helpful comments upon the design and user interface.

I would like to thank my friends and family for still being there for me when I finish writing this thesis and am with them again.

Chapter 1

Introduction And Problem

In many different areas of study, various visual representations are used in order to explain or examine certain concepts. There is a subset of these representations known simply under the title of "graphs" which consists of only vertices (nodes) and edges (lines). A graph is a mathematical concept which consists of pairs of vertices which are connected by edges. This is represented mathematically by $G = (V, E)$ where V is a set of vertices and E is a set of edges. Combined, the two sets allow the graph to represent a set of relationships. The relationships are represented by the edges and the objects being related are represented by the vertices.

Within a graph a vertex does not need to be connected to another vertex; such a relationship is represented by the absence of an edge. Also any vertex could be connected to any other vertex via an edge. All of these connections, when combined together, make up the graph. Edges, when represented visually, are lines which connect two vertices together while vertices are visually represented by circles. The combination of circles and lines represent the entire graph in a visual format. It is this visual representation which is the focus of this thesis.

The edges between nodes, as well as the vertices themselves, can all be labelled. A label upon an edge usually represents a weight (or a cost) which is associated with that edge. This, for example, could mean that there is a cost associated with travelling from one vertex to another, and the label is the quantity of that cost. A label upon a node is used to name and identify the labelled node. Often every node

in a graph is named with a unique name. However, edges do not always have an associated weight.

Visualizations of the sort described here are most common in graph theory. The study of graph theory is useful for various mathematical, biological, physical, and computational areas of study. Some applications for these graphs include: computer networks, databases, molecular chemistry, condensed matter physics, and many more areas [3, 17, 5]. The quick and simple generation of visual representations of such graphs using computer software will be the focus of this thesis. For the purpose of this thesis, the word "graph" is used to refer to the visual representation and not the mathematical concept.

1.1 Making Life Easier

Software development today is mostly driven by a goal to make users' lives easier. The point of developing the software for this thesis was to make life easier for the person who generates graphs on a regular basis. Such a person, if using a non-graph oriented graphical editor (such as MS paint or Gimp), will find himself with a lot of work ahead of him just to create one graph. The problem with this is that even after a lot of hard work, the graph still might not come out looking professional. The resultant graph's quality will be tied to the skills of the person generating the graph. If the person has little to no experience making pictures on a computer, the result could look unprofessional and may even be difficult to read.

Software whose use is to make professional-looking graphs already exists (see related work, Chapter 2). However, many of these implementations are either difficult to use, (an example being `Graphviz` [16]), or do not have fully featured output options, causing them to be useless for certain people. The biggest difficulty with `Graphviz` is the fact that graphs are not created or edited visually. The software makes use of a command line or text box in which the user inputs the code for his graph. It appears as if the development of this software was not focused on the user experience but instead was focused on developing the languages that represent the graphs (language examples include `dot`, `neato`, and `dotty` [15]). Such software also requires

more time than necessary to generate basic graphs, which may need to be generated extremely often. This means that there is a gap in graph-generating software. There is no quick and easy option which allows for both quick generation of basic graphs and outputting the created graph in many different formats.

Another downfall of such dedicated software is that it can sometimes require a set up or installation time. This set up time makes the software tough to use for someone who may not generate graphs very often, but does wish to generate a graph every once in awhile. In order to eliminate this set up time we can conceivably offer an on-line solution which allows the user to use it when he wishes. The benefit of on-line software is that the user does not need to worry about maintaining it or installing it on his computer. This presents another problem, which stems from the idea of on-line applications: they can only be used on-line.

There are already on-line software solutions which generate graphs (such as `Illuminations Graph Creator` [9], `Creately` [1], or `GraphJS` [8]). The problem is that these programs do not interact with a personal computer in the most useful of ways. This is evident in the fact that many such solutions to graph generation do not include a way to usefully output the graph that you have generated. Many also do not allow you to save your graphs locally to your computer, and many do not provide a way to use the software when you do not have an internet connection (covered in depth in Chapter 2). For someone who generates graphs on a daily basis, he may need the ability to generate a graph sometime when he is not able to access an internet connection. Throughout this paper the technologies that are used which allow the graph-generating software created for this thesis (named "`Grapha`") to overcome these limitations will be addressed.

Finally, there are very few programs which can be run on all of Windows, Android, Mac OS, Linux, and iOS, but to be fully accessible `Grapha` was developed with all of these platforms in mind. It not only runs on each platform but the difficulty of using it on mobile platforms is not such that `Grapha` becomes tough or challenging to use. This is discussed more in Section 5.1.

1.2 Problems With Generating Graphs Quickly

1.2.1 The Goal Of Grapha

The software solution (named "`Grapha`") written for this thesis is not meant for powerful customization or lengthy and specific graph generation. Instead, `Grapha`'s purpose is to provide the quickest possible way to generate and combine basic graphs. In this manner, anyone using `Grapha` who wishes to generate basic graphs on a regular basis will not need to spend a lot of time on them. The graphs generated by `Grapha` are symmetrical (with evenly spaced vertices), easy to read, and professional-looking. The attributes of the graph (its size, number of vertices, the way that the graph is laid out, its edge weights, and its vertex labels) are all easily customizable and `Grapha` will automatically generate a new graph based upon these attributes. In this way `Grapha`'s use is to generate specific "basic" graphs and save the time of the person who generates such basic graphs regularly.

The final goals of `Grapha` will be discussed fully in Section 3.1 after some discussion of other software solutions in Chapter 2.

1.2.2 Basic Graphs

Generally, in this thesis the term "basic graph" refers to a graph which is used by many graph theorists frequently (some sample basic graphs are shown in Section 1.6). The idea of what the term "basic graph" refers to is not set in stone. While some people may believe that a bipartite graph is common and basic, others may rarely generate bipartite graphs (or may think of them as complex non-basic graphs). This, combined with the fact that there is a limitless number of graphs that users may want to generate, leaves us with a problem: if we want to allow quick generation of basic graphs, we will need to code a large repertoire of graphs.

For the purpose of this thesis, a basic graph is a graph type which is built into `Grapha` (that is contained within its repertoire of basic graphs). `Grapha` has been designed to not only hold a repertoire of graphs for quick generation, but also to allow increasing the size of this repertoire in a relativity easy manner for a compet-

ent JavaScript programmer. It should even be possible for someone who is familiar with object-oriented programming, but who knows nothing about the programming language "JavaScript", to, with a little more time, add a new graph to the repertoire. The programming style used to enable this functionality is discussed further in Section 3.5.1.

1.2.3 Manipulating Graphs

In order to remain simple and quick, `Grapha`'s user interface requires a very small number of clicks to actually generate a graph and acquire it as output. Although quick and easy, this simplicity gives the user less power and influence on the final graph product. The user does not have access to manipulate the small details of the final graph. However, there are other solutions which allow every detail of the graph to be manipulated and specified. `Grapha`'s purpose is not to replace these software solutions but to provide generated graphs quickly. Furthermore, due to the way in which `Grapha` stores a graph object, the graphs that it generates are not confined to be used only with `Grapha`. `Grapha`'s outputs allow generated graphs to be useful to other programs as well.

The swiftness of `Grapha`, however, does not leave it crippled during the moment that the user wishes to specify a small manipulation. `Grapha` comes equipped with a wide range of ways in which to output the graph. The user can output it as one of many raster-based image types or as one of a number of vector-based image types. In the latter case it gives the user the code which would be used to generate the image in his program of choice. In this manner `Grapha` allows a user to take the graph that he quickly generated, and manipulate the graph with another piece of software. This gives `Grapha` the power to be used alongside a user's graph-drawing program of choice. It allows a user to save time generating a graph (especially if it is a basic graph which he will generate variations of many times over), as well as detailing that graph to fit his current scenario.

Another similar problem should be apparent here: how do we know which output formats users will wish to use? For raster-based images, the answer is quite simple:

cover all of the basics and let future programmers change it from there. For vector-based images, the answer is not as simple.

It appears that every vector-based method to draw graphs comes with its own standard, sometimes even its own code. This means that, for every user program, we will need to have an output that allows the graph to be translated into a different code and standard. The solution to this problem is similar to the solution to the basic graph's problem. We need to provide a basic repertoire of the most popular vector-based outputs and allow other programmers to expand this as needed. By doing this we open the doors to every vector-based program. As long as there is someone willing to write a translation module for `Grapha`, that person will be able to expand the software's repertoire and it will be able to support any program. This is possible since `Grapha` was designed modularly, and with such expansion in mind.

1.2.4 The Details Of A Graph

No matter how swift we make a program, if there are no controls at all, then the program is essentially useless. `Grapha` does allow a user to tell it the specifics of the graph that he wants generated. Using these controls, the user can specify which basic graph type that he wants to generate. After that has been selected, he inputs the graph's size, the size of the graph's text, the number of vertices, as well as the labels that each vertex and edge should have. This allows the user to fully customize the look, scale, and labels of any basic graph. The only edits that cannot be made in `Grapha` itself are the removal, addition, styling, or repositioning of edges and vertices (see Chapter 5 for further discussion).

1.3 Compound Graphs

For the purpose of this thesis a "compound graph" is defined as a graph in which there are multiple basic or compound graphs joined together. These graphs form a larger more complex graph when joined (for an example, see Figure 1.1). In fields where graphs are used it is not uncommon to want to connect multiple graphs together.

Once graphs begin to be connected together however, they suddenly become more complex and more difficult to generate. In order for a user to combine multiple graphs in a standard graphical program the user may need to perform a lot of copying and pasting. He may need to resize certain parts of his new compound graph and he may also wish to rotate the basic graphs held within.

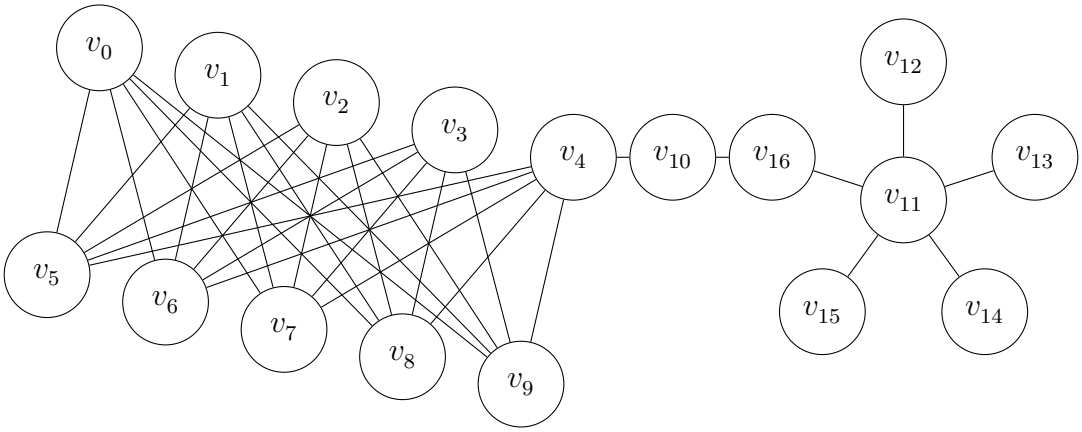

Figure 1.1: A Compound Graph

These compound graphs can be extremely useful. As pictured in Figure 1.1 a compound graph can model many connections which, normally, would be difficult to otherwise visualize.

Once a user has generated a basic graph, outputting that graph is not his only option. Instead, if the user has a more complex graph to make, he can begin to expand one graph by attaching other graphs (that he has made) to it. This can allow a user to (very quickly) expand a graph to model a more complex situation. The ability to attach multiple graphs together however, comes with a set of problems.

The first problem is, after two professional-looking graphs have been combined, the compound graph needs to look just as professional. The second is that the user now needs additional controls to allow the complex graph to appear the way that he wants. We will need to define a few more ways in which the user can interact with complex graphs so that he can make them appear the way that he wishes. Finally, we need to figure out how to connect the two graphs and how to generate the proper outputs for each graph. These problems are addressed and discussed in Section 4.2.3.

As with the basic graphs, each complex graph can be saved or output in any manner that the user desires.

1.4 User Interface

In order for `Grapha` to be usable it must have a convenient and interactive interface. This interaction will determine the usefulness and success that `Grapha` will achieve. Bearing in mind that `Grapha` is written in JavaScript, its primary running environment will be inside a web browser. This allows `Grapha` to be accessed on-line by a user as long as it is being hosted by a web server. Despite having these web application roots, though, `Grapha` can also be run locally. For any locally running application to be successful it will need to have access to the same functions that any other local application would have (these include saving and loading files, as well as taking user interaction). This means that developing a user interface for `Grapha` is not very different from developing a user interface for a desktop application. This is achieved through the use of the HTML and CSS languages, as discussed in Chapter 3.2.

In order to create a usable interface, the interface was given to some users for testing. It is one thing to claim that `Grapha` is quick to use or useful and another to have users actually produce results with the program. Throughout the development process, a few users were engaged to ensure that the graphical user interface did not become detached from the project. The user interface uses the HTML and CSS framework called `Foundation` (created by Zurb [23]). The framework provides a suitable default look and feel, as well as most of the styling of `Grapha`.

The layout and user interaction was programmed specifically for `Grapha` in hopes that it would allow the flow of creating a graph to be smooth and easy. The layout also allows for mobile platforms to render the GUI in such a way that the program is still fully usable, without much extra user effort.

1.5 Graph Knowledge

Most of the underlying knowledge about graphs that the reader needs to know was outlined in Sections 1.2.2 and 1.3. The reader will also need to know about some of the different basic graphs. Section 1.6 contains examples of each basic graph referenced in this thesis. Furthermore, an understanding of raster- and vector-based graphics, as well as the differences between them, will be required.

There are a few ways to go about generating a picture or graphic. Two of the most popular types of graphics are raster and vector graphics. A raster-based image or graphic refers to an image which is represented in the computer by a set of pixel colour values. When a raster-based image is rendered the information of each pixel is turned into a coloured dot. These dots (or pixels) are then arranged together to generate the image. A vector-based image is an image which is represented by a set of rules or commands. When such an image is rendered the commands are executed and the image is generated.

For example, if a user wants to draw a four-by-four green box with a blue line on the top, a raster-based image would contain the information for four blue pixels, and then the information for 12 green pixels (this is a very simplified explanation, but it is sufficient for the purpose of this paper). When a program is asked to render said picture, the pixels will simply be drawn on the screen one by one until all of them have been drawn. An equivalent vector-based image contains a command for drawing a green box and a command for drawing a blue line across the top of the green box. When it is rendered, the commands are executed, and the pixels are generated, then drawn to the screen.

Raster-based images are usually large for large images, while vector-based graphics are usually smaller. This is due to the fact that a single command in a vector-based graphic may tell a program how to render hundreds or millions of pixels, while the raster-based image would need to specify each pixel's properties individually. A vector-based image, however, can take longer to render to the screen, because all of the commands need to be executed each time it is rendered, before the image is able to be shown by the computer. Vector-based images are also scalable, whereas raster-based

images are not. Currently, raster-based images are also easier to use and are more widely supported. This is true because most operating systems are pre-bundled with raster-based image viewers, however fewer are accompanied by vector-based image viewers.

1.6 Basic Graph Examples

The basic graphs shown in Figures 1.2 through 1.9 will be assumed to be known throughout this thesis. Each example of a basic graph shows a specific choice of parameters. For example, Figure 1.2 shows a bipartite graph with four vertices in the first partition and five vertices in the second partition. These two numbers are not constants and can be changed to any other positive integers. A similar idea applies to the other examples shown here.

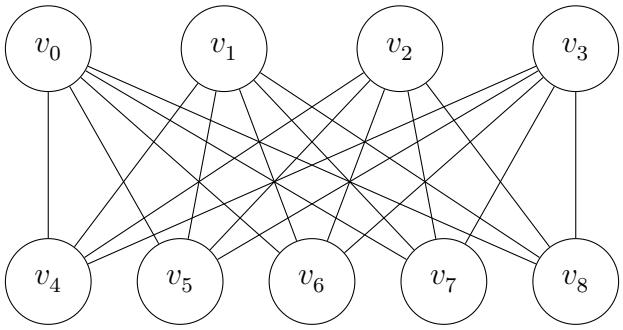

Figure 1.2: A Bipartite Graph

1.7 Developing Grapha

Throughout this thesis, the theme will be a walk-through of exactly how `Grapha` was conceived and developed. The technologies used to create the software will be discussed and the programming tactics and decisions will be thoroughly examined. The next chapter outlines some of the background knowledge needed to continue with this thesis. Starting in Chapter 3, the process of engineering the software from the ground up will be discussed.

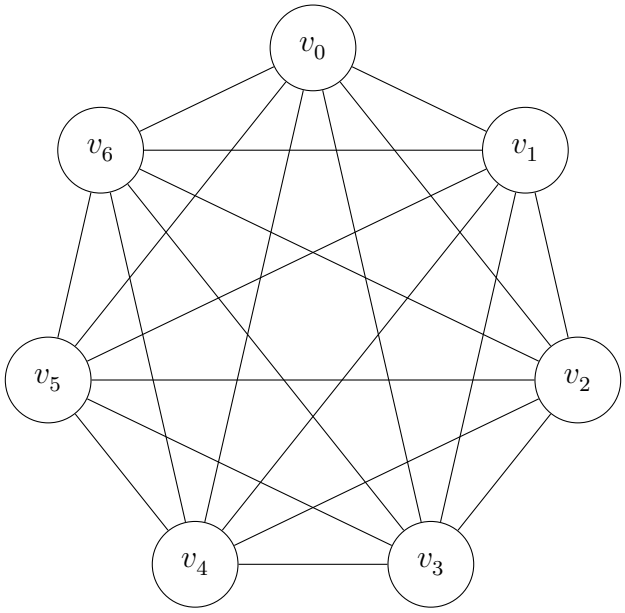

Figure 1.3: A "Round Layout" Graph

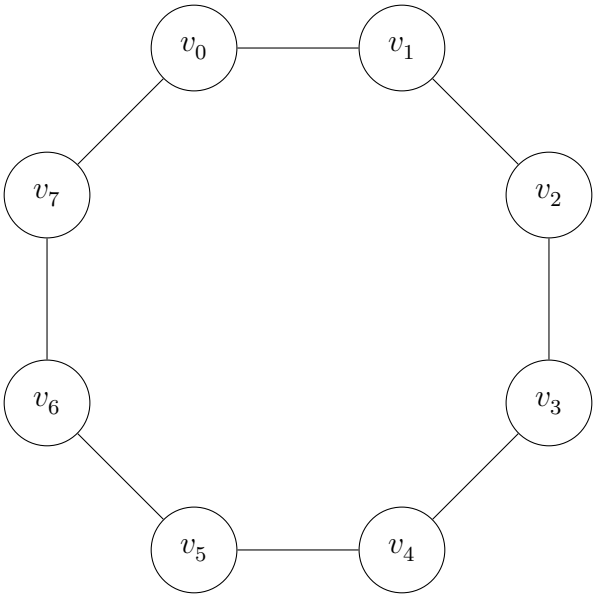

Figure 1.4: A Cycle Graph

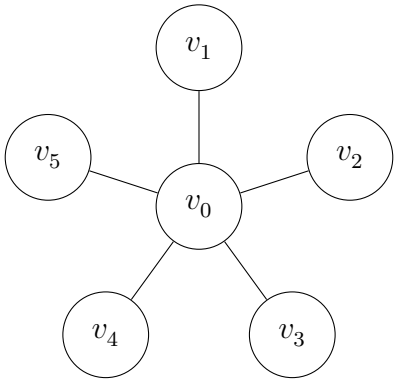

Figure 1.5: A Star Graph

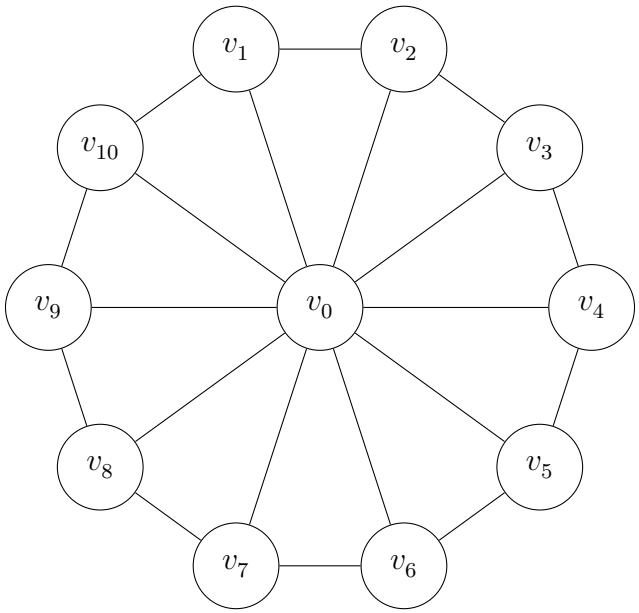

Figure 1.6: A Wheel Graph

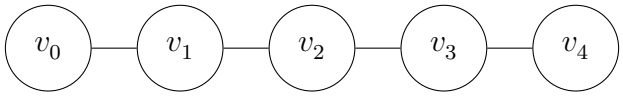

Figure 1.7: A Path Graph

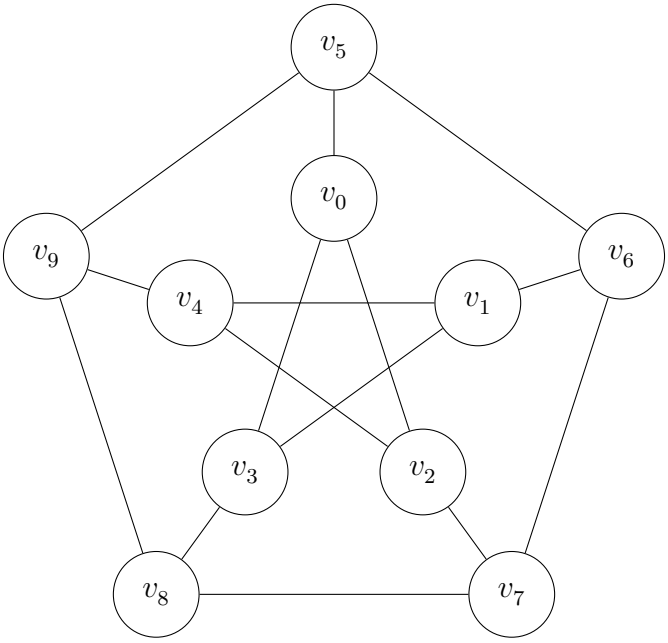

Figure 1.8: The Petersen Graph

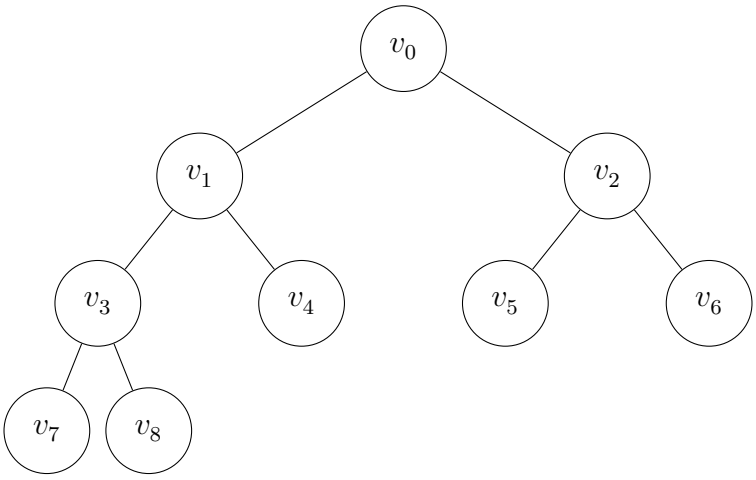

Figure 1.9: A Balanced Binary Tree

Chapter 2

Research And Related Solutions

In the graph theorist's world `Grapha` is not the first or only piece of graph-generating software to exist. Many more software solutions exist which have goals of generating graphs for the user. This chapter will focus on some popular solutions to graph generation and how they are both similar and different from the software outlined in this thesis. The chapter will also talk about the various research which was required to create `Grapha`.

During a search for graph-generating software, users will find that most of the solutions they come across will fit into three different categories.

The first is an approach which comes from a simple point of view. In this approach generating a graph is done via a small and simple point and click interface, with a few options to back it up. In these software solutions, users are given few options (usually essentials) and a limited way to output the graph that they have created. In addition, there is very little automation in the software which causes a user to have to specify everything manually. Examples are: `ILLUMINATIONS Graph Creator` [9] and `Creatly` [1].

The second approach is to make the software and its functions focused upon graph theory. This development approach allows the software not only to create and edit graphs but it also allows the user to run graph theoretic algorithms on the created graph. These tools usually take an average (relative to other graph-generating programs) amount of time to learn and use. They may also require that

the user learn a specific language for dealing with the graphs (such as a computer programming language or a graph description language). Good examples are `Sage` (which is primarily used for math but has graph functions) [4], [22] and `GraphTea` [21].

The last paradigm is to have an extremely complex and powerful piece of software. The user is expected to spend a very large amount of time with the software, both learning and generating graphs. This allows the user to output the graph in a variety of ways as well as specify every detail of the graph. In most cases the software also gives the user the ability to begin interacting with the graph and explore it with advanced two dimensional or three dimensional visuals. Similar to the second type of graph software there is usually an ability to perform graph theoretic algorithms on any graph in the program. Good examples include `GraphViz` [16] and `Gephi` [7].

The first piece of software I will look at is very simple. ILLUMINATIONS developed it and named it, appropriately, "`Graph Generator`" [9]. This piece of software allows a user to create a graph from many places as it is `Flash`-based and hosted on a website. The user must have an internet connection and a `Flash`-enabled device in order to run the software. Other than that, the software is extremely simple to use. It can take some time, however, to create intricate graphs due to the fact that each vertex and edge must be manually positioned and specified. In addition, the program has no output methods; instead of being a resource for generating graphs to use, it appears to be a resource simply for building them, performing a few small algorithms on them, and then moving on.

A typical use of the software is shown in Figure 2.1. This piece of software was chosen to be discussed in this thesis because it is a good representation of entry level graph-generating software. The use of discussing this piece of software is to outline a group of software solutions to graph generation which all have similar properties. There are similar pieces of software in the same group as `Graph Generator` which are powerful and can create any graph that the user wishes, but they lack speed and the ability to transfer those graphs elegantly to another program.

`Creatly` is another example of such a program; this piece of software does not focus specifically on graph generation, however, generating graphs is one of its capabilities. It also lacks the ability to automate any graph generation and can take quite a bit

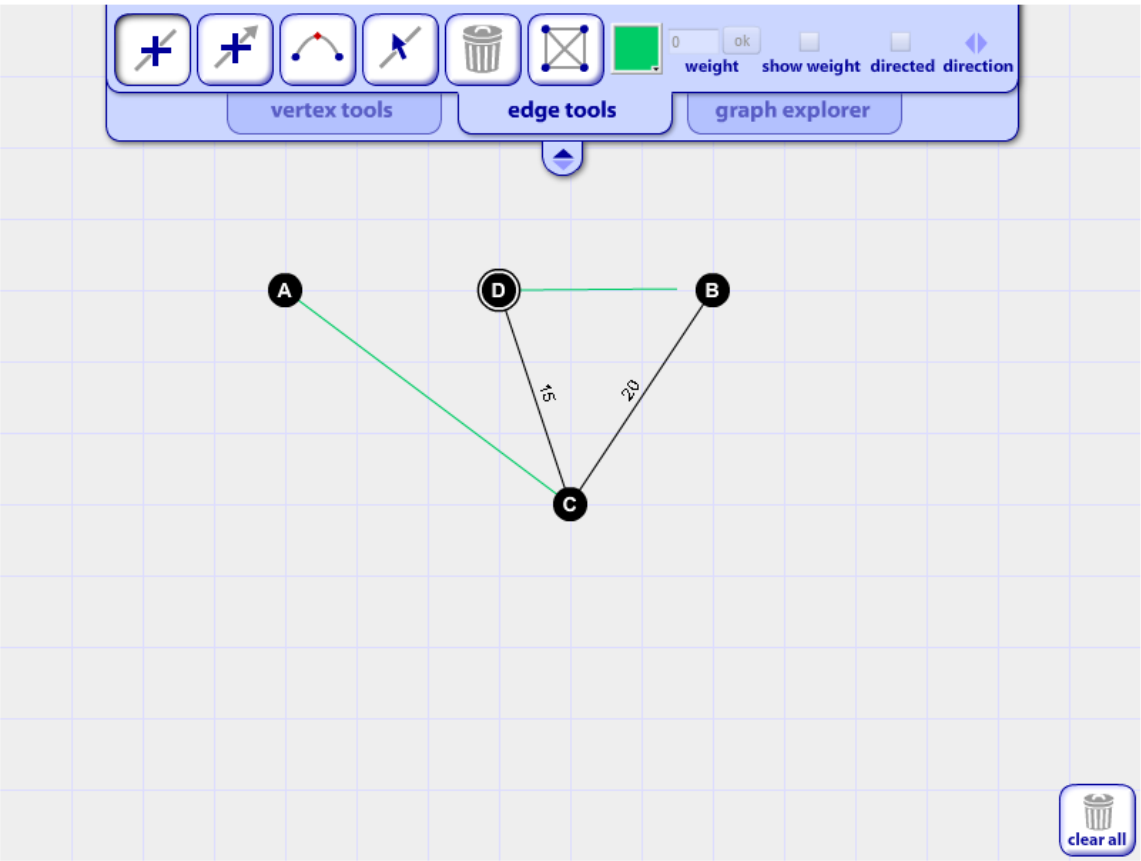

Figure 2.1: ILLUMINATIONS Graph Generator

of time to create a usable graph. Although it does have more output methods than `IMMUNIMATIONS Graph Generator` the fact that it is not focused on generating graphs makes it fall short in that area.

The next piece of software that will be looked at is `GraphTea`. `GraphTea` is perhaps the closest piece of software to `Grapha` that could be found. `GraphTea` allows a user to generate a graph and it also has a small repertoire of basic graphs which can be automatically generated. `GraphTea` is also powerful in that a graph can be customized. A graph can be specified and dragged to desired dimensions. This means that creating a professionally drawn graph can take quite a bit of time. Also, learning the software takes a small amount of time. `GraphTea` also focuses primarily on graph theory algorithms which it is very adept at solving.

Overall, the GraphTea software fits into the second category of graph-generating software. It does take time to learn and it takes time to draw professional graphs but it will not run on-line and it cannot be accessed by any computing device (computer, phone, tablet, etc.); it must be installed on Windows, Linux, or Mac OS. This increases the start up time required to get the program running. Also, the fact that it needs to be installed means that it cannot be used casually and on the fly when the user needs a graph generated quickly. It allows for outputting a graph in a raster- or vector-based language (such as LaTeX) but it does not include SVG. Figure 2.2 shows an example of a graph created in GraphTea.

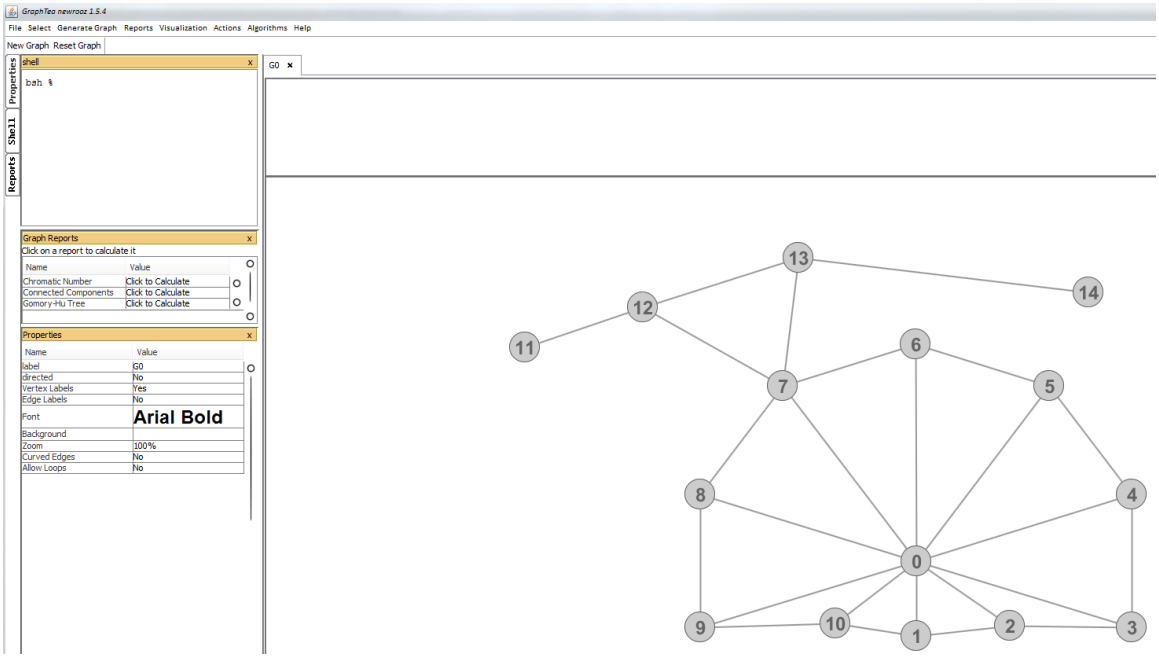

Figure 2.2: GraphTea Software Use

The final pieces of software which will be reviewed are GraphViz [16] and Gephi [7]. Graphviz is a commonly used application for generating graphs and relies upon graph description languages, in which the user writes his graphs, to generate the graphs. Its most commonly used graph description languages are Neato and Dot [15]. GraphViz and Gephi can both be used very powerfully to produce any graph that the user can imagine and in the case of Gephi can even perform multiple complex algorithms on

them. These software solutions are both very powerful and contain plenty of features and extras, but with these features comes complexity.

Both pieces of software take a very long time to learn and can take even longer to produce a nice looking graph. `Gephi` seems to be the more recently developed solution and allows very advanced visualization of the user's created graphs. It can also output graphs into PNG, SVG, or PDF. In contrast, `GraphViz` seems older and less refined. It requires users to mainly use commands and write their graphs out in a graph language in order to achieve their desired graphs.

All of the solutions mentioned here can make the graph-generating user's life easier, and each solution has positive points and negative points unique to them. These points are summarized in Table 2.1.

Table 2.1: Tables of functionality for researched software.

(a) Supported platforms and portability

Software	Desktop	Mobile	Installed	On-line	Off-line
`Graph Creator`	✓	✗	✗	✗	✓
`Creately`	✓	✗	✓	✓	✓
`GraphTea`	✓	✗	✓	✗	✓
`GraphViz`	✓	✗	✓	✗	✓
`Gephi`	✓	✗	✓	✗	✓

(b) Outputs

Software	Raster-Based Output	LaTeX Output	SVG Output
`Graph Creator`	✗	✗	✗
`Creately`	✓	✗	✓
`GraphTea`	✓	✓	✗
`GraphViz`	✓	✗	✓
`Gephi`	✓	✗	✓

(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

Chapter 3

Design And Software Engineering

This chapter will focus on the design and early development stages of the software named "`Grapha`". Furthermore, this chapter will focus upon the research used to implement these designs and the methods used to realize the design possibilities.

3.1 Goals of Grapha

Having discussed the positive and negative points of the visual graph-generating software which is currently accessible to users, I will outline exactly what `Grapha`'s goals are.

`Grapha` is not meant to compete with the other software solutions available to a user. Instead, `Grapha` will focus on having many features to accompany generating graphs quickly and automatically. `Grapha` will focus on achieving the goals outlined below.

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.

Some of the software solutions mentioned met many of these requirements. There was no single piece of software which met all those goals at the same time. These goals propose `Grapha` to be a piece of software which is quick, easy, useful, and portable, all at the same time. `Grapha` focuses on the swift generation of graphs and the quick combination of those generated graphs. None of the researched software solutions carry that ability in addition to the other desired traits. `GraphTea` is the closest solution to `Grapha` but it lacks the simpleness and the various output methods that `Grapha` has (refer to Table 2.1).

In order to achieve these goals the development paradigm of agile development was utilized. This development paradigm allowed `Grapha` to be user-focused. Users were interacted with throughout the design and implementation of `Grapha` and the goals were updated according to the users' feedback. Agile development allowed `Grapha` to remain flexible enough for users to make requests and see them implemented. This allowed `Grapha` to change during development and become easy and quick. The idea of how to make `Grapha` quick and easy to use was based upon the users' experiences with the software.

3.2 Programming Languages

In order to be prepared for reading the content contained within this chapter, there are a few things that the reader will need to know ahead of time. The requisite knowledge includes knowing some general information about graphs and how they look (all covered in Chapter 1). Furthermore, a bit of general information about the various computer programming languages which have been used to create the `Grapha` is required. These languages include JavaScript, HTML5, and CSS.

3.2.1 HTML5

HTML5 is the newest complete standard for the markup language from the W3 consortium [10]. HTML5 (which stands for Hyper Text Markup Language) is the main markup language which is used on the internet for telling a browser how to display a web page. Upon a request for a web page, servers provide HTML, CSS, and JavaScript to a user's browser. The browser then interprets the code and lays out the page.

HTML5 (as opposed to the previous version of HTML) includes new tags for displaying media (such as music or video) as well as displaying dynamic graphics. Of most importance for this thesis is the new tag named "`Canvas`", which has been utilized in the `Grapha` graph-generating software. The `Canvas` tag allows for raster-based image creation in a web page. In order to perform this image creation JavaScript is used to manipulate the contents of the `Canvas` tag in a pixel by pixel fashion. This image creation is how `Grapha` displays an image to the user and allows a user to save his images as a raster-based output (discussed fully in Section 4.3)

Not every browser is fully compliant with the HTML5 standard. It is also not uncommon for different browsers to implement different HTML elements in different ways, or to expect unusual syntax during interpretation of some code. This can make developing HTML difficult, as the programmer must always be mindful of the different browsers that his software will be run on and exactly what those browsers support and how they support it.

3.2.2 CSS

CSS (which stands for Cascading Style Sheet) is the most popular language for the styling of HTML on the internet. The styling of an element is a catch-all term which refers to all of the various attributes which the element has. It also refers to the effects that those attributes have upon the look of the element. We can specify the element's height, width, font colour, font size, margins, borders, padding, background colour, placement and much more.

When a browser receives a page of HTML, the HTML may request the browser to also obtain CSS. The CSS can also be specified within the HTML page, in which case it is not necessary to obtain CSS from a server separately. When a browser obtains the CSS from a server, either from a CSS page or through the HTML, the browser then parses the CSS and applies the styling to the HTML that has been rendered. In a perfect situation the HTML is never perceived as being styled in any other manner other than that specified by the CSS on the page. This allows a web developer to more easily lay out his content and to style every element of the page. This allows the web developer to make the page inviting and easy to use from a user's perspective.

CSS has a lot of specific features which make it difficult to summarize. If you wish to learn CSS, the best way to familiarize yourself with it is to lookup or read some tutorials (a good source for this is w3schools [25]). It is not necessary to read such tutorials for this thesis. In summary, CSS is used for styling an element's colour, border, font, margin, position, background, and much more.

In order to aid with the development of CSS and HTML, frameworks are often utilized. A framework is a package of pre-written code which sets up some common code snippets for use by the programmer. This allows the programmer to write code more quickly and it reduces duplication of work. The frameworks used by web developers set up a lot of the CSS ahead of time for the developer. They also sometimes include a few custom HTML tags. During the development of `Grapha`, a framework by the name of "`Foundation`" was used for these purposes. Developed by Zurb, `Foundation` is free to use and helps in developing front end web pages [23]. It allows a programmer to create a user interface quickly and cleanly. To use the framework, a developer must first set up a colour profile for his application. After that,

the developer is able to make use of the extensive library of `Foundation` styles which are not only professional and clean looking, but also user-oriented and responsive.

`Foundation` helped with a lot of the user interface and styling details of `Grapha`. It allowed development time to be focused upon programming, functionality, and user goals. Instead of a lot of time being spent on styling and design details, a more appropriate amount of time was spent on them, allowing development time to be spent enhancing the software's capabilities.

3.2.3 JavaScript

The first thing to understand is exactly what JavaScript is. However, the language that we call JavaScript may not refer to a concise thing. Every company which implements some version of what we call JavaScript is really implementing something different. What the companies are really doing is creating a language which is compatible with ECMAScript (ECMA stands for European Computer Manufacturers Association), but may not be JavaScript. ECMAScript is the standard upon which all other companies develop their ECMAScript-like (JavaScript) language [18]. ECMAScript's standards tell the developers of any JavaScript-type language what to implement and how to do so [13]. Every company names this language that they developed something different. For example, Firefox and Chrome use a language called JavaScript, Internet Explorer uses JScript, and Opera simply uses ECMAScript. This means that 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 and are based upon ECMAScript.

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. These engines are developed independently (except for their common compatibility with ECMAScript). The various engines used by browsers may differ slightly on exactly which code they

support and the functionality of various pieces of code. This means that while programming JavaScript, sometimes the developer needs to consider where it is going to be run. The programmer can then adapt to ensure that all relevant platforms run the code in the same manner.

JavaScript is an event-oriented scripting language which is used primarily for web development. Saying that JavaScript is event-oriented is similar to saying that JavaScript is driven by events or occurrences. This means that JavaScript responds primarily to user interaction or other events that occur (such as the completion of the web page loading). The event-oriented paradigm also means that when JavaScript updates the page or changes things in the browser it performs all of those changes at once. This is usually performed at the end of each function.

Although it is described as being event-oriented, JavaScript is also object-oriented and supports functional programming. It is not compiled, but rather interpreted by another program. JavaScript, although having the word "Java" contained within its name, is very different from Java. Similarities to Java include, but are not limited to, naming conventions, object-oriented programming, and portability. JavaScript has syntax which is heavily influenced by the programming language named "C". This thesis will focus on the usage of JavaScript in `Grapha`, where it is used alongside HTML to create an interactive web page.

3.3 Cross-Platform Code

3.3.1 Language Choices

One of the goals of the `Grapha` software was that it would maximize portability. In order to achieve this, `Grapha` must be able to be run on a maximum number of devices. Furthermore, it was desired that `Grapha` would also run as a web page hosted on the internet. Using this web page, devices which do not allow the user to install programs would also be able to run `Grapha` quickly and without set up.

With both of these goals in mind the choice of programming languages to use was obvious. HTML, JavaScript, and CSS are languages which work well together and

can be run on any device which has a JavaScript-enabled browser. Also, as long as all of the required code can be placed within one file, the user can access it via a web page or save it and run it off-line. Since JavaScript-ready browsers are available on Mac OS, iOS, Windows, Android, and Linux, creating a web page which runs solely using JavaScript, CSS, and HTML allows a user to run that page everywhere that he can have Chrome, Firefox, Safari, Android Browser, or even Internet Explorer installed.

Despite `Grapha`'s portability, it cannot be run absolutely anywhere. `Grapha` utilizes bleeding edge features of both `HTML5` and `JavaScript` in order to allow its various functions to run smoothly and seamlessly. Due to `Grapha`'s reliance on a web browser the user must be running a current version of any web browser (this includes Internet Explorer 9.0 and up, or any current version of Firefox (5.0.1 and up), or Chrome (9.0 and up)). This however, does not limit the usage of `Grapha` as long as the user is willing to upgrade to an up-to-date browser. The most popular browsers are available on any device for free. Many devices even come with one pre-installed.

In order for a user to run `Grapha` off-line on his own computer, he can simply right click on the web page, and click `Save as...` or his operating system's equivalent function. It is valuable to note here that not every operating system has this functionality. Doing this will create a local HTML page which can be run in any browser of the user's choice. This means that both the web version and the local version of the page will run and behave identically. Consistency is valuable to users, so that they do not have to re-learn a program the moment that they wish to use it slightly differently. `Grapha`'s use of JavaScript, HTML, and CSS preserves the user's experience from one device to another therefore reducing the learning curve of the software.

In order to obtain `Grapha` a user must either have access to the internet, or some way to transfer the saved `Grapha` file onto his computer. Without either of these things, the user would not be able to obtain the software.

3.3.2 Saving and its Challenges

In conjunction with using three languages, `Grapha` has also been tested for compatibility across multiple browsers. `Grapha` will work on any up-to-date version of Chrome, Firefox, or Safari and it will also work on Internet Explorer 9.0 and greater. In order to achieve this, some of the JavaScript had to be sculpted to better fit all of those browsers. The most notable difference was with the way in which browsers both save and load items to their cache or to the computer.

In order to save or load graphs locally `Grapha` makes use of the object named "`localStorage`" in JavaScript. The `localStorage` object is a way to save key-value pairs into the browser's cache for use later. You can save and load, to or from these objects using the methods `localStorage.setItem(key, value)` and `localStorage.getItem(key)`. The keys and values used must be strings. This means that any object which is to be saved into `localStorage` needs to be converted into a string and needs to be parsable back into its original object form. In order to do this JavaScript's JSON (JavaScript Object Notation) object was used. JSON is both an object and a concept in JavaScript. It is the concept that any JavaScript object has a string representation which is easy to parse and it is an object which has useful methods for the stringification and objectification of various objects and strings. The methods used to do this are called as follows: "`JSON.stringify()`", and "`JSON.parse()`".

Using these methods, `Grapha` is able to save an array of graphs into local storage by stringifying them first using `JSON.stringify(object)`. Then whenever the user issues a request to load such a saved graph, `Grapha` uses the "`JSON.parse()`" command to convert the string back into an object.

This does not run as smoothly as one might expect, though. As it turns out, the JSON representation does not store the type of the object that was saved in the string. As such any methods that the stringified object had may not be present when an object is parsed. In order to fix this, `Grapha` has the following functions: "`makeSimpleGraph(toSimple)`" and "`makeComplexGraph(toComplex)`" which take parsed objects as parameters and return the corresponding complex graph and simple graph equivalents.

When Internet Explorer is run from a local computer, it does not allow a user to save an item to the browser's cache. This creates a problem where none of the user's individual graphs can be saved temporarily. Although cache saving is not an option, temporary saving to RAM (as with all programming languages) is still an option. To allow the user to have a similar experience though, a new way of temporarily saving graphs had to be created. The solution was to check if the browser has the `localStorage` object set up already, and if not then one needs to be created. The core of the browser cannot by manipulated though. This means that in order to allow saving in `Grapha` a temporary (volatile) object named "`ieStorage`" was created which supports all of the same functionality as the `localStorage` object. The user can then proceed to use all of the various functions of `Grapha` as he normally would; however, if he closes the browser session he will lose his temporary graphs. `Grapha` also has the ability to save graphs to a user's file system; this means that Internet Explorer users will have to save and load their graph library every time that they begin and finish working.

This brings us to another one of Internet Explorer's shortfalls. When a user wants to save his graph library to his hard drive on any other browser the user only has to click the save button. This uses the download attribute which is a feature of HTML5. Internet Explorer does not yet support this feature so in order for a user to save his library the user must open a GUI and type out a name manually, then click save. To accommodate this, `Grapha`'s JavaScript code checks for when the user is using Internet Explorer and changes the way that the application responds to a request to save.

Another place where saving files is an issue is the area of mobile platforms. Some mobile platforms do not expose the file system to the user. This can make it very difficult for the user to save his graph libraries or load them. Furthermore, it will also be difficult for the user to save his raster-based images to his file system. Many of these downfalls of mobile devices cannot be overcome by `Grapha` and the device users are left with a slightly less functional piece of software.

3.4 Object-Oriented Code

During the development of `Grapha`, the need for having a modular program which could be easily expanded became apparent. There is no way that a programmer could identify all of the basic graphs which the end user would want to generate. There is also no way that a programmer could identify or predict all of the necessary outputs that the user base would want. This led to the realization that `Grapha` would need to use easily created objects which represent graph types and outputs. This allows future programmers to expand the functionality of `Grapha` to meet future needs.

`Grapha` was developed in multiple files which, for production, are combined into one HTML file. One of the files (named `GraphTypes.js`) defines the different basic graphs and graph types which the program generates. Adding an additional graph to `Grapha`'s repertoire is as simple as adding a function which defines the new graph to that file. The function simply initializes the graph and places it into the repertoire array called "`allGraphTypes`". Then, the function called `initializeGraphs` calls the corresponding initialization function. In Figure 3.1 an example of such code is shown. A similar technique is used for creating new output methods in the file named "`outputs.js`".

Further definition of all of the parameters to the `new GraphType(...)` object is outlined in Section 3.5.1.

Creating new output methods is much more complex, however, because it is necessary to define exactly how to render the graph to be output and, in the case of vector-based image formats, how to output the generating code. Each output method corresponds to a method with which the user will obtain his graph from `Grapha`. Outputs could be PNG, JPG, `TikZ` code, SVG code or others. Writing an output module requires some knowledge of the inner workings of `Grapha` but it can be done independently of the rest of the code, without modifying it. The programmer creating a new output type will create an object in JavaScript named "`Output`" which takes a name and a reference to a function which will generate that output. Then the programmer adds that object to the array named "`allOutputs`". This should be done in the function named "`initializeOutputs`". The complicated part of the output function

```
function initializeBipartite(allGraphTypes)
{
  var biparConn = new Array();
  biparConn[0] = new Array();
  biparConn[1] = new Array();
  biparConn[0][0] = "none";
  biparConn[0][1] = "fullCompl";
  biparConn[1][0] = "none";
  biparConn[1][1] = "none";
  var biparNam = new Array();
  biparNam[0] = 1;
  biparNam[1] = 2;
  var biparNum = new Array();
  biparNum[0] = 1;
  biparNum[1] = 2;
  var biparRes = new Array();
  biparRes[0] = 0;
  biparRes[1] = 0;
  var biparRender = new Array();
  biparRender[0] = "horizLine";
  biparRender[1] = "horizLine";
  var biparUser = new Array();
  biparUser[0] = true;
  biparUser[1] = true;
  allGraphTypes[allGraphTypes.length] = new
     GraphType("Bipartite", true, false, 2, biparConn, biparNam,
     biparUser, biparNum, biparRes, biparRender, undefined,
     undefined, undefined, undefined, undefined, undefined,
     undefined, undefined, undefined, undefined, undefined,
     undefined, undefined);
}
```

Figure 3.1: The code for initializing the bipartite graph type

is exactly how the generate function generates output and gives it to the user. These things will take an experienced JavaScript programmer some time to figure out, but by examining the current output functions the programmer will begin to understand how to implement his own.

3.5 Graph-Related Objects

3.5.1 The GraphType Object

A graph type (or a basic graph) is the framework for how to generate a graph. This is represented in `Grapha` as an object named "`GraphType`", which holds all of the rules for generating a graph of that type. Some examples of graph types are bipartite, wheel, and star. A graph type is defined as a type or family of basic graphs for the purpose of this thesis.

During the development of `Grapha` it was decided that the type of a graph would be the basis for which to begin creating the graph. The graph type has information on how to render the graph, how the vertices connect to each other, and how the vertices are divided into partitions.

```
var thisNewGraphType = new GraphType(name, complToggle, isSquare,

    numOfSections, secConnSec, secNamingGroup, userSecPerNam,

    secNumOrder, secNodeReserve, secRenderType, getNameP,

    getComplToggleP, getIsSquareP, getNumOfSectionsP, getSecConnSecP,

    getSecNamingGroupP, getUserSecPerNamP, getUserSecPerNamAtP,

    getSecNumOrderP, getSecNodeReserveP, getSecRenderTypeP,

    getSecConnSecAtP, getSecNamingGroupAtP, getSecNumOrderAtP,

    getSecNodeReserveAtP, getSecRenderTypeAtP);
```

Figure 3.2: The parameters necessary to initialize a graph type object

In order to hold the definition of a graph, a graph object was designed to have a fixed number of variables which cover all of the properties of a graph type. A list of the variables can be found in Figure 3.2. It is valuable to note here that in a graph type object, vertices which are rendered similarly and connect similarly belong to

a section or partition. For example, the wheel graph has two sections. One section corresponds to the vertex at the center of the graph while the other corresponds to the vertices which surround it. These are two separate sections because one set of vertices is rendered as a single center vertex while the other set of vertices are rendered in a circle. In addition, the center vertex connects to each outside one, while the outer vertices connect to each other (in addition to connecting to the center vertex). The fact that these sets of vertices are rendered differently and connect to each other differently means that they belong to different sections within a graph type object. A section in a graph type has a number which uniquely identifies it (0, 1, 2, ...).

The full initialization call for a new graph type object is shown in Figure 3.2. The letter "P" in that figure's variables stands for pointer.

When `Grapha` wishes to know information about a section, (how it should be rendered, how it connects to other sections, etc.) it looks up that section using its number. All information concerning a given section can be found at its identification number in a given array. For a one-dimensional array which refers to sections, the software can look up how a section behaves by looking at the index which corresponds to the section number. For example: if a programmer wants to look up how a section numbered x is to be rendered, he would look up the array named "`secRenderType`" at index x (`secRenderType[x]`). For a two dimensional array which relates sections to each other, lookup happens in a very similar manner. How sections relate to each other is found by looking at the numbered indices which correspond to the arrays that are to be looked up. For example, if you wanted to see how section 0 connected to section 2 you would look up the two dimensional array named "`secConnSec`" at indices 0 and 2 (the code would be `secConnSec[0][2]`).

Following is a list important attributes which the GraphType object has. A description of each attribute is following each attribute name.

`name`: A string which represents the name of the basic graph which is displayed to the user. This is also used as a primary key for graph types, so all graph types must have unique names.

`complToggle`: A Boolean which represents whether the graph can be toggled as complete or not.

`isSquare`: A Boolean which represents whether the graph's rendered dimensions are square or not.

`numOfSections`: An integer which represents the number of sections or partitions of vertices that the graph type contains.

`secConnSec`: A two dimensional array of integers which represents how sections of the graph type connect together. It is this array which specifies which edges are to be generated and which two vertices they connect. Example: `secConnSec[0][0] = "none"; secConnSec[0][1] = "full";`. The two examples just mentioned refer to two facts respectively: first, that the vertices in section 0 do not connect to other vertices in section 0 and second, that every vertex in section 0 connects to every vertex in section 1.

`secNamingGroup`: An array of integers which defines which sections are in the same naming group by assigning each naming group to a number and then giving sections in the same naming group the same number. A naming group is a group of vertices which share the same label (such as v_1, v_2, etc.). Example: if there are four sections, and the first and last sections have the same naming group and middle two have the same naming group the array will be: `secNamingGroup = {1, 2, 2, 1}`.

`userSecPerNam`: An array of Booleans which defines whether the user has the ability to edit the quantity of vertices which are contained within a naming section. If not, then the reserved number of vertices is used. This number is held in the `secNodeReserve` array described below.

`secNumOrder`: An array of integers which defines the order of vertex numbering for items which are in the same naming group. This is specified by the numerical order held in this array. For example, if there are three sections in a graph type and the first two are in the same naming group (specified by the fact that both have the same number contained in the `secNamingGroup` array and the third section having a different number), this array could be: `secNumOrder = {1, 2, 1}`, indicating that the first section obtains lower numbers and the second section obtains higher numbers in the first naming group. The numbers referred to here are the vertex numbers which are part of the vertex label.

`secNodeReserve`: An array of integers which defines how many vertices are automatically reserved from the user's amount in a naming group for a section. This is because the user cannot specify multiple amounts in the same naming group so sections of fixed size are specified here. An example would be a wheel graph: there are two sections of vertices, the first section is the vertex which lies in the middle of the graph and the other section refers to the vertices which surround the middle. In this case, we assign both to the same naming group and allow the user to define the graph with a single number. This means that the first section will have 1 node reserved which is subtracted from the user's entered amount to be used as the middle vertex. In the case of a wheel graph, this array would then look like: `secNodeReserve = {1, 0}`.

`secRenderType`: An array of strings which defines how to render a section. Currently implemented types are: "`horizLine`", "`verticLine`", "`roundOutside`", and "`center`". They render a section's vertices in, respectively, a horizontal line, a vertical line, a circle which is outside of any previous circles, or a center vertex. If a programmer wishes to expand this list, it can be done.

With all of the attributes specified above it should be possible to create most imaginable graph types, so long as that type of graph follows rules and does not have a random or haphazard way of being rendered or generated. In order for a programmer to create a new graph type to add to `Grapha`'s repertoire, the programmer should initialize all of the variables in a `GraphType` object inside of the initialization function for that graph type. These attributes give the programmer the ability to tell `Grapha` exactly how to generate the graph, what it looks like, and what the user's options are when creating the graph.

These variables, however, are limited in the options that they give for making a graph type. The set, static variables constrict the graph, making it rigid and inflexible. For example, if a graph type object should be rendered differently based upon the number of vertices contained within each section, then the programmer will not be able to accomplish it with these variables alone. `Grapha` supports another feature which makes this kind of dynamic graph type available as well. Any programmer

who creates a new graph type object will be able to redefine all of its functions, including the values that those functions return.

The initialization of a graph object takes pointers to functions. For example, a graph type has functions: `getSecConnSec` and `getSecConnSecAt(index)` which return the `secConnSec` array, and an element from the `secConnSec` array respectively. At object initialization, when a programmer passes a function pointer to the graph type initialization function, `Grapha` over-writes that function with the programmer's new function. Then when the proper function is called to access one of the graph type object's variables, the programmer's new code is run instead. This functionality could allow a call to `getSecRenderTypeAt(2)` to (instead of just returning the initialized value of `secRenderType[2]`) check to see how many vertices are in section 2 and return a different string representing how to render the section based upon that number. This function over-write capability allows `Grapha`'s graph type objects to be extremely flexible and customizable. Using the function defined in Figure 3.2, a programmer can easily add to the repertoire of `Grapha` and expand the basic graph line-up.

3.5.2 The SimpleGraph Object

A `SimpleGraph` object refers to a single user-generated instance of a `GraphType` object. It is the basis upon which all graphs are based. The `SimpleGraph` object stores the user's values that he wishes to impose upon his chosen graph type and holds those values for generation and rendering. The initialization function for a `SimpleGraph` object has the following prototype:

```
var thisNewSimpleGraph = new SimpleGraph(name,
    type, vertLabel, complete, sectionAmounts,
    sectionNames, sectionNumbers, scale,
    textSize, xRes, yRes, weights, weightPos,
    rotate);
```

A `SimpleGraph` has a name, which must be unique. If the user attempts to save a `SimpleGraph` whose name is equal to the name of another saved `SimpleGraph` then

he is prompted to specify a different name. The object also has a type, which is a reference to a `GraphType` object. The rest of the variables all correspond to user-entered values. Examples are: `vertLabel`, which is true if the user wants the vertices to be labelled and false if the user did not want them to be labelled; `complete`, which is true if the graph is complete and false if it is not; `sectionAmounts`, which holds an array of the number of vertices in each section naming group; `rotate`, which holds the rotation (in degrees) around its center; and `xRes`, which holds the resolution in the x direction for the graph when rendered as a raster-based image.

Most of the parameters are very straightforward and self explanatory, the scale parameter refers to the node scale size while the `textSize` parameter refers to the text size (in points). The more interesting objects are `weights` and `weightPos`, which each hold an array whose values correspond to each edge. For every weight and weight position there is an edge to which they correspond. This is covered in more detail in Section 4.1.2.

The purpose of the `SimpleGraph` object is to simply hold the values that the user has entered. It allows quick creation of graphs for the user and it supplies `Grapha` with an object template which is used to save, load and delete graphs in the cache and on the user's hard drive. This is covered more in Section 4.1.2.

3.5.3 The ComplexGraph Object

A `ComplexGraph` object refers to a combination of one or more `SimpleGraph` or other `ComplexGraph` objects which have been generated by the user. `ComplexGraphs` are objects which have been generated by the user by combining multiple `SimpleGraph` objects. `SimpleGraph` objects are combined in some way (for example: via their vertices or via an edge) and are layered one next to the other. The initialization function for a `ComplexGraph` has the following format:

```
var thisNewComplexGraph = new ComplexGraph(name, graphs,
    graphFrom, graphTo, graphFromSec, graphToSec,
    graphConnHow, graphFromSecAt, graphToSecAt, graphDominate,
    graphRotate, horizFlip, vertFlip);
```

A `ComplexGraph` object has a name, which must be unique. It also holds all of the graphs which are used in the generation and rendering of the complex graph. The variables `graphFrom`, `graphTo`, `graphFromSec`, `graphToSec`, `graphConnHow`, `graphFromSecAt`, `graphToSecAt`, `graphDominate`, and `graphRotate` are all arrays whose indices correspond to a relationship between any two graphs in the complex graph. For example, all of the arrays except for the `graphs` array describe a connection relationship between two graphs. The first item in the `graphs` array is simply placed on the graph, then all of the other items are connected to it, or to items connected to it. This is done using the items in the above mentioned arrays.

The connection arrays in a `ComplexGraph` object tell of a connection at each of their indices. At array index 0 they will describe a relationship between the graphs in `graphFrom[0]` and in `graphTo[0]`.

The variables `graphFrom` and `graphTo` designate which graphs in the array `graphs` are to be combined, while `graphFromSec`, `graphToSec`, `graphFromSecAt`, and `graphToSecAt` are all used to specify the place at each graph that is to be the combining point (if any). The `graphFromSec` and `graphToSec` arrays both hold the section number of each graph that they are to be identified with, while the arrays `graphFromSecAt` and `graphToSecAt` are used to specify the vertex number in the section at which the graphs are to be combined.

A connection between two graphs in the complex graph also has three more attributes: `graphConnHow`, `graphDominate`, and `graphRotate`. The variable `graphConnHow` contains a string which holds the way in which the graphs are to be combined. This variable was included for future proofing the software and to allow different connection types. It will allow future programmers to enable graphs to be connected at more than just vertices. Right now there are two ways in which graphs can connect with each other. The first is via the string "`none`" which tells `Grapha` that the two graphs are not attached in any way; instead the graphs should simply be rendered side by side. The second way that graphs can be attached is identified by the string "`is`" which tells `Grapha` that the identified vertex on one graph is the identified vertex on the other one.

The `graphDominate` variable tells `Grapha` which graph is to keep its identified vertex's name and size and which graph's vertex's name and size are to be overridden by the other's. Vertices which are not dominant are not rendered. The final attribute of a connection between two graphs is the array named "`graphRotate`", which defines how the new graph should be rotated. This variable specifies how the new graph should be rotated around the combined point, if possible.

There are two special attributes here which are not arrays; they indicate whether or not the compound graph object should be flipped horizontally or vertically; `horizFlip` and `vertFlip`. These attributes act upon the entire compound graph and change the method in which it is rendered.

With all of these variables `Grapha` has the ability to combine large numbers of graphs as per the user's requests. The graphs can be forced together in various ways, then rotated. Graphs can then "pile up" upon each other, which preserves the rotation and the structure of each graph. For more information on the rendering of graphs see Section 4.2 and Section 4.3.

Chapter 4

Implementation and Software Explanation

4.1 Manipulating Graphs

In this chapter, the way in which `Grapha` manipulates graphs will be fully explored. Graph rotation, graph combination, graph outputting, and the ways in which `Grapha` handles graphs will all be discussed within this chapter.

4.1.1 How Objects Are Used, Saved, and Loaded

Within `Grapha` there are a number of different global arrays which hold all of the initialized objects. There is one named "`allGraphTypes`" which holds all of the different `GraphType` objects which have been initialized. One global variable is named "`allOutputs`" and it holds all of the different `Output` objects which have been initialized. Another is named "`allWeightPositions`" and it holds the five different weight positions that are available to the user. Finally, there is a variable named "`allScaleTypes`" which is global and holds all of the different `GraphScaleType` objects.

These global arrays are used during the launch of `Grapha` to dynamically change the user interface. This means that every time a programmer needs to add a new

output type or graph type to `Grapha` he does not need to edit any other code, except to add that initialized object to `Grapha`'s global arrays. In addition, whenever one of these objects is required by `Grapha`, the software can look up the details of it within that array. This allows `Grapha` to be modular on a programming level. A lot of the details of implementation can be difficult to understand or can take a long time to fully learn. The modularity of `Grapha` can ease a programmer's burden by simply providing an object to be created. Then, once created and placed in its proper location, the programmer need not worry about the other details involved in using that object.

Another variable which `Grapha` uses often is the `localStorage` variable. This variable is what gives `Grapha` access to the browser's cache for storing graphs. `Grapha` simply calls `localStorage.setItem(key, value)` or `localStorage.getValue(key)` and the value which is identified by that key is stored or loaded from the browser's cache, respectively. `Grapha` stores one key-value pair, which has the key "`userGraphs`". This key stores a JSON (JavaScript Object Notation) representation of an array of all graphs in the user's cache. When `Grapha` needs to access this array, it simply calls `localStorage.getValue("userGraphs")` and then turns that string back into an array using the JSON JavaScript object, as described in Section 3.3.2.

`Grapha` can then use this array of graphs to populate the load and delete drop-down menus in the user interface, or recall all of the properties of a graph to render or output it. When a graph is to be saved back into this array `Grapha` performs a few tasks. First `Grapha` must check to see if this name is already used. If so, then the software prompts the user to see whether or not he would like to replace the current graph stored under that name. After doing this, `Grapha` either stores the new graph into the browser's cache (depending on the user's response), deletes the old graph and replaces it with the new one, or simply does nothing.

Saving and loading the array of graphs to and from a file is very similar to saving and loading the array to and from cache. In order to save an array of graphs (a library of user-created graphs) to a file, `Grapha` first prepares the array as JSON. Following that, `Grapha` converts that string into a base-64 encoded ASCII string. Then using the new HTML5 download capability, the `download` attribute of the "`Save All Graphs`

`As A File`" button element is set to the user-entered file name and the `href` (or link) attribute is set to the base-64 encoded ASCII string prepended with the text `data:application/octet-stream;charset=utf-8;base64`.

Once these things have been accomplished, the user simply has to click on the "`Save All Graphs As A File`" button and the browser recognizes that the file at the `href` address (the link, or in this case the file encoded directly into the link address) is to be downloaded. The browser can make this connection due to the fact that the download attribute of the clicked element is populated with the name of a file. `Grapha` then sends the file at the `href` specified location to the user's computer under the name which is specified with the `download` attribute.

4.1.2 Creating, Saving, and Editing a Basic Graph

A user's first task when launching `Grapha` for the first time will be to pick a graph type to be the framework for his first graph. This graph type, when selected, is searched for in the `allGraphTypes` array. Once the graph type (whose name matches the drop-down menu element that the user clicked on) is found, `Grapha` accesses the attributes of the graph type and determines which options the user needs. Once `Grapha` has determined this, it then updates (changes the HTML) the user interface to reflect these options.

To perform these user interface updates `Grapha` must check if the graph is square or not, using the `graphType` object's `getIsSquare` function. If it is not square then the user needs to have access to both width and height; if it is, then a single text box for both dimensions will do. `Grapha` must also check if the user can toggle the complete state of the graph, as received from the `getCompToggle` function of the graph type object. This determines whether the user will see the "`Complete?`" check box in the user interface. Recall that a graph being complete refers to the idea that all possible edges are drawn for that graph.

In addition, `Grapha` must also give the user options for the number of vertices in each section and the label and starting number of each section. The latter option is a little bit more difficult to do then the former two options. First of all, two or

more sections can be combined internally. In this case only one is displayed to the user (thus two sections are in the same naming group, like a wheel graph discussed in Section 3.5.1). Second, perhaps a section is unavailable for user editing, such as in the Petersen graph where the user cannot change the number of vertices.

The issue of user input will be further explored in Section 4.2.2. For now, `Grapha` takes each user input number of vertices as well as every label and label number which the user has typed and attributes them each to a naming group. Thus `Grapha` simply stores the user variables into the arrays `sectionAmounts`, `sectionNames`, and `sectionNumbers`.

To set up the user interface properly, `Grapha` must pay close attention to the graph type object for which it is giving the user options. To place the proper number of naming groups on the user interface `Grapha` checks each section of the graph type (recall that a section refers to a group of vertices which share similar properties). When `Grapha` encounters a section which is part of a naming group that it has not encountered yet, `Grapha` adds a new section for vertices and naming. However, there is one catch: the programmer might not have wanted the user to be able to edit the number of vertices in that section. `Grapha` must then use the graph type's function `getUserSecPerNamAt` to obtain information on whether or not the user should receive control over the vertices of that section. After eliminating these sections, each remaining user interface text box corresponds to a unique naming group in the graph type.

When the user submits a request to create a graph, `Grapha` simply initializes the basic graph by placing the values from the user interface into the `SimpleGraph` object's initialization function. The graph is then stored in a global variable which holds the user's current basic graph. This global variable is the variable which is used to populate the HTML (or graphical user interface portion of `Grapha`) with the proper values. It is also the object to which future updates are pushed.

In order to allow the user to update a graph, `Grapha` has event listeners which listen for clicks (both mouse up and mouse down) and key presses. Once an event has occurred, `Grapha`'s JavaScript then executes the function which corresponds to that given request. For example, when a user wishes to change the node size of a

graph, the user will click on the corresponding node size that he wants in the drop down menu. `Grapha` (which has been waiting patiently for this event to happen) then reads in the name of the clicked node size. The software then sorts through the array named "`allScaleTypes`" (which holds all of the different scale types) and finds the one which corresponds to exactly what the user clicked. This value is then written to the basic graph object.

Edge weights and their positions are another user-editable part of the basic graph. When `Grapha` generates a list of edges from a graph object, it then pairs each edge with corresponding entries in two arrays which are part of the `SimpleGraph` object (described in Section 3.5.2). The first array, `weights`, holds all of the weights that a basic graph's edges has. The second array, `weightPos`, holds all of the positions of each weight for each edge. When the edges are generated, each weight and weight position is simply placed onto the edge object. How edge objects are generated is explored in Section 4.2.

When a user edits a weight, the array which corresponds to the edit (or both of the arrays) is edited. This is done in the same manner as any other attribute edit. The global object is then updated and the graph is re-rendered.

Once `Grapha` has identified the new values to be used in the graph, the global graph object (which represents the user's current graph that he is editing) is then changed. After this change has occurred, `Grapha` renders the graph and displays it to the user (further information on rendering graphs is located in Section 4.3). When the user requests to save the graph, this global graph object is then pushed into the `userGraphs` array located in cache.

When a user requests to load a graph, via the tab named "`Edit Basic Graphs`", `Grapha` pulls all of the user-created graphs from cache. Upon doing this `Grapha` then fills out the appropriate drop down menu and adds event listeners to each item in the drop down menu. Once these listeners are in place the software will respond to a click on one of the graph names in the menu. `Grapha` then takes the name of the element which the user clicked on and searches through the cached graphs for one with the same name. Once this has been found the user interface is updated to reflect that graph and its options. In addition the graph is rendered and shown to the user.

In this manner `Grapha` listens and administers user requests to load, save, and change basic graphs.

4.1.3 Combining and Editing Compound Graphs

Within `Grapha` there is the ability to combine basic graphs into compound graphs. This utilizes the `ComplexGraph` object. The `ComplexGraph` object (as described in Section 3.5.3) contains all of the necessary attributes for defining multiple graphs which have been combined into one. A compound graph is composed only of basic graphs; the `ComplexGraph` object has very few attributes which change the entire compound graph. Thus compound graphs in `Grapha` do not store information about the graph as a whole, instead the complex graph simply stores basic graphs as well as the ways in which those graphs connect to each other.

The basic graphs which are stored in the compound graph are copies of their originals. This means that if the user decides to edit a basic graph while it is within a compound graph, then the saved basic graph will not be changed. In addition, when the user edits a saved basic graph which is a part of a compound graph, the compound graph will not change. `Grapha` stores no link that would connect saved basic graphs to saved compound graphs.

When the user decides to combine two different graphs, `Grapha` must decide exactly what the user meant by requesting that combination. First, if the user has clicked upon a vertex from each graph then the software knows to combine the graph by identifying those two points. Second, if the user has omitted clicking on a vertex from one (or both) of his graphs that he wishes to combine, then `Grapha` assumes that the graphs are to be placed next to each other and no vertices are to be combined or identified with each other.

If the user requested that the graphs are to be connected to each other `Grapha` needs to combine them. If the graphs are both basic graphs then this is a relatively easy task. `Grapha` must initialize a new `ComplexGraph` object which has two graphs in its `graphs` array. Then `Grapha` fills in the other arrays of the `ComplexGraph` object in line with the connection point between the two graphs. Once the arrays have been

filled in (as described in Section 3.5.3), the complex graph is stored in a global object (just as simple graphs are) for editing or saving by the user.

When the user requests that a compound graph be combined with a basic graph `Grapha` deals with the request in a similar manner. In this case `Grapha` makes the compound graph the base for the rest of the graph. Then `Grapha` simply adds to each array another connection between two graphs. More specifically, another graph is added to the graphs array, one more section and vertex number is added to each of the identifying arrays. Then the `graphFrom` and `graphTo` arrays each have a number added to them to specify the graphs to be connected.

As can be seen so far, when a user creates a graph `Grapha` does a minimal amount of work to save all of the rules and attributes of that graph. The details of how the graph is rendered or generated is hidden at this point. `Grapha` has other functions which take a graph and create a list of nodes or edges. Following that, `Grapha` then uses functions which take a list of nodes or edges along with the graph as parameters. These functions then render the graph or create output for it.

The situation becomes a little more difficult when a user requests that two compound graphs be combined into one graph. The problem with this is that before now, we were simply inserting a new connection into the arrays. It is not as simple to do this time. Now it is necessary to determine the primary graph and secondary compound graphs.

When `Grapha` receives the request to combine two `ComplexGraph` objects it picks one of the `ComplexGraph` objects and considers this the base (primary) `ComplexGraph`. Once a primary graph has been arbitrarily chosen it becomes the `ComplexGraph` which is added upon to create the new compound graph. After setting a primary compound graph, `Grapha` then begins adding basic graphs and connections from the other (secondary) `ComplexGraph` object to the primary graph one by one. This occurs until they have all been added and all the basic graphs reside within one `ComplexGraph` object. This has to be done in a particular order due to the fact that basic graphs must be in the `graphs` array in the order of their connection for the new graph to be able to be rendered. For example, if graph four attaches to graph two, we cannot add graph four to the new compound graph before graph two or else graph

four will not have any way to attach itself. Therefore only graphs which connect to a previously added graph may be added at any point.

In any case, once all of the basic graphs have been combined into one `ComplexGraph`, `Grapha` then stores the graph into a global variable for manipulation or saving by the user. This is done similarly to how it was done for basic graphs.

Manipulating a compound graph is also very similar to that of basic graphs. When the user clicks on a basic graph (which is a part of a compound graph) in the "`Edit Combined Graphs`" panel the user can then edit that basic graph. When this occurs `Grapha` simply updates that graph in the `graphs` array of the `ComplexGraph` object.

The other edits that a user can perform when he is editing a compound graph are: toggle vertex labelling for all basic graphs contained within, vertically flip the entire graph, horizontally flip the entire graph, auto number the entire graph, or change all of the text and node sizes of all the basic graphs at once. Toggling the labelling of the basic graphs causes `Grapha` to run through all of the graphs in the `ComplexObject`'s `graphs` array and make the `vertLabel` variable of each one true or false. A similar solution is performed by the software when the user requests that all node sizes be changed at once, except this time the variable of each basic graph called "`scale`" is changed to the desired value. When a user wants the compound graph to be vertically flipped or horizontally flipped, the flag contained within the `ComplexGraph` object is simply set to true or false. When the user requests the graph to be auto numbered, the basic graphs which have been stored into the compound graph's `graphs` array are changed so that no numbers overlap and no numbers are skipped. All of these user requests come from the user interface. After the user interface element has been interacted with `Grapha` sets the flags appropriately.

All of these edits to any compound graph are saved in that compound graph's properties. They travel with the graph, even if it is saved into cache or saved to the hard drive (or another user storage device) as a library. These properties are then transferred when two graphs are combined, allowing a user to customize his compound graph and then combine it with another different compound graph. Users can chose to save or output their compound graph at any time that they deem appropriate.

4.2 Generating Lists of Nodes and Edges

Up to this point this thesis has covered some very important pieces of the `Grapha` graph-generating software. The muscle of `Grapha`, however, is the way in which it generates lists of nodes and edges. None of `Grapha's` useful functions can be accomplished without first transforming each graph that the software knows about into a list of nodes and a list of edges. The software's render, preview, and output methods all require lists of nodes and edges to work. This section will focus on the details of generating such lists.

4.2.1 The Node and Path objects

In order to produce any useful output or feedback on the graphs that the user has created, the ability to generate lists of nodes and edges must be available. To begin discussing this, the idea of a node and an edge will be explored.

A `Node` object contains ten attributes: `fromGraph`, `sectionNum`, `vertInSecNum`, `vertNum`, `name`, `number`, `placement`, `scale`, `textSize`, and `visible`. The purpose of each attribute is described below.

`fromGraph`: A reference to the basic graph (a `SimpleGraph` object) which this node is a part of.

`sectionNum`: An integer which represents the section number that this node is a part of. It specifies which section of vertices this vertex is from on the graph that it is a part of.

`vertInSecNum`: An integer which represents this vertex's number in its section. This variable also corresponds to the order (within a section) in which this vertex was generated. For example the first vertex in a section is 0, the second is 1, etc.

`vertNum`: An integer which is the number of this vertex with respect to all other vertices within the same graph. For example the first vertex generated for a graph is number 0, the second is number 1, etc. This number also corresponds to the order in which vertices are generated (i.e. created from the graph object in the software).

`name`: A string which is the label name of this vertex.

`number`: An integer which is the displayed number of this vertex.

`placement`: A reference to either a `Coordinate` or a `CoordinateAngle` object which details the place to put the node in the graph. These two objects are described below.

`scale`: A number which is the scale that the node is to be drawn at.

`textSize`: An integer which specifies the size of the text (in points) for this node.

`visible`: A Boolean which, if set to false, specifies that the node will not be rendered, however any edges attached to it will still be rendered. This is used for identifying vertices, to keep both of them and their labels from appearing.

A sample initialization of a `Node` object follows:

```
var aNode = new Node(fromGraph, sectionNum, vertInSecNum, vertNum,
name, number, placement, scale, textSize, visible)
```

The `CoordinateAngle` and `Coordinate` objects are detailed below. `CoordinateAngle` has five attributes: `x`, `y`, `angle`, `rad`, `relTo`. The `Coordinate` object also has five attributes: `x`, `y`, `xRel`, `yRel`, `relTo`.

In both cases:

`x`: A number which represents the x coordinate at which the center node should be drawn.

`y`: A number which represents the y coordinate at which the center node should be drawn.

`relTo`: A reference to the node to which this node is to be positioned relative to (this applies to the `relX` and `relY` or `angle` and `rad` variables and is currently only used for `TikZ` code). If this is undefined then the node is to be rendered relative to (0,0).

For a `CoordinateAngle`:

`angle`: A number which is the angle at which the node is to be rendered.

`rad`: A number which is the radius at which the node will be rendered from the center of rotation.

For a `Coordinate`:

`xRel`: A number which is the relative x position of this node compared to `relTo` node.

`yRel`: A number which is the relative y position of this node compared to the `relTo` node.

The `Path` object (which represents an edge) holds four attributes. These are defined below.

`node1`: A reference to one of the nodes that this edge connects.

`node2`: A reference to the second of the nodes that this edge connects.

`weight`: A string, which is the weight to be rendered with this edge.

`weightPos`: A string which details where the weight is to be drawn relative to the center of the edge.

These objects, once all generated, can be used to output graphs to any format as seen in Section 4.3.

4.2.2 Lists of Nodes from Basic Graphs

When a request is made for `Grapha` to generate an output or render the graph to a canvas in the user interface, `Grapha` must first generate a list of nodes and a list of edges. In order to do this `Grapha` must first parse the basic graph object that it has into arrays which are more directly usable.

One slight problem here arises from the user interaction. In the case of a wheel graph the user only inputs the number of vertices for one section. `Grapha` simply stored this value into the `SimpleGraph` object's `sectionAmounts` array as it was. When `Grapha` needs to generate lists from a graph it must then process this array to fill out the number of vertices in both of the graph's sections. This is done by iterating through all of the sections contained within that naming group and subtracting the number of vertices reserved for that section from the user-entered amount. Any left over vertices are allotted to the section in the naming group which can take any number of vertices.

`Grapha` accomplishes this via the graph type's `getNodeReserveAt` function which returns how many nodes that a section needs to have reserved. For example, in a wheel graph the center vertex is one section (since it is rendered differently than the other vertices) and the circle of surrounding vertices makes up another section. The

section containing the middle vertex needs one reserved node out of the user-allotted amount. This means that, out of the number of vertices entered by the user, one will be used for the center vertex, and the remaining will be used on the outside ring of vertices. `Grapha` then assigns these vertex totals to each section in the graph. These values are then used during the rest of the node-generating process.

Once this has been determined, the placement values (coordinates) for all of the nodes are then generated. They are generated differently based upon the way in which the particular section is to be rendered. To determine how a section should be rendered `Grapha` retrieves the graph's `type` attribute which contains a `GraphType` object. This object's `secRenderTypeAt` array is then retrieved and, depending upon the string contained within, `Grapha` performs a different style of set up for the coordinates.

For illustration here I will go over the details of two different render methods: horizontal lines, and circles. If a particular section is to be generated as a horizontal line, as in the bipartite graph or the path graph, the string retrieved will be "`horizLine`", if it is to be generated as a circle the string retrieved will be "`roundOutside`". The coordinates generated from these render types are then placed into a node object along with the node's other properties (which are trivial to generate). Most of them are simple lookups into the arrays of the `SimpleGraph` object or are clear insertions based on which number the vertex is in order of generation.

When `Grapha` retrieves the string "`horizLine`" `Grapha` then determines two different things. The first is exactly how much space it has to work with. This can be accessed from the graph's `xRes` and `yRes` attributes. The second is: how many other "`horizLine`" sections does this graph have? For each "`horizLine`" section the graph should be divided vertically into one additional part. Thus for one "`horizLine`" section the graph will consist of two parts divided by that one horizontal line. If there are two "`horizLine`" sections then the graph will consist of three parts divided by the two horizontal lines.

`Grapha` uses the total space to find the y coordinates of these lines. `Grapha` then splits the total x width of the graph by the number of nodes which are to be generated in that line. This is done in a similar method to that of determining vertical space.

One difference is that each side of a horizontal line starts near the edge of the graph, while the horizontal lines themselves sit vertically at evenly spaced points in the final graph. One exception to this is that if the horizontal line only contains one or two nodes the nodes are placed with even spacing on all sides of them instead of being pushed to the edge. See Figure 4.1 for an illustration of this. This rendering method, combined with the exceptions for lines with few vertices, cause the rendered nodes to have a pleasing look. The placement algorithms keep the nodes looking neat, professional, and symmetrical.

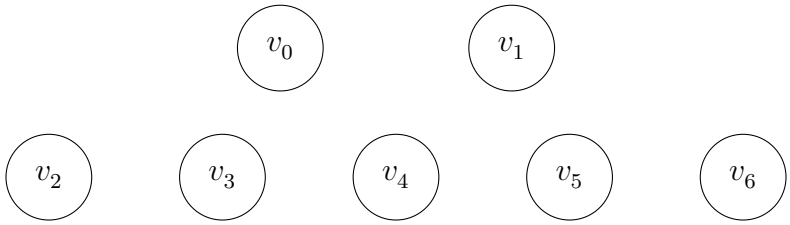

Figure 4.1: Two node and five node horizontal lines

As a second example, when `Grapha` reads the string "`roundOutside`" `Grapha` then determines two different things. The first is exactly how much space it has to work with. This can be accessed from the graph's `xRes` and `yRes` attributes. The second is: how many other "`roundOutside`" sections does this graph have? For each "`roundOutside`" section the graph should be divided into evenly spaced parts by radius. Thus if a graph only has one "`roundOutside`" section, the graph will consist of two empty spaces (in terms of radius from the middle) split by that one circle of nodes. If there are two "`roundOutside`" sections in a graph then the graph will consist of three empty spaces (by radius) which are divided by two circles of nodes. In this manner there are three sections of white space divided by two circles of nodes.

`Grapha` uses the total space to find the radius of these circles of nodes. `Grapha` then splits the total circumference of the circle by the number of nodes which are to be generated. This will make the space between each node on the circle equal. See Figure 4.2 for an illustration of this. This rendering method gives a pleasing look to the final layout of the nodes. It keeps them looking neat, professional, and symmetrical.

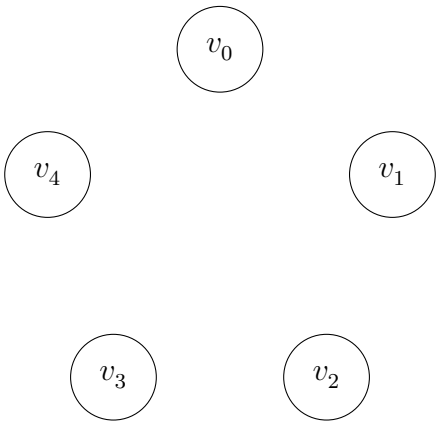

Figure 4.2: Five nodes generated with the string "roundOutside"

After any nodes are generated using this method, the nodes are then rotated around their center by the rotation amount stored in the `SimpleGraph` object. This places all of the nodes in their final position which is where they will be rendered. For the sake of making pictures smaller, any excess white space surrounding the graph is also trimmed off before the user sees the nodes and before the user outputs the graph. This could translate all of the nodes to the left, or upward but the nodes will end up in the same space on the graph relative to all other nodes. Sometimes this can make the graph a bit smaller than the user's desired dimensions, but the difference is negligible (unless only a few nodes are used) and does not affect the look of the graph.

4.2.3 Lists of Nodes from Compound Graphs

For compound graphs, nodes are generated using a very similar method as that used for basic graphs. In fact, the compound graph node generator uses the basic graph node generator once for every basic graph contained within the compound graph. This means that for each basic graph, there is a list of nodes generated. Then after these lists of nodes have been generated, the nodes are changed slightly to better suit their use in a compound graph.

The first change is that some nodes should now be invisible. This is true of some nodes which are identified with nodes from other graphs. If the node is not on the

dominant side then it should be invisible. Which graph is dominant in a connection is indicated by the array `graphDominate` as described in Section 3.5.3. The nodes which are not dominant have their `visible` attribute set to false.

The other change to be made is that now every node which is not a part of the first graph in the list needs to be relative to another node. This means that the `relTo` attribute of each node's placement attribute should be changed to the node which the two graphs are connected by.

After these changes have been made to the nodes, they are then translated so that any identified nodes line up perfectly. They are also then translated again in order to account for removing excess white space. Once that has been accomplished the nodes are then flipped horizontally or vertically if the flags on the `ComplexGraph` object are set to do so. This involves changing all of their y values and x values so that the nodes are mirrored across the center lines, which lie on the x and y axes.

4.2.4 Generating Lists of Edges

Generating edges is another key part of setting up a graph for rendering or output. Generating edges is a simple matter which involves taking the list of generated nodes and the values from a `SimpleGraph` object's `secConnSec` array and creating `Path` objects which hold the proper nodes. To do this `Grapha` simply takes each entry in the `secConnSec` array and, depending upon the string describing how to connect the sections, creates `Path` objects.

For illustration I will go over a path-generating method. If the string retrieved is "`full`" then `Grapha` knows that every node in one section (the sections to be connected are specified by the indices on the `secConnSec` array as discussed in Section 3.5.3) is connected to every node in the other section. For every node in both arrays, `Grapha` creates a `Path` object which holds a reference to each node.

Once `Grapha` has found all of the edges needed for the graph (or in the case of a compound graph, graphs) `Grapha` looks through the entire list of edges and deletes duplicates. This is to keep `Grapha` from having dozens of hidden edges which were generated twice. Following this, `Grapha` then passes through the array of edges and

the arrays `weights` and `weightPos` and pairs each `Path` object with a weight and a weight position.

Finally `Grapha` needs to change some of the edge's nodes due to the fact that some nodes are invisible. The invisible nodes are known to all be identified with another graph, this means that the edge should no longer travel to the node which is hidden but rather to the dominant node from the other graph. These edges are searched for and their nodes are changed to the proper new nodes.

Now that the lists of nodes and edges have both been generated, `Grapha` is ready to use them to output the graph into various formats or render it to the screen. This step is described in Section 4.3.

4.3 Rendering and Outputs

In this section, outputting graphs and rendering graphs to the screen will be examined. Although both of these processes are quite simple, `Grapha` would not be a useful program without them.

When a graph is going to be output for rendering `Grapha`'s task is simple. Most of the difficult work was accomplished when the nodes and edges were generated. Now `Grapha` simply has to traverse the lists of nodes and edges and convert them into the proper format.

4.3.1 Rendering the Graph and Obtaining Raster-Based Images

In order for a graph to be rendered to the screen `Grapha` uses the HTML5 `canvas` tag. This tag allows raster-based images to be drawn into the browser window. It does this by providing `JavaScript` functions which edit pixels. These functions allow a programmer to specify which pixels are to be coloured, where they should be coloured, and how they should be coloured.

By using all of the attributes of a `Node` object `Grapha` has the ability to place a node, fill it with a label and number, scale it to the proper size and determine whether

or not it should be rendered. These things are all easily accomplished in `JavaScript` by calling functions which act upon a context which is obtained from the `canvas` object.

The functions named `stroke`, `fillText`, `arc`, `lineTo`, `moveTo`, `fillStyle`, and `strokeStyle` are all important to using a canvas to draw images. A tutorial of how to use these functions to draw graphics can be found at the `w3schools` website [24]. Due to the nature of this thesis it is not important to go into the details of how to use each function. They manipulate the pixels in a `canvas` tag in a raster-based way which is specified by `Grapha` using the attributes of a node or an edge.

Following all of the node rendering, edges are then drawn in a similar manner. The lists of nodes and the lists of edges are traversed and each object found within either list is drawn to the screen. This occurs until the entire graph has been drawn.

Once all of the nodes and edges have been rendered to the screen, the user may wish to output the graph. For raster-based images `JavaScript` has a built in function which makes this easy. The "`toDataURL`" function acts on any `canvas` element and allows what has been drawn on that canvas to be output as a picture in another format. The function is used as follows: `canvas.toDataURL("image/" + dataType)`. The word "`dataType`" is a variable which holds the name of the format into which the graph will be output. Formats which are supported by the `toDataURL` function include PNG, WebP, JPEG, and BMP; however, this support is not universal and in some cases a browser might support some formats and not others. If a format is not supported it will default to one of the supported formats and output that format instead.

4.3.2 Obtaining Vector-Based Images

In order to output vector-based images `Grapha` has to work much harder. Vector-based images usually are written using code and each type of vector-based image usually has its own code which is used for rendering the image. How to generate `TikZ` code and how to generate `SVG` code will be covered below.

TikZ [11] is a language for specifying different kinds of vector graphics for TeX [12], LaTeX [6], or ConTeXt [20]. TikZ is a high level language which is used for drawing many different types of diagrams.

When generating TikZ code there first needs to be some set up. The various attributes of the tikzpicture will need to be set before we can tell TikZ where to draw the nodes and edges. For a simple graph this set up only happens once, at the beginning of the tikzpicture. This set up includes things such as the tikzpicture's xscale, yscale, and rotate attributes as well as the style and size of nodes.

Following is an example output for a basic bipartite graph. The attributes xscale, yscale, rotate, and the style and size of nodes can all be inferred from the attributes of a SimpleGraph object.

```
\begin{tikzpicture}[rotate=0,xscale=1,yscale=1, thin,
every node/.style={scale=1.00,minimum size=0cm,inner sep=0pt},
nodeStyle/.style={scale=1.00,shape=circle,minimum
size=1.89cm,inner sep=0.7pt,draw}]
  \node (v0) at (0.0000,0.0000) [nodeStyle] {$v^{}_{0}$};
  \node (v1) at (3.1341,0.0000) [nodeStyle] {$v^{}_{1}$};
  \node (v2) at (0.0000,-4.5290) [nodeStyle] {$v^{}_{2}$};
  \node (v3) at (3.1341,-4.5290) [nodeStyle] {$v^{}_{3}$};
  \path (v0) edge[] node {$$} (v2);
  \path (v0) edge[] node {$$} (v3);
  \path (v1) edge[] node {$$} (v2);
  \path (v1) edge[] node {$$} (v3);
\end{tikzpicture}
```

After the initial set up has been achieved, Grapha must then output code for each of the nodes and each of the edges. This requires looping through the list of nodes and the list of edges (generated from a graph) and pasting all of the code necessary to draw those objects within the TikZ output. As seen above, the code for a node is quite simple, and can all be obtained from the Node object. The x and y coordinates use the xRel and yRel attributes of the node's placement attribute and the name is

a combination of the various other attributes. The labels for each node are derived from the `Node`'s `name` and `number` attributes.

For compound graphs, generating `TikZ` code becomes slightly more complex. Below is the `TikZ` output of a bipartite graph combined with a path graph. The path graph is also rotated 30 degrees. The coinciding picture is Figure 4.3.

```
%This graph was designed for a particular text size.
%However, the graph
%borrows the text size from the document,
%any differences in size are due to this.
\begin{tikzpicture}[rotate=0, xscale=1,yscale=1, thin,
every node/.style={scale=1.00,minimum size=0cm,inner sep=0pt},
nodeStyle/.style={scale=1.000,shape=circle,minimum
size=1.134cm,inner sep=0.7pt,draw}]
  \node (v0;0) at (0.0000,0.0000) [nodeStyle] {$v^{}_{0}$};
  \node (v0;1) at (3.0507,0.0000) [nodeStyle] {$v^{}_{1}$};
  \node (v0;2) at (0.0000,-3.1703) [nodeStyle] {$v^{}_{2}$};
  \node (v0;3) at (3.0507,-3.1703) [nodeStyle] {$v^{}_{3}$};
  \begin{scope}[rotate=-30, xscale=1, yscale=1,
every node/.style={scale=1.00,minimum size=0cm,inner sep=0pt},
nodeStyle/.style={scale=1.00,shape=circle,minimum size=1.1340cm,
inner sep=0.7pt,draw}]];
  \node (Graph1Position) at ($(v0;1)+(0,0)$)  [nodeStyle,
  draw=none] {$$};
  \node (v1;0) at ($(Graph1Position)+(0.0000,0.0000)$)
  [nodeStyle,draw=none] {};
  \node (v1;1) at ($(Graph1Position)+(2.7790,0.0000)$)
  [nodeStyle] {$v^{}_{4}$};
  \end{scope};
  \path (v0;0) edge[] node {$$} (v0;2);
  \path (v0;0) edge[] node {$$} (v0;3);
  \path (v0;1) edge[] node {$$} (v0;2);
```

```
  \path (v0;1) edge[] node {$$} (v0;3);
  \path (v0;1) edge[] node {$$} (v1;1);
\end{tikzpicture}
```

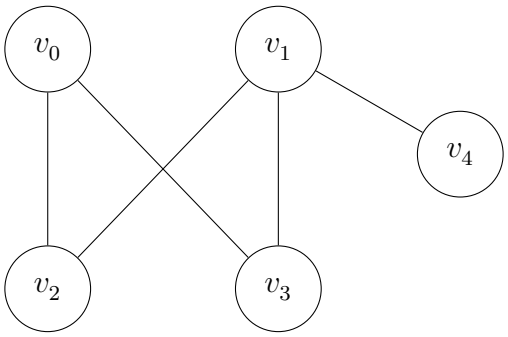

Figure 4.3: A bipartite graph with a path graph attached

As can be seen, every different basic graph requires the creation of a new `TikZ` scope. In this manner, each basic graph can retain its unique attributes and rotation. Following that, in order to have the basic graphs line up properly, every node in a combined graph which is not part of the first basic graph has a calculated position. This calculated position uses the relative coordinates of the node's `placement` attribute. It uses the `relTo` attribute as well as both the `xRel` and `yRel` attributes. When combined these create a new node (the `relTo` node) in `TikZ` and then `Grapha` proceeds to make all the rest of the nodes in that basic graph relative to that new node.

In this way, every basic graph is translated into another scope in `TikZ` and then its nodes and edges are written out in `TikZ`. These written nodes and edges are contained within the basic graph's scope in order to give the nodes and edges all of the appropriate properties. This allows `TikZ` code to be generated gracefully and readably. The generated code is easy to edit as well, so if a user wishes to make any custom changes to his graph he could easily make them to the generated `TikZ` code.

In addition, if a complex graph is to be flipped horizontally or vertically, the corresponding scale attribute (`xScale` or `yScale`) must be made negative when the code is output for `TikZ`.

`SVG` code is not as complicated to generate as `TikZ` code. This is due to the fact that the `SVG` code utilizes the `xRes` and `yRes` portions of a `Node` object's `placement` attribute. This means that there is no scope or rotation attributes to be edited. Instead, each node and edge is drawn as a separate entity and is unaffected by the other code generated. A separate styling element is also generated for every node in order to keep the resulting code easy to edit and quick to read.

To generate `SVG` code `Grapha` simply has to loop through every node and edge and generate a bit of code for that item to be drawn. Below is a sample of `SVG` code to generate a bipartite graph.

```
<svg version="1.1" height="244" width="193">
  <circle cx="39.5" cy="39.50000000000003" r="35"
  stroke="black" stroke-width="1" fill="none"/>
  <g font-size="26" font="sans-serif" fill="black" stroke="none" >
    <text x="27.5" y="46.00000000000003">v</text>
  </g>
  <g font-size="20" font="sans-serif" fill="black" stroke="none">
    <text x="40.5" y="49.50000000000003" >0</text>
  </g>
  <circle cx="154.83333333333337" cy="39.5" r="35"
  stroke="black" stroke-width="1" fill="none"/>
  <g font-size="26" font="sans-serif" fill="black" stroke="none" >
    <text x="142.83333333333337" y="46">v</text>
  </g>
  <g font-size="20" font="sans-serif" fill="black" stroke="none">
    <text x="155.83333333333337" y="49.5" >1</text>
  </g>
  <circle cx="39.50000000000006" cy="206.16666666666666"
  r="35" stroke="black" stroke-width="1" fill="none"/>
  <g font-size="26" font="sans-serif" fill="black" stroke="none" >
    <text x="27.500000000000057" y="212.66666666666666">v</text>
  </g>
```

```
<g font-size="20" font="sans-serif" fill="black" stroke="none">
  <text x="40.50000000000006" y="216.16666666666666" >2</text>
</g>
<circle cx="154.83333333333337" cy="206.16666666666666"
r="35" stroke="black" stroke-width="1" fill="none"/>
<g font-size="26" font="sans-serif" fill="black" stroke="none" >
  <text x="142.83333333333337" y="212.66666666666666">v</text>
</g>
<g font-size="20" font="sans-serif" fill="black" stroke="none">
  <text x="155.83333333333337" y="216.16666666666666" >3</text>
</g>
<line x1=39.500000000000014 y1=74.50000000000003
x2=39.50000000000004 y2=171.16666666666666 stroke="black"
stroke-width="1"/>
<rect x="39.50000000000003" y="111.83333333333334"
width="0" height="22"style="fill:rgb(255,255,255);"/>
<g font-size="20" font="sans-serif" fill="black"
text-anchor="middle">
<text x=39.50000000000003 y=127.83333333333334
fill="black"></text>
<line x1=59.41636240690613 y1=68.28087053021117
x2=134.91697092642724 y2=177.38579613645553
stroke="black" stroke-width="1"/>
<rect x="97.16666666666669" y="111.83333333333334"
width="0" height="22"style="fill:rgb(255,255,255);"/>
<g font-size="20" font="sans-serif" fill="black"
text-anchor="middle">
<text x=97.16666666666669 y=127.83333333333334
fill="black"></text>
<line x1=134.91697092642727 y1=68.28087053021116
x2=59.416362406906174 y2=177.3857961364555 stroke="black"
```

```
  stroke-width="1"/>
  <rect x="97.16666666666671" y="111.83333333333333"
  width="0" height="22"style="fill:rgb(255,255,255);"/>
  <g font-size="20" font="sans-serif" fill="black"
  text-anchor="middle">
  <text x=97.16666666666671 y=127.83333333333333
  fill="black"></text>
  <line x1=154.83333333333337 y1=74.5 x2=154.83333333333337
  y2=171.16666666666666 stroke="black" stroke-width="1"/>
  <rect x="154.83333333333337" y="111.83333333333333"
  width="0" height="22"style="fill:rgb(255,255,255);"/>
  <g font-size="20" font="sans-serif" fill="black"
  text-anchor="middle">
  <text x=154.83333333333337 y=127.83333333333333
  fill="black"></text>
</svg>
```

The resulting graph which is generated from the above code is pictured in Figure 4.4.

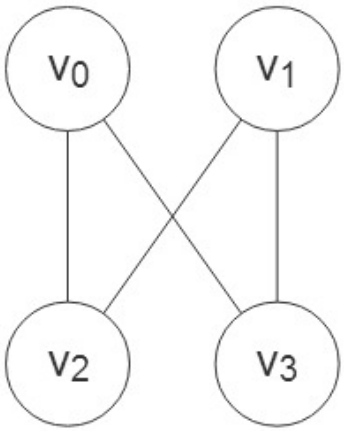

Figure 4.4: A picture of a bipartite graph in SVG

As can be seen, this way of generating `SVG` is not the most space efficient or the easiest to read, but it is easy to edit. Due to the fact that every node and edge has its own section of code and styling, a user can easily change them. Since all of the code is generated by software very rapidly it seems appropriate to have the software do the work and make it long yet easy to edit.

This `SVG` code can be used anywhere that `SVG` images are able to be interpreted and rendered. The output of `Grapha`'s `SVG` is tuned for use in HTML but it should be noted that with a few quick edits the code should be usable anywhere.

Chapter 5

Software Use

This chapter will provide a run through of the general usage scenario for `Grapha` on a desktop computer. It will also provide examples of the various features which `Grapha` has, and show examples of how to use them.

5.1 Overview Of The User Interface

There are two ways in which to run `Grapha`. The first way is to run `Grapha` from its public site. This is currently hosted at: `http://graph-drawing.acadiau.ca/`. The user can open `Grapha` via that link and run it with little difficulty. The other way that `Grapha` can be run is via the user saving it to his computer. By right clicking on the `Grapha` web page at any blank spot and clicking `Save as...` or `Save Page as...` as seen in Figure 5.1, a user can save `Grapha` to his computer. A user will then be prompted to give the program a name and a way to save it. The user may select any name and the selection box should be set to show the words: "`Webpage, Complete`". After this the user can decide upon a location within his computer to save the program to. Then in the future when that user wishes to run `Grapha` he may simply open (for example by double-clicking) that saved icon. After this has been performed, `Grapha` can be run off-line. Note that graphs saved to the browser's cache will not transfer when you make this save.

`Grapha` also will run on any device which has a `JavaScript` engine; this means that the software runs on Android and iOS (mobile operating systems) in addition to desktop operating systems. The framework used for the design of `Grapha` provides it with the ability to adapt the user interface to suit smaller devices better. Furthermore, `Grapha` is still fully functional on a mobile device and is not overly difficult to use.

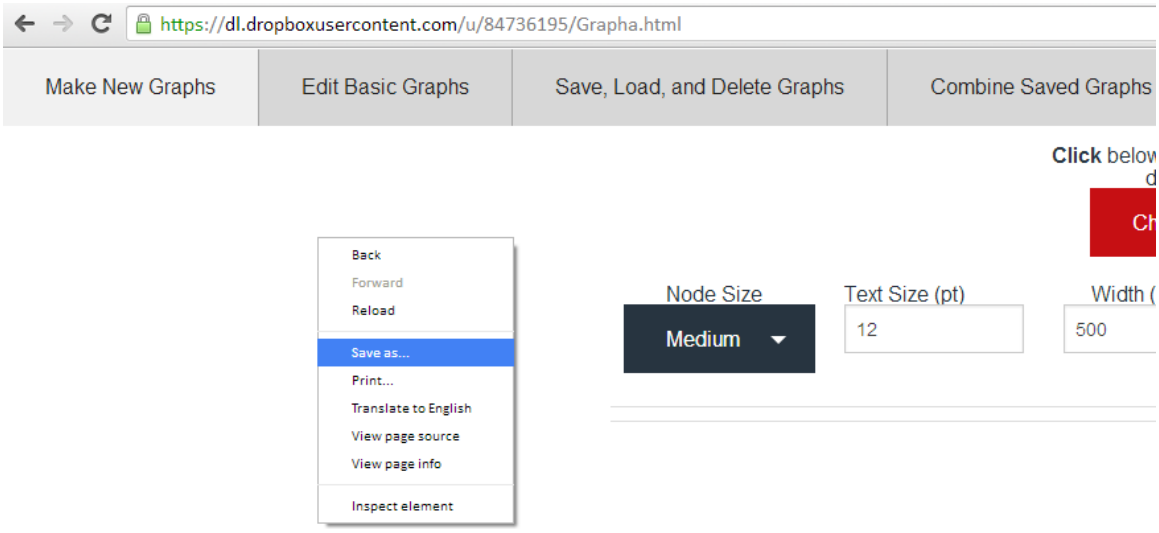

Figure 5.1: Downloading `Grapha`

Regardless of the method of running `Grapha`, the user interface is the same. The user will be greeted by a five-tabbed single-page user interface. On every launch of `Grapha` the user will be placed at the `Make New Graphs` tab. Every tab contains a different set of functions which can be performed. The flow of the user interface is left to right. The user does the most basic and simplest tasks in the leftmost tabs, and as he works he will slowly move right until he is performing very complex tasks (such as combining and rotating graphs).

The tabs available to the user are: `Make New Graphs`, `Edit Basic Graphs`, `Save, Load, and Delete Graphs`, `Combine Saved Graphs`, and `Edit Combined Graphs`. The `Make New Graphs` tab allows the user to begin generating basic graphs with a few commands. The `Edit Basic Graphs` allows a user to perform many operations (edit, save, load, output, change size, etc.) on basic graphs. The `Save, Load, and`

`Delete Graphs` tab allows a user to save his graph library, delete graphs, or load libraries of graphs. The `Combine Saved Graphs` tab allows a user to combine two basic graphs into a combined graph and the `Edit Combined Graphs` tab allows a user to perform operations on combined graphs.

The user interface utilizes the current trend of flat user interfaces. This trend specifies that 3D buttons, pop-out elements, or flashy graphics are not used in the design of the user interface. Instead, colours, simple shapes, and a grid are used to tell the user what to click on, and how to traverse the software. This design style has seen most of its success on mobile platforms due to the fact that it lends itself well to them and "smart" mobile platforms are currently the fastest moving and the most frequently adopted platforms [14], [2]. This design was chosen due to the fact that it promotes a simpler looking interface. Due to the fact that `Grapha` was designed to be simple and quick the design paradigm pairs well with it. This design paradigm can be seen on Android and iOS (where operating system 3D elements and flashy textures have almost dissipated) and on Windows 8. Also included in this list are many Google services such as Gmail [19].

Due to the fact that the developer for `Grapha` had little to no prior experience in developing nice looking graphical user interfaces, a relatively straightforward design was created. `Grapha's` interface is the result of trial and error, combined with a lot of research and user feedback. Additionally, research was performed to find a paradigm of user interface design which promoted simple styling and an easy to use interface. The design paradigms of flat user interfaces were attempted and implemented. The result is acceptable; however, it does not compete with the work of professional designers.

5.2 Make New Graphs

In the tab named `Make New Graphs` the user will experience a screen with a few options. Most of these options are meaningless upon the first launch of the program because there have been no graphs generated yet. However, as indicated by the red box, the button named `Choose A Type To Begin` is the first item that the user

should turn his attention to. Upon clicking this box, the user will be presented with `Grapha's` repertoire of graph types as seen in Figure 5.2.

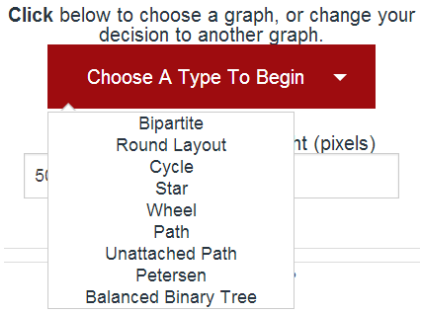

Figure 5.2: Choosing A Graph Type

The user then will click the type of graph that he wishes to generate. Once the graph type has been selected, `Grapha` will present to the user the options which are associated with that graph as seen in Figure 5.3.

The graph selected in the Figure 5.3 (bipartite) has a lot of options with regards to its layout and supports all of `Grapha's` possible features for a graph type. The `Complete?` check box tells us whether or not `Grapha` should fill in all of the edges on the specified graph. A user should check this box if he wishes the graph to be connected. The `Width` and `Height` fields specify an integer which represents the dimensions of the graph. The `Node Size` drop down is used for changing the size of the vertices while the `Text Size` field is an integer which corresponds to the size of the text within the graph. These attributes are all applied when the graph is generated.

Also present are fields for specifying the attributes of each section of the graph. A bipartite graph has two partitions or groups of vertices (referred to as sections in the code of `Grapha`). In a bipartite graph, edges may not connect two vertices in the same partition together (as per the definition of a bipartite graph). The first two fields (whose default values are "5") specify the number of vertices which each partition of the bipartite graph will have. Following those fields the user can see a `Label The Vertices?` check box. This tells `Grapha` whether or not the graph to be generated will have its vertices labelled with the number and label below. At the

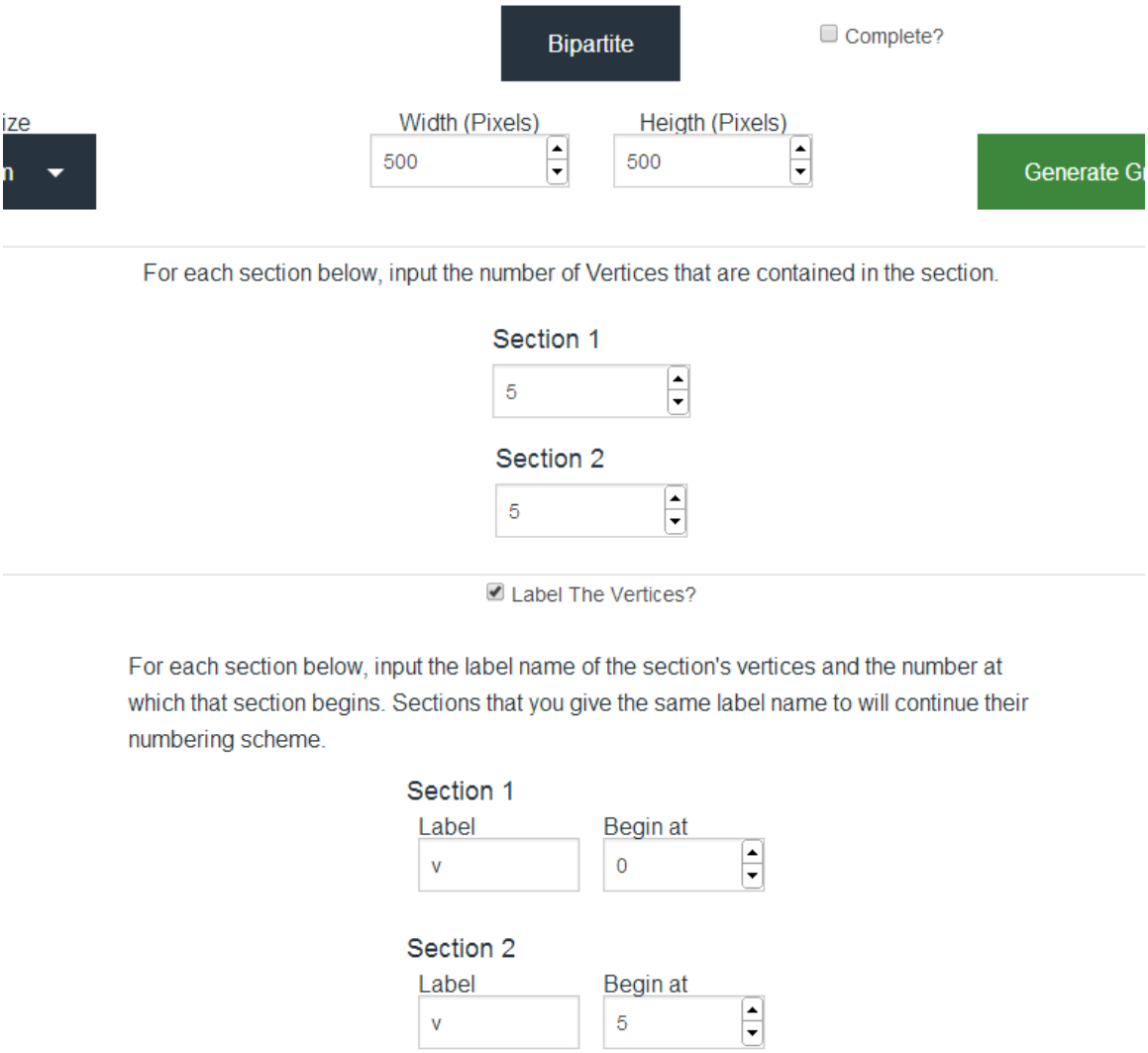

Figure 5.3: Options Of Generating A Graph

bottom of the page are the text fields which allow the user to define how a partition will be labelled. Each partition has a label name and a number to begin at. The label name can be any text and will show up in the center of the vertex, along with a number. The number field specifies what the first vertex number in the partition will be (`Grapha` automatically increments this number as it draws nodes for a partition).

Users should feel free to experiment with these options to get a better understanding of how the options can interact with graphs to be generated.

While changing the values of any number box in `Grapha`, the user has two options. The first is to input a number as usual by typing it into the box (or filling it out via voice or some other method). The second is to use the arrow keys (up/down) to automatically increment or decrement any number field.

Finally, once the user has detailed a graph to his specifications, he should click the `Generate Graph` button. Upon clicking this, the user will automatically be taken to the "`Edit Basic Graphs`" tab where his graph has been generated and where there are more options available.

5.3 Edit Basic Graphs

If a user enters the `Edit Basic Graphs` tab without generating a graph, he will see a very minimalistic view. In order for the user to bring up a graph that he wants to edit he should click on the `Load Graph` drop down. This will allow the user to load a graph and `Grapha` to populate the `Edit Basic Graphs` tab with the options available to that graph (as shown in Figure 5.4). It will also show the user the generated graph in the blue outlined box at the bottom of the page. The other way in which a user can enter this tab is by clicking the `Generate Graph` button on the first tab, once he has populated all of the specifying fields. The user will then be taken to the second tab where the contents have been changed to show the user his generated graph and the options pertaining to that graph.

Either way, once a user has loaded a graph he will be able to perform a few tasks. First of all, the user can edit his graph. All of the options available on the first page (with the exemption of graph type) are available to be edited here. Again, the user may use the arrow keys on his keyboard to edit any number field. When the arrow keys are used the graph is automatically updated, this allows the user to edit his graph on the fly. The user will also note that there are a few additional options available, the first of which is the option to save his graph. At any time the user can decide to save a graph to his cache; this means that the graph will be available from

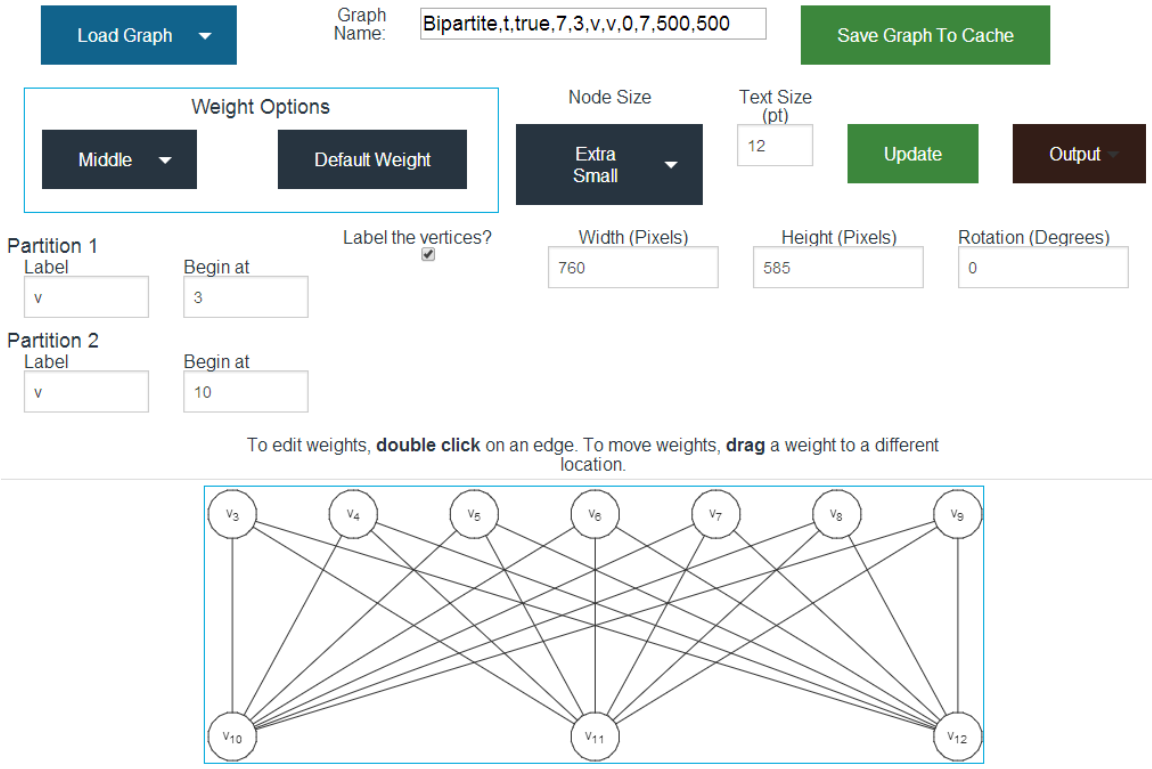

Figure 5.4: Editing Basic Graphs

any load menu where basic graphs can be loaded. The name which is given to the saved graph is specified by the `Graph Name`: labelled text box which is located on this page.

The second new option is the ability to edit weights. The user can either edit all of the edge weights at once with the buttons in the `Weight Options` box or the user can edit each edge's weight individually by clicking on a weight and dragging it or by double clicking on an edge. Double clicking will bring up a prompt which will ask a user what new weight that he would like. Clicking on a weight and dragging it in any direction will allow a user to change where the weight is drawn relative to its edge.

The third new option is the ability to output the graph in a specified format (such as PNG, JPEG, `TikZ` code, etc.). The preview graph in the blue box at the bottom of the tab shows the user a representation of what his graph looks like. This image corresponds closely to the graph which will be seen by any output method.

Also added to this page is the ability to rotate a graph. The user can place any rotation into this box to have the graph rotate around its center. After the user has changed any options located on this page, clicking the update button will update the graph in `Grapha`'s code and allow the user to preview his updated graph. However, if the user used the arrow keys (up/down) to edit a number box then the graph will automatically be updated every time the number changes. After any edits the graph may be saved under any name for use later, or output now for use in the user's desired program. Cached graphs will be stored in memory until the user clears his browser cache or deletes the graph manually. This means that the graph is persistent across multiple launches of the software.

5.4 Save, Load, and Delete Graphs

On the `Save, Load, and Delete Graphs` tab the user has the ability to save his graph library. This can be done by clicking the button named `Save All Graphs As A File`, as seen in Figure 5.5. This includes every graph, both basic or complex, that the user has in his cache. Saving a graph library will save a file to the user's specified location (hard-drive, USB stick, etc.) which contains all of the user's graphs. `Grapha` can later read this file and load all of the saved graphs back into the user's cache. This feature also allows multiple users to share graph libraries across multiple computers. It saves the file under the name that the user has entered into the box above the button. The default name contained within that box is "`Library.txt`".

The second feature of this tab is the ability to load a graph library. This is done slightly differently depending upon the browser that the user is running `Grapha` within and under which operating system the user is running the browser. In any case, the ability to load a graph library is located in the blue box labelled `Choose A Library To Load`. Within that blue box, the user should click on a button which will allow him to begin browsing his file system for the file that he would like to load. Once a user has chosen this file, he will be presented with two options: `Overwrite` and `Merge`. If the user chooses the overwrite option, all of his cached graphs will be replaced with the library stored in his chosen file. If the user wishes to keep all of his

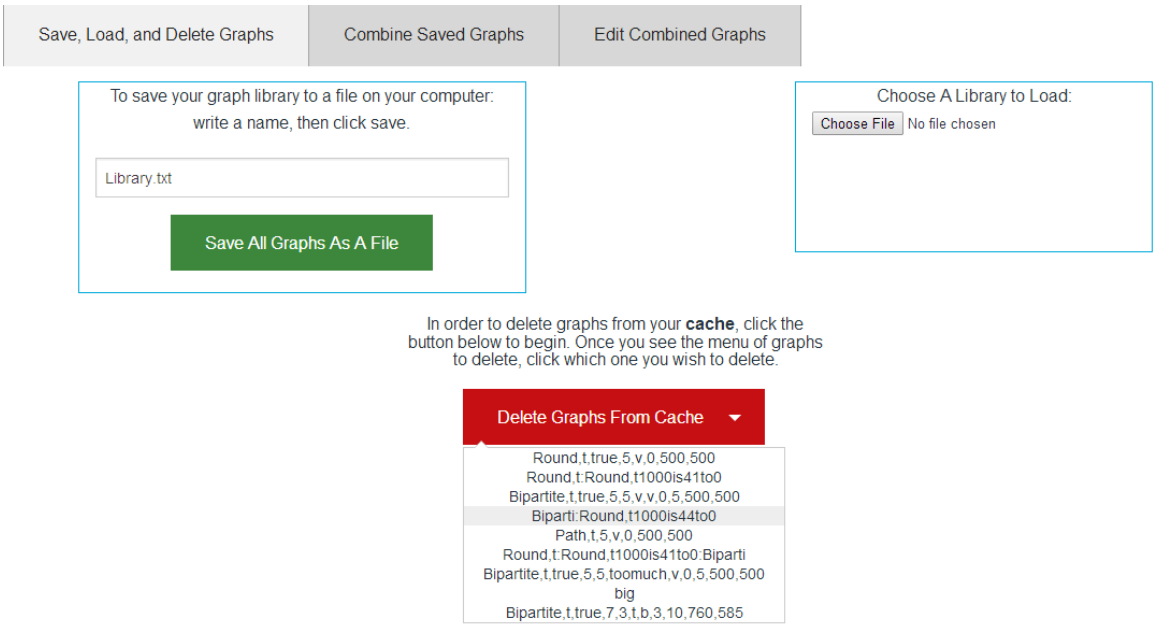

Figure 5.5: Save, Load, and Delete Graphs

graphs which are in cache and simply add the graphs from the file into his cache, the user should click the button labelled `Merge`.

Finally, this tab allows the user to delete graphs from his cache. When a user clicks the `Delete Graphs From Cache` drop down button, a list of the names of all of his saved graphs appears as seen in the picture above. After that, the user can simply click on any graph which he would like to delete, then in the confirmation box, click `ok`.

5.5 Combine Saved Graphs

In the `Combine Saved Graphs` tab the user has the ability to identify any graphs in his cache with each other. As seen in Figure 5.6, the tab has very few options. It has two drop down menus for loading graphs as well as an option to tell `Grapha` which graph is to be used on top, i.e., which graph gets to keep its vertex.

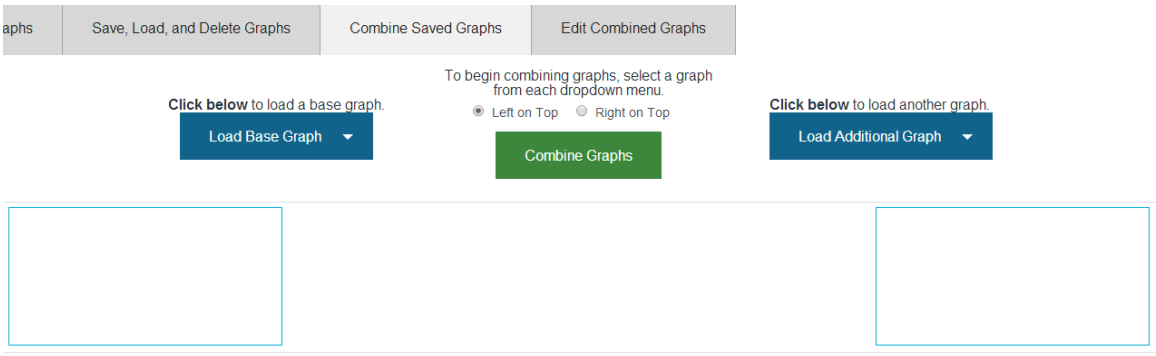

Figure 5.6: Combining Graphs

Upon clicking either of the load menus the user will be presented with a graphical representation of every graph which he has saved within his cache, as seen in Figure 5.7.

Once this representation has been rendered, the user can browse and click on the graphs that he would like to combine. Clicking upon one of these thumbnails will bring the graph into full view. Once a user has loaded a graph on each side of the screen (one for each load menu) the user can specify how they are to be combined. His first option is to simply combine them: this will simply put both graphs side by side. The second option is to click a vertex of each graph. This will tell `Grapha` which two vertices are to be identified when the graphs are combined. In addition, the user should click the radio button which corresponds to his preferred dominant graph. If the user would like the right graph to retain its identified vertex then the radio button `Right on Top` should be checked. Otherwise the left graph will retain its vertex during the combining process.

In any case, when the user clicks combine graph while two graphs have been loaded, the program automatically turns the two graphs into a larger compound graph. The user will automatically be taken to the `Edit Combined Graphs` tab and his graph will be shown to him in that tab's blue box at the bottom of the screen.

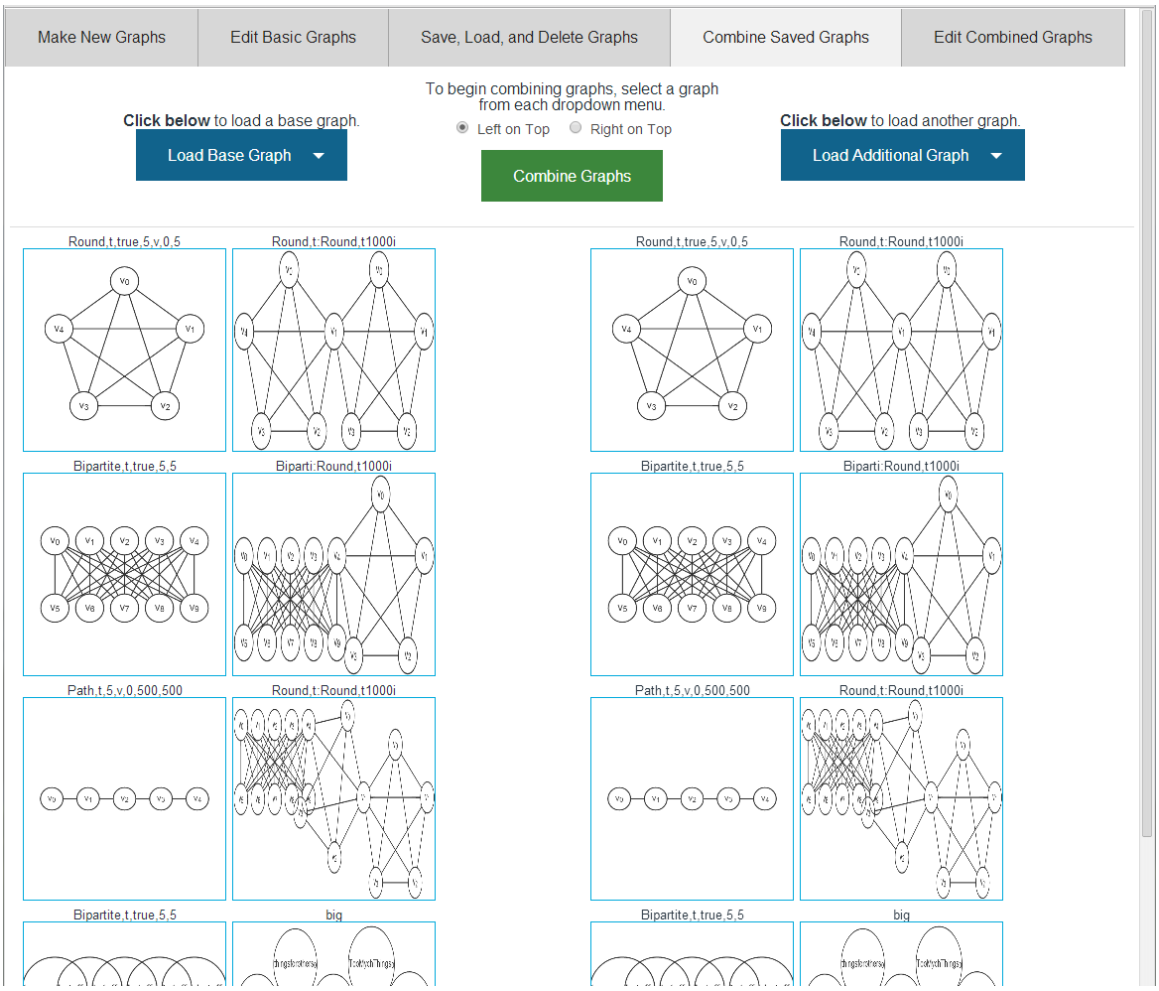

Figure 5.7: Loading Graphs To Combine

5.6 Edit Combined Graphs

Similar to how the `Edit Basic Graphs` tab worked, the `Edit Combined Graphs` tab (which is the rightmost tab) can be entered in two ways. The first way is by combining two graphs on the previous tab. Doing so will automatically take the user to the last tab and generate a preview of his combined graph. In addition, the user could enter this tab on his own. When arriving at the `Edit Combined Graphs` tab in this manner the user will need to click the drop down named `Load Compound Graph` in order to begin using this tab.

Clicking `Load Compound Graph` will show the user a list of all of the cached compound graphs that he currently has saved in `Grapha`. The user can then click which graph that he would like to edit. Once the user has clicked this graph the tab will be populated with a preview of his graph. The user can now begin to edit this compound graph.

The first way in which a user can edit his compound graph is by clicking upon a basic graph contained within the compound graph. This will populate some fields with all of the options available to edit that basic graph. These options are labelled `Highlighted Graph Controls`, as seen in Figure 5.8. The second way in which a user can edit a compound graph is by clicking the `Edit Entire Graph` button which is sitting above the preview. Clicking this button will reveal a number of options that allow a user to edit the entire compound graph at once. These options are contained within a blue box labelled `Entire Graph Controls`, as seen in Figure 5.8.

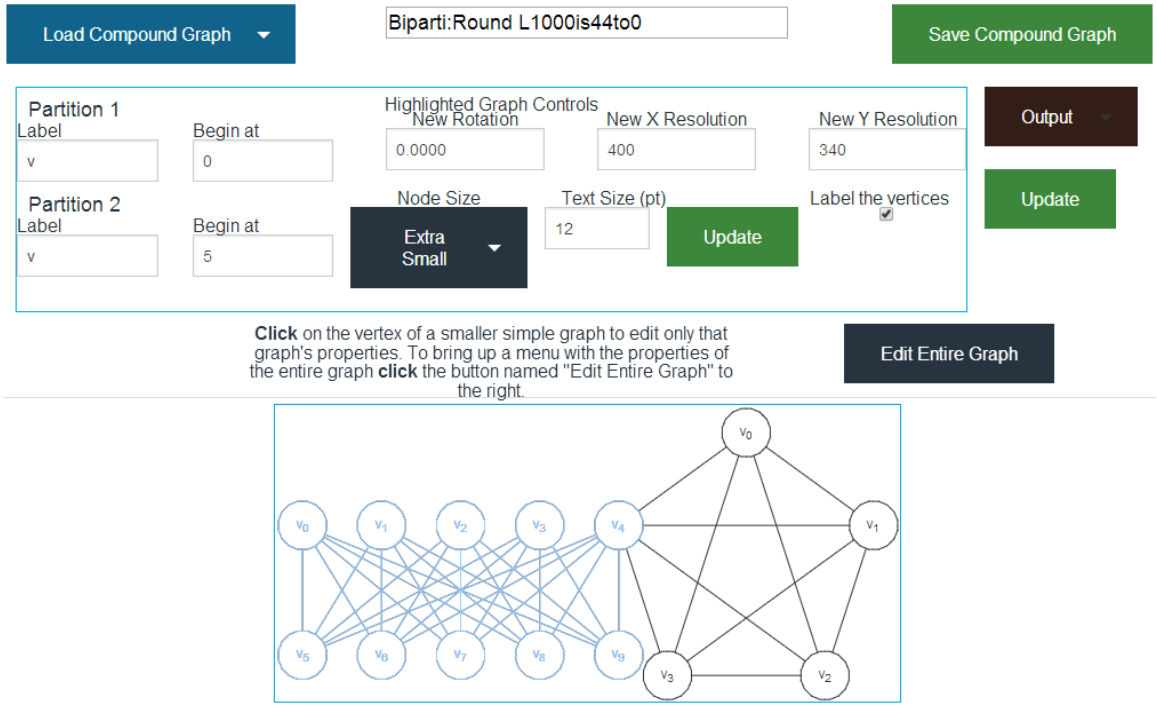

Figure 5.8: Edit Combined Graphs

Within the `Highlighted Graph Controls` box a user can edit all of the properties of a single graph. The only things not editable here are the edge weights. Having edge weights editable here would have hurt the flow of the software and also would have cluttered the user interface too much. Other than that most of the options seen within that box are the same as the options seen in the `Edit Basic Graphs` tab.

The second box seen on the screen named `Entire Graph Controls` holds options which will manipulate the whole compound graph at once. The first option here is `Label Vertices?`. This check box will turn on or off the vertex labelling for every graph within the compound graph. The next set of options for the entire graph are horizontal and vertical flip. Clicking either of these options will flip the graph horizontally or vertically, respectively.

Also available is a convenient `Auto Number` button. This button causes `Grapha` to traverse the entire compound graph and number each vertex in an increasing fashion. It first prompts the user for the lowest number in the vertex numbering sequence. This allows the graph to be consistent (no two vertices with the same number) and look neater. It is a quick way to edit everything at once. The final option in the `Entire Graph Controls` repertoire is the ability to change the text size and node size of the entire graph at once. Clicking this drop down menu will allow the user to select a new size for all the the compound graph's vertices.

In addition to editing the properties of the graph, the `Edit Combined Graphs` tab also has the ability to output, preview, and save the graph, just as the `Edit Basic Graph` tab did. The "save" here is a save to cache.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In Section 3.1 the goals of `Grapha` were laid out. The software has achieved all of those goals and has a few extras as well. `Grapha` can generate a repertoire of basic graphs very easily with only a few interactions needed from the user. `Grapha` is quick to use and has many advanced features such as combining graphs. Many of the advanced features only require a few clicks and can all be done quickly. The software is easy to use due to the user interface being self-explanatory and straight forward. In addition, `Grapha` is quick to learn because it is not complicated by having small and specific manipulations built in. `Grapha` also has many different output formats in order for the user to obtain his graph for use in other programs. These different output formats allow a user to obtain his graph in one of many different raster-based images or one of a few vector-based formats. Finally, `Grapha` is portable. It runs on iOS, Android (both with slightly reduced usability and slightly fewer features), Windows, Linux, and Mac OS. Furthermore, `Grapha` is able to be hosted on-line (allowing it to be run without being installed) and off-line (so that the user does not require internet access in order to use the software). This gives `Grapha` many different options with regards to how it is run on a user's computer.

`Grapha` achieved the goals set out for it in a very generous manner. With that in mind `Grapha` fills a gap in graph-generating software. It reduces the amount of time

that a graph theorist needs to use when making basic graphs look professional. It allows these graphs to be tweaked within it, or output in the user's desired format for specific editing. The modularity of `Grapha`'s most crucial pieces (graph types and output methods) is designed to make it long-lasting and able to cater to niche markets. `Grapha` also will run on many more devices than most other solutions and can be run on-line or off-line. These things place `Grapha` amongst the other software solutions as a time-saving, modular, and highly compatible solution to generating graphs. A version of Table 2.1 has been updated and is shown in Table 6.1. It was updated in order to include `Grapha` in the list of software.

Table 6.1: Tables of functionality for researched software and `Grapha`.

(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

During the development of `Grapha` many of the methods used to create the software (which were decided upon in the design stage) worked very well. HTML5, CSS3, and JavaScript were very good programming language choices for the software. They enabled many of the different features of `Grapha` to be implemented smoothly and across many platforms.

Midway though the development of `Grapha` a problem was discovered. The user interface was too static and did not change dynamically enough based upon the user's device or the programmer's desired changes. This made changing the user interface the most time consuming part of working on `Grapha`. A future programmer would do well to make the user interface more modular and allow future programming to be done quicker.

6.2 Future Work

The current version of `Grapha` achieves the goals which were set out for it. It is a full software suite which allows for the quick and easy generation of basic graphs or combinations of basic graphs. Despite this, `Grapha` can still improve in many different ways. The most prominent way that it can improve is in the addition of plug-ins for the software. In addition, the many modular properties that `Grapha` has can be expanded upon.

One idea for future work would be for a programmer to expand the number of basic graphs which `Grapha` implements. Right now the software has a simple repository of some common basic graphs. However, one can be certain that no group of people could think of every possible basic graph to be implemented. The more basic graphs that the software has the more reach that it has. It would also give `Grapha` a wider audience and allow it to be more powerful in terms of which graphs can be created and what can be accomplished by combining certain graphs together.

Related to the last suggestion is the idea of expanding what a graph type object has the capability to do. Right now the user interface cannot be directly changed by the graph type objects; instead, the user interface is automatically changed based upon the values that the `GraphType` object holds within its arrays. The ability to

change the user interface and have a larger number of (and more descriptive) input fields could be quite valuable to niche graphs or families of graphs which do not use sections filled with vertices but rather use something along the lines of columns and rows.

Another area for future work would be to increase the number of outputs that `Grapha` supports. `Grapha`'s number of outputs right now is powerful yet small. The software has many raster-based formats but only two vector-based formats for output. Although the currently implemented formats are judged to be very useful formats, a future programmer could easily expand upon them to increase the number of output formats. This would once again allow `Grapha` to appeal to a wider audience and even fill some niche gaps in graph generation, by supporting scarce or obscure output formats.

The next case for future work includes increasing the number of ways in which `Grapha` can combine graphs. It was brought up during the development of the program that having a way in which edges could be identified would be very useful for the graph theorist. In addition, one could think of many different ways in which two graphs could be combined. A future programmer would need to learn about the inner workings of `Grapha` and learn about the code written for the software in order to add additional connection types. Despite this, the functionality for different connection types has been worked into the program already. Therefore, it would not take a reworking of the entire program to add more connection types. Rather, the programmer would be adding more code to `Grapha` which would enable users to identify two graphs with their new desired connection type.

Currently, `Grapha` allows the user to edit or manipulate the various aspects of combined graphs. Although `Grapha` is to be used for quick and easy generation of graphs, an ability to group together many different basic graphs within the "`Edit Combined Graphs`" tab would be a good feature to have. This would allow the user to make many edits to a large number of basic graphs at once. In addition, the ability to edit more parameters regarding the entire compound graph would also be useful. Attributes such as rotation on the entire compound graph could be implemented.

Finally `Grapha` could use some plug-ins which enhance the functionality and granularity of the generated graphs. Right now there is no plug-in architecture in `Grapha`, so an interested developer would first need to create some plug-in architecture. After creating such functionality, plug-ins could be developed for `Grapha` which allow graphs from the program to be edited on a vertex-by-vertex or edge-by-edge basis.

Future plug-ins could allow for the custom movement or colouring of nodes or even allow for the ability to have specific edges and nodes to have unique styles. Possible features include removing nodes, edges or sections of graphs and adding new edges, nodes, or sections of graphs. These things were excluded from the original specification due to the fact that `Grapha` was meant to be fast and lightweight. Although this goes against the quick and easy graph-generating work flow which `Grapha` encourages, it might be very useful for a user to take his generated graph and (inside `Grapha`) edit the fine details of the graph in the same program. This way a user can have the graph suit his needs perfectly at the time of outputting it. In addition, the feature of editing multiple basic graphs which are part of a compound graph at once would be a great feature for `Grapha` to support. This way making edits to compound graphs would be quicker for the user.

Some future plug-ins could also allow for alternate ideas regarding each output. For instance, the `TikZ` output could have the option to only use coordinates instead of using scope. This would make the `TikZ` easier to read, yet harder to edit if you are an advanced user of `TikZ`.

Most of these areas of future work were thought of throughout `Grapha`'s development and, due to that fact, the software was written in a way which supports expansion. In cases where it was possible, `Grapha` was made to be modular and in all cases, functions from `Grapha` (which do not rely on the user interface) were made to work independently of each other.

Bibliography

[1] Creately. Creately Website, February 2014. URL http://creately.com/.

[2] Smartphone users worldwide will total 1.75 billion in 2014, March 2014. URL http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536.

[3] Alexandru T. Balaban. Applications of graph theory in chemistry. *Journal of Chemical Information and Computer Sciences*, 25(3):334–343, 1985. URL http://www.ijaiem.org/volume1Issue2/IJAIEM-2012-10-11-017.pdf.

[4] Nathann Cohen. Sage in graph theory. ALGO Research Group, February 2014. URL http://www.steinertriples.fr/ncohen/tut/Graphs/.

[5] E. Estrada. Graph and Network Theory in Physics. *ArXiv e-prints*, February 2013.

[6] Michel Goossens Frank, Mittelbach. *The TAXTEX Companion*. Addison Wesley, 2004.

[7] Gephi. Graphi makes graphs handy. Gephi Website, February 2014. URL https://gephi.org/.

[8] Andy Graulund. GraphJS, February 2014. URL http://dl.dropboxusercontent.com/u/4189520/GraphJS/graphjs.html.

[9] NCTM Illuminations. Graph creator. NCTM Illuminations Website, February 2014. URL http://illuminations.nctm.org/Activity.aspx?id=3550.

[10] Ian Jacobs. HTML5 Definition Complete. W3C Website, February 2014. URL http://www.w3.org/2012/12/html5-cr.

[11] Stefan Kottwitz Kjell Magne Fauske. TEXample.net, March 2014. URL http://www.texample.net/tikz/.

[12] Donald KNUTH. *The TEXbook*. Addison Wesley, 1996.

[13] Paul Krill. Javascript creator ponders past, future. InfoWorld, February 2014. URL http://www.infoworld.com/d/developer-world/javascript-creator-ponders-past-future-704.

[14] Taylor Martin. User interfaces are going flat. what comes after that?, March 2014. URL http://pocketnow.com/2013/06/05/flat-user-interface-design.

[15] AT&T Labs Research and Contributors. Documentation. Graphviz website, February 2014. URL http://www.graphviz.org/Documentation.php.

[16] AT&T Labs Research and Contributors. Graphviz. Graphviz website, February 2014. URL http://www.graphviz.org/Home.php.

[17] Anita Singhrova Suman Deswal. Application of graph theory in communication networks. *Journal of Application or Innovation in Engineering & Management*, 1 (2):66–70, 2012. URL http://pubs.acs.org/doi/abs/10.1021/ci00047a033.

[18] ECMA International Sun Microsystems. ECMA Script Documentation. exmascript website, February 2014. URL http://www.ecmascript.org/docs.php.

[19] Adrian Taylor. Flat and thin are in. Smashing Magazine Website, February 2014. URL http://www.smashingmagazine.com/2013/09/03/flat-and-thin-are-in/.

[20] CONTEXT Team. Context garden, March 2014. URL http://wiki.contextgarden.net/Main_Page.

BIBLIOGRAPHY

[21] GraphTea Team. Graphtea. GraphTea Website, February 2014. URL http://graphtheorysoftware.com/.

[22] Sage Team. Sage. Sage website, February 2014. URL http://www.sagemath.org/.

[23] Zurb Team. Foundation, February 2014. URL http://foundation.zurb.com/.

[24] w3schools. HTML5 Canvas. w3schools website, February 2014. URL http://www.w3schools.com/html/html5_canvas.asp.

[25] w3schools. CSS Tutorial. w3schools.com, February 2014. URL http://www.w3schools.com/css/.