University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Visualization of Classical Graph Theory Problems

Fatma Al-Sayegh

20 November 2022

Abstract

Whereas, formalism in mathematics gives the subject a structure and a level of abstraction which makes it applicable to the most general of the situations. Visualization of a concept on the other hand, restricts the theory to a particular example but it makes one to see a problem in a more concrete pattern. It may also allow the learner to see the same problem in a new light. The learner can then apply or extrapolate the learning to other instances of the problem in general. In this project we have tried to elucidate some classical problems in Graph Theory by the way of visualization on a Web application. The method of visualization are animations and user interaction with animations. Such methods, it is belived can help young students and self-learners to get the first brush with the subject of Graph theory.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Fatma Al-Sayegh Date: 20 November 2022

Contents

1 Introduction

There are various kinds of phenomenon in nature and computer science which can be modelled as graphs. Graphs can be thought of as points (called vertices or nodes) related to each other by edges. The points can be humans in a social netwrok, genes in a gene expression network, proteins and metabolites interacting with each other in a cell, various living species in an ecology or a food-web. The nature of relations between such objects can be causation, interdependence, interaction, etc. Neural Networks, both natural and artificial are graphs too. There is a new field emerging in computer science and statistical mechanics called *Complexity*, which promises to enlighten us about phenomenon such as life, ecology, intelligence, society and economy. Graphs stand as one of the fundamental tool to model and study such complexity. See Chapter 1 of Gros (2015).

Reducing a real world problem to a graph may amount to loosing details but as graph theory has developed a number of tools for analysis and insight these tools become available immideately for a well formed inquiry. Therefore once a concept in Graph theory is learned then just like other species in Mathematics such as integers or vectors etc, they can be applied in a variety of fields of study.

Although the most dependeble way to study Graph Theory should be mathematical formalism to state the terms, definitions and theorems; visualization of such concepts can act as a first stepping stone for the uninitiated. It can also act as an aid for a practitioner to enrich his understanding or to view the same concept the same concept in a different light.

It's true that using a particular example to define a topic in mathematics may diminish it's generality. But a particular instance of a topic can give a concrete confidence to a student about his understanding of the topic in consideration. It is expected that a concrete example can be extrapolated by a student's mind for a variety of situations and also a general understanding.

2 Background

This chapter discusses concepts which are essential to understand the forthcoming chapters in this report. It starts out by litrature review discussing the books and research papers refered and consulted for this project.Then it goes onto discuss Graph Theory definitions and it's classical problems, which are useful to understand their implementation in the app. Discussion of prior work on visualization of computer science topics which have an influence over this project is also an part of research.

2.1 Literature Review

The books and the research papers which were referred to for this project range from theoretical texts on Graph Theory, to Programming books on functional programming and the functional programming language for web front end called Elm.

2.1.1 Graph Theory and Algorithms Literature

Parts of the following literature was used to get familiar with the basics of graph theory, some basic definitions and theorems and also explanation of some classical graph theory problems. This literature also played a role in shortlisting the graph theory problems which were chosen for the purpose of this project.

Networks: An Introduction. This book was consulted for definitions, explanations and understanding of basics of Graph Theory and it's problems. It discusses the mathematical and computer science perspectives of the subject in dedicated sections. See Newman (2010).

Algorithm Design. This book lent the description and the example of *Tree-width* problem incorporated for an animation in the application. See subsection 2.2.6 for the explanation of the topic. In general, this book covers most kinds of data structures and and has excellent portion on mathematical and computer science aspect of Graph theory. See Kleinberg and Tardos (2006).

2.1.2 Functional Programming and Elm Programming Literature

In this section, literature covering functional programming and in particular Elm programming language which were helpful in this project is reviewed.

Why Functional Programming Matters. This paper is a terse tutorial using a Haskell like programming language called Miranda, expositing the power of functional and modular thinking to glue functions together to create and manipulate data structures like lists and trees. It also discusses recurring patterns in programming and their abstraction in form of higher order functions such as map, filter, and the folds. In a steep learning curve it goes onto discuss some advanced concepts in Artificial Intelligence. See Hughes (1989).

Elm in Action. This book can be used as a step by step tutorial introduction to the Elm language and it's application in creating real world web apps. It starts with small applications and gradually moves to the art of managing projects of considerable size. It acts as a bridge between functional

programming and front-end design; it also dwells into front-end design recipies like how to implement a single page application. See Feldman (2020).

2.1.3 Aesthetics

The Beauty of Simplicity. This paper provides arguments for how optimal simplicity, minimalism and intuitiveness makes a user interface more usable and trustable. This paper was the inspiration for keeping the website simple with the minimum of complicated control and features. See Karvonen (2000).

2.2 Discussion of Classical Graph Theory Problems

This section formally discusses the concepts of Graph theory which are elucidated visually in the application. If you are already familiar of the topics discussed in this chapter then by all means skip over. If not, then it's recommended to go through the section as the material discussed here is essential to understand the discussion in the further chapters. The forthcoming chapters will refer to the subsections here for definitions and explanations.

2.2.1 Definitions

A *Graph* $\boldsymbol{G}$, can be understood as a collection of vertices which are connected to each other by edges. A *Vertex* $\boldsymbol{v}$ can be understood as a point and an *Edge* $\boldsymbol{e}$ is a pair of vertices. The *set of all the vertices* in a graph G is represented as $\boldsymbol{V(G)}$ and the *set of all the edges* in G is represented as $\boldsymbol{E(G)}$.

For a vertex v, it's *degree* $\boldsymbol{deg(v)}$ is the number of edges connected to it. An *isolated vertex* v is such that $deg(v) = 0$. An *end vertex* is a vertex w such that $deg(w) = 1$. Two vertices are *adjacent* to each other if there is an edge connects them.

A *bipartite* graph, is a graph G such that it's vertices $V(G)$ can be split into two disjoint sets A and B such that each edge of G joins a vertex of A and a vertex of B. See Newman (2010).

2.2.2 Graph Isomporphism

Two graphs G_1 and G_2 are isomorphic if there is a one to one correspondence between the vertices of G_1 and G_2 such that the number of edges between any two vertices in G_1 is equal to the number of edges joining the corresponding vertices of G_2. Given two graphs, detecting if the graphs are Isomorphic is a problem to solve as the graphs may appear to be different in appearance and in the labeling of the nodes and edges. See Newman (2010).

Application The graph isomorphism problem finds application in the field of bioinformatics for finding network motifs (sub-graphs isomorphic to an input pattern) in a larger biological network. A network motif is a recurring pattern of connection of vertices in a large graph signifying their evolutionary selection over random patterns. See Bonnici et al. (2013).

2.2.3 Max K Cut

A maximum cut, is partioning the vertices of a graph in two groups such that the number of edges between these two groups is maximum. In a weighted graph, where the edges are weighted, the weights of the edges are also taken into consideration. A maximum k-cut, is generalized version of maximum cut, where the graph is partitioned into k subsets, such that the number of edges between these groups is maximized.

It is important to note that a bipartite graph (refer to the Definitions section above) is a trivial example of Max Cut there are no edges among the vertices of a set A and no edges among the vertices of set B and all the edges are from the vertices in set A to vertices in set B.

2.2.4 Graph Coloring

It is an optimization problem where the objective is to assign to the vertices of a graph a color such that no two adjacent vertices have the same color, while keeping the number of colors employed to a minimum. Here a color can be thought of just any symbol from a finite set of symbols.

2.2.5 Minimum Vertex Cover

Minimum Vertex cover of a graph is the minimum amount of vertices such that, all the edges in the graph must have one of such vertices as at least one of their endpoints. This is also a optimization problem in which the constraint is that all the edges must be covered while keeping the number of vertices in the set of Minimum Vertex Cover to the minimum.

2.2.6 Tree Width

We will explain in two parts. First we will define what a tree decomposition of a graph is. Then we will define tree width of the graph.

To decompose a Graph in a tree is to put nodes into sets called pieces, subject to certain conditions. The first condition is that all the vertices of G should belong to at least one piece. Every edge of G, must be present in at least one piece which contains both ends of the edge. And finally, in the tree decomposition, if there is a node n present in a walk from a node $n1$ to $n2$, and if both $n1$ and $n2$ have a vertex v in common, then the node n also contains that vertex v.

Any graph can be decomposed into a tree. Trivially, a graph can be tree decomposed by putting all of it's vertices in just one node. But it will not be a very useful tree decomposition. Therefore a good tree decomposition of a tree is the one which has small pieces. Tree width is defined as the size of the biggest piece $V_t - 1$. The smaller the tree width the bigger the better the tree decomposition. See Kleinberg and Tardos (2006) in the bibliography.

2.3 Prior Work

This project takes subtle inspirations from some of the work which is available on the internet as web applications for visualization of popular algorithms. Although the works which are discussed in this section are focused on understanding algorithmic solutions of computer science problems, the visualization of graphs, trees and lists in these projects have been inspiring for depiction of graphs and their animation in this project.

2.3.1 Data Structure Visualizations

This tool was developed by David Galles, Associate Professor, University of San Francisco. See Galles, in the bibliography section. It covers topics from various categories of computer science problems such as Dynamic Programming, Geometric Programming, Trees, Heaps, Graphs etc.

The design of the tool has several important features. The user has the facility to define his own data-structures rather than they being predefined or hard-coded. There are control buttons which allow the user to start pause and restart the animations. There is a slider to tune the speed of the animation as well.

There is a dearth of textual explanation of the algorithms while they run. Perhaps, the main purpose of this tool is as a teaching aid such that the teacher first explains the topic and uses the tool as a visual demonstration to show his students the working of the algorithm on real datastructures.

2.3.2 VisuAlgo

This tool was developed by Dr. Steven Halim of National University of Singapore. See Halim in the bibliography. It covers topics from the subject of data structures and algorithms. Most relevant for this project are the topics related to graph theory; which are Maximum Flow, Minimum Vertex Cover, Traveling Salesman and Steiner Tree, although the emphasis is on algorithmic solutions to the problems and not problem visualization which is the emphasis of this project.

For most topics the user is able to construct his instances of datastructures. Unlike Data Structure Visualization application mentioned in subsection 2.3.1, there is an ample amount of textual information in terms of theory, tutorial and instructions. The explanation of the topics is done in text blocks which appear at appropriate places in a slide show like fashion. For organizing the textual information, it has drop down content menu for easy access to various sections. Although, different positions of the text blocks can be a little distracting. In this project, the text explanation is given at one place.

2.3.3 Algmatch

This Web app was developed as a final year individual project dissertation for by Liam Lau under the supervision of Sofiat Olaosebikan. The application visualizes the matching algorithms such as Gale-Shapley Stable Matching and Extended Gale-Shapley Stable Matching algorithms applied to stable marriage and hospital/residents problem. See Lau in the bibliography. The app lends ideas about user friendliness and intuitive usage.

It has a panel which describes the algorithm steps while in an animation the matching algorithm works on an instance of the problem.

Aesthetically, the most noteworthy features of the app are, playback and speed controls which are as intuitive as media buttons on a media player and smooth page transitions resulting in a pleasing user interaction.

3 Analysis/Requirements

In this chapter, the scope of the project, the criterion of selection of the problems in graph theory, the thinking behind choosing the methods of elucidation of the selected topics will be discussed. Finally to conclude the analysis the requirements of the project are stated.

3.1 Scope of The Project

Understanding a problem in mathematics is a necessary first step in trying to solve it. It also enables a student to abstract out a formalized version of a problem from a real life scenario present in the fields of science and engineering.

This idea has guided this project to be restricted to one which helps a learner to understand the problems in graph theory. Whereas the solution or suggesting an algorithm to solve the problem, if required, is the second important step which has been deliberately not touched upon to keep the scope of this project clear, precise and specific.

3.2 Criteria for Selection of Problems

One of the most important criteria for selection of the problems for the project was based upon the importance of the topic in the field of graph theory. There are several text books (see Newman (2010) and Kleinberg and Tardos (2006)), in graph theory, which discuss various theorems and problems in the subject. There are a few problems which occur commonly and frequently in them. The order of their inclusion in the text books is based logically. Building the concepts from the basics to advanced. Therefore the problems included in the project should represent all levels of difficulty.

Since imagining a graph theory problem is largely a visual exercise, there was no dearth of problems which could offer themselves as a subject of an interesting visualization. The additional criteria therefore for filtering the candidate problems was based on whether they could be elucidated in the form of a simple and meaningful example, employed for animation and user interaction. The simplicity of the example doesn't in anyway imply triviality of the problem. Indeed here the assumption is that a simple example problem can make a student reach to the heart of the concept in it's generality fairly quickly. From there on she can extrapolate the learning to more complicated examples.

It is important to mention here that a survey among peers in the field of software engineering and computer science was conducted to gather suggestions on the shortlisted topics and methods. The data from the survey had a role in determining topics and the methods chosen.

3.3 Methods of Elucidation

As it has been discussed in the previous section that feasible methods of elucidation/exposition played a prominent role in the selection of the problems in the first place. It was decided that

for this project, such methods can be broadly classified as animations and user interactions or a combination of both.

3.3.1 Animation

Humans are primarily visual learners from the days of hunting and gathering. We sometimes like to imagine even the most abstract concepts visually. Therefore Computer animations have been increasingly adopted to elucidate complex concepts in mathematics and science. Animations are used in this project to make problems like Graph Isomorphism, Max Cut, and Tree Width. For instance, the example problem of Graph Isomorphism, was explained by morphing a graph to change it's shape to acquire another radically different shape. It will be explained later how it can be considered as a visual proof that the two graphs in the scene are thus isomorphic.

3.3.2 User Interaction

Although animations go a long way in terms of having a user's mind involved in the learning process they only offer a linear narrative. On the other hand, user interaction with an animation not only makes the experience more immersive, it also leads to a natural multiplicity of stories from a single program. This happens as a human input results in a novel path from one state to another in the program just like a video game.

3.4 Technical Scope of The Project

To achieve the above mentioned features The program has data structures to hold graphs and algorithms to visually and geometrically manipulate them.

It's important to note here the program does not contain algorithms to solve the listed problems. As the scope of the program was limited to the purposes of visualization and not coding the algorithms which can solve instances of the mentioned problems. Therefore in this project, although the data type of Graphs (Set of Vertices and Set of Edges) and the associated functions, are quite general and can support operations of various kinds, care is taken that I provide the solution to the visualization program before hand to give enough information to the various animations and user interactions.

3.5 Requirements

Based on synthesis of the above sections it is concluded that there must be a web application written for the desktop browser with user interactive animations. The requirements for the application are categorized into two priority levels. The application must definitely have the features mentioned in the *Must Have* section which is the minimum requirements necessary to have a functioning application without any bells and whistles. It should have the additional desirable features mentioned in the *Should Have* section relating to visual aesthetics, wider use and code-extendability. Finally to precisely define the boundries of the project there is section of *Will Not Have*.

3.5.1 Must Have

The application must elucidate the following enumerated *Classical Graph Theory Problems* by employing user interactive animations of their respective examples.

1. Graph Isomorphism
2. Max Cut

3. Graph Coloring
4. Minimum Vertex Cover
5. Tree Width

And to achieve this by programming the following items -

1. Data structures which represent vertices, edges and graphs.
2. Display the above entities as Scalar Vector Graphics on screen.
3. Translation and shape transformation of the graphs for animation.
4. Generation and handling of events triggered by user interaction with the elements in the animation for user interaction.
5. Display appropriate text in synchronization with the animations and user interaction.

Doing so in a manner which fulfills the following subjective qualities -

1. Substantive learning impact
2. Ease of Use
3. Coherent Story telling

And finally, evaluating the application with the help of the peers on parameters which can be broadly classified into the following categories -

1. User Experience
2. Learning Impact
3. Quality of Elucidation

3.5.2 Should Have

Although not detrimentally essential for the basic utility and functioning of the application, a few desirable features should be included for wider use and additionally ease of contribution by others.

1. Pleasing Aesthetics
2. Device Compatibility, the application must be friendly to most screen sizes.
3. Contribution friendliness, in the way of good code organization and documentation.

3.5.3 Will Not Have

For the reasons elaborated in section 3.1 and section 3.4 the project will not implement algoritmic solution of the the graph theory problems. Although it will have some algorithms implemented to check if a given solution is correct specially in user-interaction tasks.

4 Design

This document discusses the design choices made for the application and the guiding principles behind such choices. It progresses to how a graph are visually displayed and animated in the app without going in the technical details of implementation. Finally, I shall discuss the intended experience of the user while going through the various topics explained in the application. We will also discuss the learning impact of each topic on the user as it forms an essential aspect of overall design.

4.1 Guiding Principles

4.1.1 Simplicity

For the purpose of elucidation of Mathematical concepts, which requires an undivided attention of the learner, it was decided that the user interface must have the minimum amount of clutter possible, without giving away the minimum amount of functionality required. The learner should not get distracted by an overpopulated user interface. Simplicity and elegance would also lead the user to stay longer on the application without getting visually exhausted. Furthermore the users on the web today are extremely goal driven and don't want any obstruction between them and their goal. See Karvonen (2000).

4.1.2 Intuitiveness

The layout of the user interface should be such that it not only has utility, but should also help communicate the intention of the designer about the usage of the application. For example a play button, just like it was found on media devices for decades, invites the user to kick-start an animation even without going through the text which tells him to do so explicitly. The size and placement of the play button on the page, therefore becomes important. Right placement of user interactive elements guide the user through the story which is intended to be told.

4.1.3 Meaningfulness

For a substantial learning impact, the elucidation of the topics must reach at the heart of the topic. Furthermore they must have a story line which is meaningful and coherent. The learning outcomes of the animation or a user-interactive task must be well defined before an attempt is made to implement them.

4.2 Wire-frame and Navigation

The web application is a **Single Page Application**. Where the navigation from one topic to another occurs according to the user inputs. When the state moves from one graph theory to another topic, the data on the screen, that is the graphics and the text, change on the same page without loading a new HTML each time. The user however will notice the url change with

navigation from one topic to another. This will also give the user the ability to use forward and backward buttons in the browser to navigate through the history of URLs visited. As a new HTML page is not loaded every time the user navigates from one topic to another the screen therefore does not turn briefly white and the transitions are imperceptible by default.

There were several iterations made for the layout of the web page, one of the is shown in ??. Finally, a layout was chosen, see ??, in which the page is vertical divided into two parts, the left part of the page contains an instance of an animation, the right contains explanation of the topic and advice on how to interpret the animation along with navigation and control buttons.

The text in the explanation part of the page is dynamic in nature, if the animation has facility of user interaction with it's elements, the corresponding text on the right responds with advises on the state of affairs and what the user should do next.

There is a navigation bar at the bottom of the page, with left and right arrows, along with the names of the previous and the next topics, to hop from one topic to another.

4.2.1 Home Page, About Page and Header Bar

The home page is the landing page of the application. It gives the user clickable icons for navigation to the topics in the application. The icons must have the name of the topic and a miniature graph which best represents the topic.

The about page has a short description of the purpose of the application with a short write up introducing the developer and the supervisor of the project. It also has an acknowledgement section mentioning the people who have been important in the completion of the project.

Every page has a header bar, which has buttons to navigate to home and about pages respectively. The bar stays at it's place even if the rest of this page is scrolled down for access to navigation.

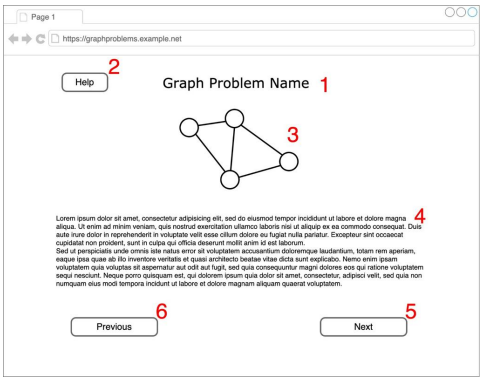

Figure 4.1: *An initial wireframe*

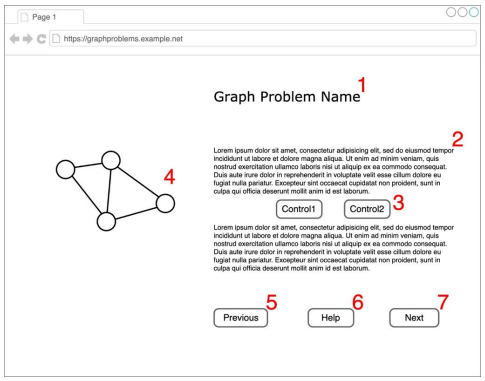

Figure 4.2: *The final wireframe*

4.3 Animation Panel

As mentioned in the preceding section, the left half of the page is for graphics, which contains an animation, a user-interaction session or a combination of both. The animation contains one or more than one graphs. These graphs undergo, according to needs of the topic in consideration transformations of appearance and annotation.

4.3.1 Visual representation of graphs

The graphs are represented as vertices and edges joining those vertices. Although the geometrical placement of the vertices is of no consequence in the subject of graph theory, for the purpose of visualization, vertices are assigned a 2 dimensional position. The edges, don't contain attributes such as length or positions of endpoints of a line segment, they are rather defined as a relation between a pair of vertices.

Appearance of Vertices The vertices of graphs in the animations are color filled circles of the same size save for certain exceptions. The color of the vertices have been allotted by varying mostly the hue and keeping saturation and lightness relatively same in the **HSL** (Hue Saturation Lightness) color space. The vertices contain a name inside the circle as an integer. The names of vertices were chosen as integers as it was assumed that such a representation would help the developer and also the user to keep a track of them as order of integers is understood better by humans.

When a particular vertex is needed to be shown differently than the rest then it's size and color are displayed differently. For example, a user-selected vertex in some of the animations are shown bigger and it's color changed to golden. The gold color it was observed makes the vertex in consideration stand out differently from other colors in the animation.

Appearance of Edges Edges, although are defined very algebraically as a relationship between a pair of vertices it gets drawn by referring to the positions of the related vertices as a straight line segment white in color.

When an edge is supposed to shown differently than the rest of the edges, it's width is increased and color changed to the same value of golden as a selected vertex. Again it was found that this color and thickness, made the edge standout from the rest of the edges and helped in showing it distinguished without being unpleasantly distracting.

4.4 Explanation Panel

Whereas, the animation panel on the left page contains all the graphical components of the topic elucidation. The right half of the page is occupied by the Explanation Panel. It contains the title of the problem being explained, it's definition and instruction on how to go ahead with starting the animation or user-interactive tasks. It also contains control buttons for starting, re-starting and pausing animations. For user-interactive tasks it has buttons to reset the task.

As the animation or a user-interactive task progresses the explanation panel generates explanatory and instructive text.

The explanation panel also has navigation panel with buttons to navigate from one topic to another.

4.5 User Stories for Elucidation of Topics

This section describes, what the user is intended to experience while interacting with the individual topics in the application with a special consideration towards learning impact and understandibility.

When the user opens the web application on a browser of her choice, the first topic she sees is that of Graph Isomorphism. She may stay there to interact with the example of the topic or navigate to other topics using navigation buttons at the bottom to have a bird's eye view of other topics.

Each graph theory problem has it's own character and require a different approach for elucidation. The following sections will explain these approaches with their intended experience on the user

with learning outcomes which may be achieved.

The survey mentioned in the section 3.2, also gave quite a few suggestions on different ways to elucidate the short listed topics. The suggestions were very topic specific, and included various ideas such as animations and games and the ability to construct user defined graphs etc. Some of the suggestions could be included in the project, some could not be accomodated as they did not fit the flow of narrative, while others while being brilliant ideas, act as inspiration for future work due to constraints of time.

4.5.1 Graph Isomorphism

The user is presented with a graph on the left and textual explanation on the right of the page. The textual explanation portion of the page also has some media buttons such as play/pause and reset to interact with the explanation. The text explanation briefly defines graph isomorphism and advises the user to press the play button.

Animation: When the user presses the play button, a new graph emerges out of the old one while the keeping the edges between any two vertices conserved. While keeping the connectivity between the vertices intact, the graph transforms into a completely new shape, almost giving a visual proof that the two graphs on the screen are isomorphic to each other.

User Interaction: After the two isomorphic graphs have separated from each other, the user is advised in a text panel to choose a vertex by either hovering over a vertex or pressing the corresponding number on the keyboard. Doing so, will change the visual appearance of the selected vertex, the edges incident on the selected vertex and the adjacent vertices to the selected vertex in both graphs. The selected vertex will be enlarged to a new radius and change its color from it's original color to golden color making it stand apart from the rest of the vertices. The edges incident on the vertex will change their colors to the same gold color. The adjacent vertices to the selected vertex will form a golden halo around them. This color transformation will distinguish a kind of a subset in the two displayed graphs. At this point of time, in the text panel the user is pointed out that the selected vertex has the same number of edges connecting to the same adjacent vertices in both the graphs. The user is also advised to inspect other vertices of the graphs and convince herself that each vertex of the graph has the same adjacent vertices in both the isomorphic counterparts.

Learning Impact: The transformation of a graph into a radically different looking graph but being essentially the same as far as the connectivity between the vertices go acts as a visual proof that the graphs are isomorphic. While individually inspecting each vertex will re-confirm this idea to the user. After having experienced the concept of graph isomorphism in this way it is assumed that the concept and definition of the term would be clear to her.

4.5.2 Max k Cut

Max Cut has two animations one after the other. The first animation is about Max 2 Cut and the second is about Max 3 Cut. It is assumed that the user will extrapolate the concept of the general Max k Cut after understanding the first two cases ($k = 2$, $k = 3$) and extrapolating it over greater values of k. The examples shown in the animations are a nearly bipartite and a tripartite graph for $k = 2$ and $k = 3$ respectively. A nearly bipartite and a tripartite graph is used to elucidate the topic as it is easier for the user to visualize how the two graphs can be segregated to sets of vertices such that the maximum number of edges pass between such sets.

In both the cases of $k = 2$ and $k = 3$, the weight of all the edges is taken to be equal to 1. This decision has been taken as with $w = 1$, the answers to both the max cut problems are more visual than unequal weights.

Max 2 Cut Animation: It starts with an Original graph on the left and definition of Max K Cut and Max 2 Cut on the right of the web page. The right part of the page also contains media buttons to pause and play the animations. It also has a button for switching from Max 2 Cut example, to Max 3 Cut example. The Max 2 Cut animation starts with a graph, which starts when the user presses the play button. As the animation progresses a new graph emerges out of the original one and translates towards right changing it's shape to segregate it's vertices into two sets forming a Maximum 2 Cut. The two sets move vertically up and down and increase the distance between themselves, revealing the number of edges passing from one set to another which the user can intuitively tell is greater than the number of edges between any other two sets which may have been formed from the vertices of the graph.

The user is advised to put up a pre-defined horizontal line by pressing a button in the explanation panel. The line is drawn between the two sets of vertices. The intersection points between the edges and the max cut line is shown by blue dots. These user is advised to observe the number of intersection points which tell the user the number of edges passing from one set to another.

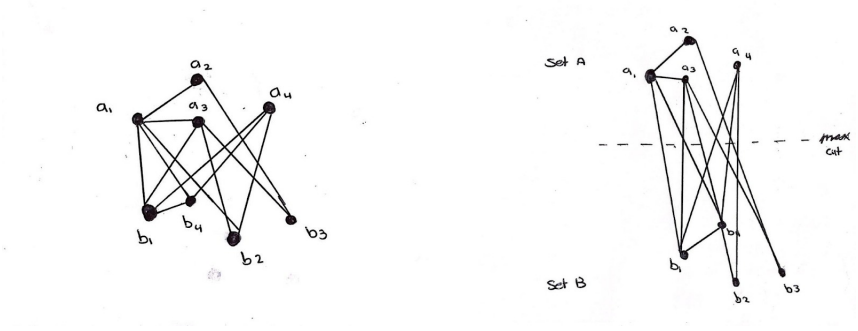

Figure 4.3: *Initial design of Max 2 Cut*

Max 3 Cut Animation: Just like the animation for Max 2 Cut, this animation is started by the user by pressing the play button. The animation on the right starts with a tripartite graph arranged in a circular form. As the animation progresses the graph gets Divided into three sets of vertices in which the three sets translate in directions which are set 120° apart from each other. The graph transforms from a circularly arranged one to a triangular form. The user can draw the Max 3 Cut lines at any point in the progress of the animation. These are three lines separating each set from the rest of the graph, with points of intersection shown in blue showing the number of edges passing from one set to the rest of the sets.

Learning Impact: Although the examples in the Max 2 Cut and Max 3 Cut can be seen as simple ones as the first one was nearly a bipartite graph and the second one was a tripartite graph, they do a good job at defining the problem well. Not just that such visualization explains the problem well it also may act as artwork especially in the case of Max 3 Cut, which may inspire an imaginative student to want to investigate the subject further.

4.5.3 Graph Coloring

User Interaction: A user-interactive task was chosen for explaining Graph Coloring as the nature of the problem lends naturally for such. The user is presented with a graph which has all the vertices in color white. On the explanation panel to the right he is given the definition of the problem and advice on how to complete the task.

The task is to choose colors from a color palette of three colors namely red, green and blue, and color vertices in the graph such that no two adjacent vertices have the same color. Whenever the user colors two or more adjacent vertices the same color the text panel warns them to make

amends. There is a reset button in the explanation panel to un-color all the vertices to start all over again if the user wants to start from the beginning.

The user is challenged to first challenged to color the graph successfully in just two colors but as it would be soon clear to the user it can only be done in three.

A user-interactive task was chosen for explaining Graph Coloring as the nature of the problem lends naturally for such.

Learning Impact: The user interactive task will help the student retain the meaning of the Graph Coloring problem in their memory for a long period of time than just watching an animation as a spectator.

4.5.4 Minimum Vertex Cover

User Interaction: Minimum Vertex Cover, by the nature of the problem is chosen to be elucidated by the help of a user-interactive task. The user is given a graph along with explanation of he Minimum Vertex Cover problem. He is also explained how to complete the task. The task is to select vertices successively either by clicking them or pressing a number corresponding to the vertex of choice on the keyboard. When he selects the vertex, the selected vertex and all the edges incident on it are displayed differently. He has to thus highlight all the edges by selecting the minimum number of vertices in the graph. If he has done this task effectively then he would not choose more vertices than required to cover all the edges in the graph. When the user covers all the edges by only selecting four vertices, he is given a congratulatory message for having done the task right. If he covers the graph by selecting more than four vertices then he is advised to do the same in just four.

Learning Impact: Just like Graph coloring, in the case of Minimum Vertex Cover too, it is assumed that a user-interactive task is effective not just in explanation of a topic but also, retention of the concept for a long period of time.

4.5.5 Tree Width

Animation: The topic Tree Width is explained in a multi-part animation. The first part begins with a graph in which the vertices are arranged in a circular pattern. The circular form conceals a tree like structure which can be abstracted out of the graph.

The user while reading the explanation is instructed to press the 'forward' button to move to the first part. The user hops from one part of the animation to another by pressing this 'forward' button. In the first part the graph which was hitherto arranged in a circular pattern transforms into a regular lattice like pattern. The tree-like pattern is more apparent in the new visual form of the graph. This is also pointed out in the explanation panel.

The next part of the animation shows an example of a piece (a sub-graph) containing three vertices against the backdrop of the graph. The significance of pieces in the tree-width concept is discussed in the background chapter. The piece is also represented by a blue dot at the centroid of the three vertices. In the next part of the animation the whole of the graph is marked by it's constituent pieces by blue dots.

In the final part, the pieces form the nodes of a tree. The tree's edges (branches) are colored in golden color to make it stand out from the graph in the background. At this point, the definition of the tree width is given in the explanation panel.

Learning Impact: The user is expected to learn the concept of tree decomposition of a graph. Also, by the help of animations, he will be inspired to learn abstract thinking: how a *form* of a tree can be derived out of an unassuming typical graph.

5 Implementation

This chapter discusses the platform and the programming techniques employed to implement the application. This application is a web application meant to run on browser. Traditionally, Javascript or programing languages which transpile to Javascript are employed to execute such web apps.

In the implementation of this application a functional programming language called Elm which transpiles to Javascript is used. Therefore, the chapter starts with defending the benefits of using a functional programming in general and also how Elm uses Model, View, Update architecture to implement a dynamic front end.

Furthermore, it will be discussed how the Graph as a data-structure is defined. How it is drawn on screen as SVG (Scalable Vector Graphics). How a function generates a color palette for the coloring of the vertices. How vertices are laid out in various geometrical patterns. How animations are implemented and graph are made to change shape and translate in space among other things.

5.1 Front End Development with The Elm Programming Language

The project is developed using the Functional Programming paradigm. This is a paradigm which has been in development and practice since the days of infancy of computer science. Functional programming is based on a form of computation called lambda calculus proposed by Alonzo Church. See Hudak et al. (2007).

For most of the history of computing, functional programming remained in the ivory towers of universities for purposes of exploring theoretical computer science and language research.

In the last decades however, programming languages such as Haskell and a few dialects of LISP have escaped the ivory towers to find application in the software industry.

In this section we will discuss, why functional programming was chosen as the programming paradigm of choice. How the functional programming language called Elm is used to write a well organized, maintainable, intuitive and understandable code to produce a dynamic front end.

5.1.1 Why Functional Programming?

Functional programming, makes the programmer think in a different way than what may be called imperative programming. In the functional paradigm, functions are first class citizens, which can be mashed up together in myriad different ways such as the following -

1. A function given as input to a another function.
2. A function producing another function as an output.
3. Composition of two functions dove-tailed to each other to produce another function.
4. Programming patterns being abstracted out as functions.

For such Lego like usage of functions they must be dependable, such that for a particular input a function will give a particular output just like mathematical functions and has no business outside it's scope for side-effects. With such confidence in the functions, they can be fitted with each other to make them do complex computation. See Hughes (1989).

Since the functional code is more reasonable and logical than imperative programming the runtime errors are substantially less than imperative programming and is easier to maintain.

Separation of Concerns It may be asked that if functions don't have side effects, how do they print output on the terminal or read file from the hard disk or accept inputs from a user. Functional programming environments have a way of separating the pure part of a program from the impure part, by introducing 'actions'. These 'actions' or side-effects are treated as a form of encapsulated data, which can be manipulated by pure functions, and the environment makes changes to the outside world by executing these actions.

Therefore the programmer has to himself a large part of the program where he deals with just pure functions. This allows him to exploit the perfectness of pure functional programming.

This separation of concerns of pure and impure code in the context of the Elm programming language is discussed in the subsection 5.1.3 called *the Elm Architecture*.

5.1.2 Why Elm Programming Language for This Project?

For the reasons in the previous sections, a functional programming language was chosen keeping in mind that the size of this project would be quite substantial. Unlike JavaScript, Elm does not require any external framework such as Angular or React. This makes the program easier to reason with and maintainable.

Friendly Compiler Errors It has a compiler which gives friendly error messages almost guiding the programmer for correct usage of the language. Refactoring code in Elm is easy as the friendly compiler errors guide the programmer to each line of the code which need modification while refactoring.

Elm-Ui The layout of the application on the screen can be done without any direct usage HTML/CSS by using an Elm library called Elm-Ui. This package frees the programmer from using CSS styling and gives intuitive control of the page layout with the help of rows and columns (Pure functions). With Elm-Ui, multiple rows can be situated in a column and multiple columns can be situated in a row and so on and on. When elements are put in a row they are stacked side by side horizontally. When the elements are put in a column they are stacked one below each other. The elements in such a row or a column can be put at a definite alignment and spacing from each other.

Libraries and Community Elm also has rich liblraries for linear algebra, graphics and Scalar Vector Graphics which can aid in creating a *2D* graphics web app like this one.

5.1.3 The Elm Architecture

The Elm Architecture is a pattern of writing Elm code for responsive web applications. The architecture separates the concerns of front-end development into the following categories:

1. Model
2. View
3. Update

The Model is a data structure which holds the state of a program. See Fairbank (2019). This state is used by the view function as an input to render the webpage. The webpage, when rendered has elements, which may trigger events, such as user inputs by the way of clicking an HTML element. Such events are caught by the Elm runtime and sent to the update function. The update

function takes these event messages and changes the state. The changed state is then rendered by the view function to a modified page. Therefore, the Model is changed by the update function, whereas it is used by the view function to render a webpage according to a formula set by the programmer. Hence the content of the webpage reflects the state of the program.

5.2 Events as Messages

The events described in section subsection 5.1.3 generated by animation clocks and clicks of the user on graph elements and buttons, are called messages in the Elm way of naming things. For this particular application they are defined as an Algebraic Data Type as:

```
type Msg
    = TimeDelta Float                -- Clock Ticks for Animation
    | HoverOver Int                  -- Event when Mouse over a Vertex
    | MouseOut Int                   -- Event when Mouse out from a Vertex
    | VertexClicked Int              -- Event when Vertex Clicked
    | AnimationToggle                -- Pause or Play Animation
    | AnimationStartOver             -- Restart Animation
    | ToggleVertexStatus Int         -- Select/Unselect Vertex
    | NextTopic                      -- Next Topic
    | PreviousTopic                  -- Previous Topic
```

Listing 5.1: *Abstract Data Type Msg with it's Data Constructors. These messages are emanated from buttons, graph vertices system clocks and are received by the update function to change the Model, which carries the state of the program.*

The messages are not just generated by user interaction with this application, they are are also generated by the animation clock as well as can be seen in the first data constructor of the type *Msg*. The clock ticks and the key strokes are events which initiate the update function to act on Model. The animation clock and key presses need to be subscribed from the Elm runtime in the following way:

```
subscription : Model -> Sub Msg
subscription _ =
    Sub.batch
        [ E.onAnimationFrameDelta TimeDelta
        , E.onKeyPress keyDecoder
        ]
\label{listing: subsciption}
```

Listing 5.2: *Subscription of Animation clock and Key presses services. Subscriptions are used to catch the events which are emanated outside the DOM. In this code, a system clock and key presses are subscribed to by the program.*

5.3 Implementation of Graphs

Inside the program, a graph exists as data structure which contains a list of vertices and a list of edges. The vertex, as can be seen in Listing 5.3 which is a data type defined separately consists of a name (which is an integer), a color, a 2D position (it is actually implemented using a 3D

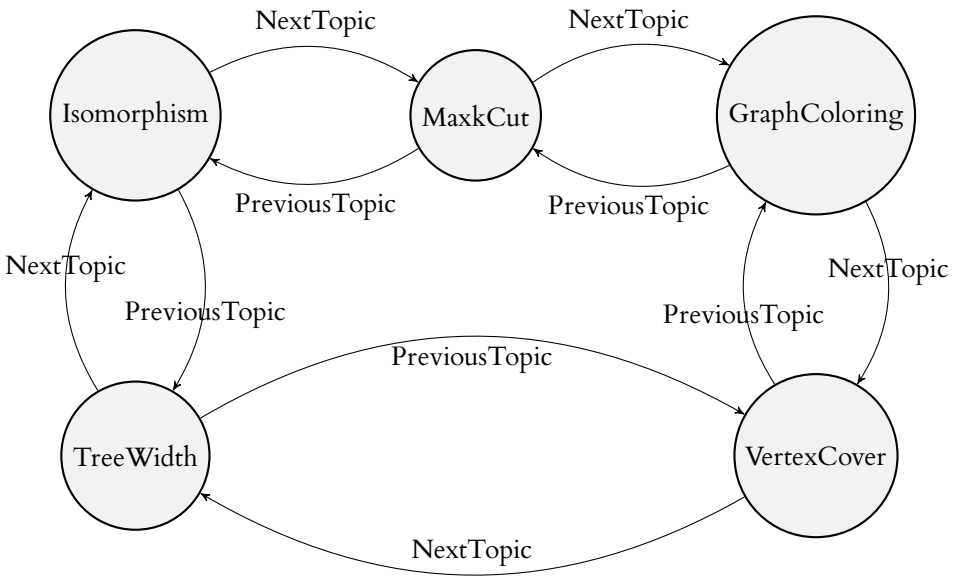

Figure 5.1: *This Finite State Machine (FSM) shows how the messages, NextTopic and PreviousTopic changes the state of the program (the Model) from one topic to another. This is a subset of a much larger FSM in which the number of states and inputs are greater in number.*

vector, with z is always kept at zero). An edge on the other hand is defined as a combination of two vertices. Such Graphs are present in the Model in and are used by the view function to be drawn as SVG.

```
1
2  -- Definition Vertex
3  type alias Vertex =
4      { name : Int, pos : Vec3, color : Color, glow : Bool }
5
6  -- Definition Edge
7  type alias Edge =
8      { vertexOne : Vertex, vertexTwo : Vertex }
9
10 -- Definition Graph
11 type alias Graph =
12     { vertices : List Vertex, edges : List Edge }
```

Listing 5.3: *Definition of Vertex, Edge and Graph. Vertex is defined as a data structure which has a name as an integer, a position as a 3 dimensional vector, color, and a flag which depicts if it has been selected by the user. An Edge has two Vertices and a Graph contains a list of Vertices and a list of Edges.*

5.3.1 Grid

In the program a Grid is a list of 3D vectors or in other words a list of position for vertices, which can be taken as an input by certain functions to construct graphs or change shapes of graphs.

A list of Vertices, for example can be formed by combining together lists of names, colors and a grid.

Grids are important in implementing animations. There is a function, for example, which which take two grids and output a grid which is geometrically in between the two grids.

5.3.2 Using Linear algebra to Initialize Grids

Linear algebra, in particular manipulation of vectors using Matrices has been used to create interesting grids for the placement of vertices in the scene. This includes rotation, scaling and translation of vectors to from polygonal patterns. Functions were created to form polygon with *n* geometric vertices which prove very handy in producing grids for various geometries like the one seen in Graph Ismoporphism and Max k Cut examples.

As a small example, here is a functional programming code in Listing 5.4 to find the centroid of of three position vectors. You can observe how first two vectors are added on line 3, and then it is pipelined to addition with a third vector, which is in turn pipelined to being scaled by 0.33 (divided by 3.0). This could have been achieved in a single line of code, but Elm reserves operators like +, −, ∗ for only numbers and they can't be overloaded to work for vectors.

```
1 findCentroidOfVertices : Vertex -> Vertex -> Vertex -> Vec3
2 findCentroidOfVertices v1 v2 v3 =
3    Math.Vector3.add v1.pos v2.pos
4    |> Math.Vector3.add v3.pos
5    |> Math.Vector3.scale 0.333
```

Listing 5.4: *Finding Centroid of Three Vertices. It logically starts with vertex the addition of positions of vertices v1 and v2. The output of which goes to the same function at line 4 albeit partially applied to position of v3. The output, which is the summation of the position of the three vertices in turn is passed to a scale function which effectively divides it's input vector by three. Hence the centroid emerges out of the other end.*

5.3.3 Creating Polygons

Two dimensional computer graphics was used to create polygons to form the Grids or subset of Grids for Graphs. Functional programming techniques such as mapping over a list was employed to create such grids. The construction of a polygon proceeds like this:

1. A list of floats containing ones is created.
2. Each is converted to an angle in radians corresponding to the polar position of the vertex.
3. This list of angles is then presented to a function which maps a horizontal vector with unit length over this list such that the unit vector is rotated with an angle equal to the angle in the list giving rise to a list of rotated vectors.
4. This regular polygon with vertices at unit distance from it's center is then scaled. The scaling factor for x is sometimes different from that of y. Which enables squeezing of the polygon along any 2 dimensional direction.
5. Mixing such polygons come handy to create more complicated shapes.
6. The polygon is then translated to an appropriate position on the SVG screen.

5.3.4 Implementing Colors

To have a list of neighboring colors acting as a color palette we work on the Hue Saturation Lightness color space (HSl), mostly varying the hue just pass a region in the spectrum of hues (First, Second or Third) and the number of colors needed as an Integer. On a scale from 0.00 to

1.00, the first region will produce hues ranging from 0.00 to 0.33, the second producing it from 0.33 to 0.66 and the third producing it between 0.66 to 1.00.

5.3.5 Edges

Edges are defined as a combination of two vertices. Since they are drawn as a straight line segment between the positions of the two vertices, they do not require positional data associated explicitly for them. It is drawn out from the vertices, they contain.

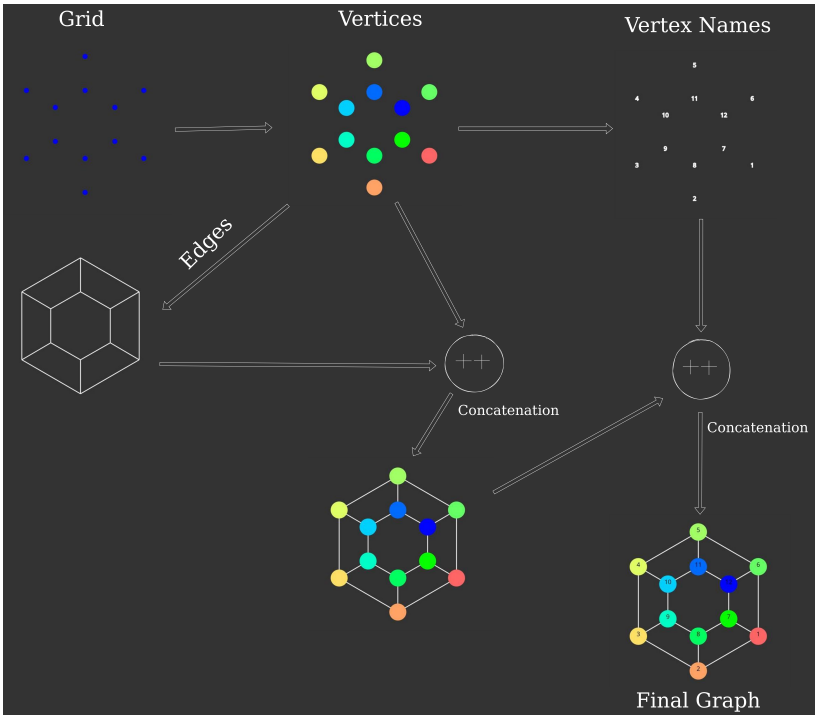

Figure 5.2: *Drawing a Graph. Starting with a grid of positions a set of vertices are created. Connecting the vertices, the edges are formed according to a list of tuples. The vertices is then laid on top of the edges by concatenating list of SVG elements representing edges with list of SVG elements of Vertices. This is followed by this new list being concatenated with a list of SVG text representing the vertex names.*

5.4 Implementation of Animations

In this section, it will be explained how various animations in the application are implemented. Though there are minor differences between animations for one topic to another, they follow a common pattern. The common pattern is this that events are generated by a quasi-regular clock. These events trigger the update function which transforms the current state of the program and changes the position of certain abstract entities. The view function while redrawing these entities takes the position information from the updated model to draw them as SVG.

5.4.1 Morphing Geometry of a Graph

In some of the animations in the application, the graph changes it's geometry to visually look different than the original. This is accomplished by a function which takes a graph and a grid to move the input graph incrementally towards the grid with every tick of the animation clock.

When the animation is started, with each tick of the animation clock, the vertices of the second graph move towards the target grid points with a constant velocity. The velocity of a vertex is calculated by obtaining the displacement vector between the target position and the current position of the vertex and also the time available before the animation ends. The displacement is calculated as $\vec{d} = \vec{p}_t - \vec{p}_v$, where $\vec{p}_t$ is the target postion and $\vec{p}_v$ is the current position of the vertex. The time available is calculated as $t = t_{available} - t_{elapsed}$. The velocity is calculated as $\vec{v} = \vec{d}/t$. The distance the vertex must travel is hence calculated as $\Delta d = \vec{v} \times \Delta t$. Where Δt is the time elapsed since the last tick of the animation clock.

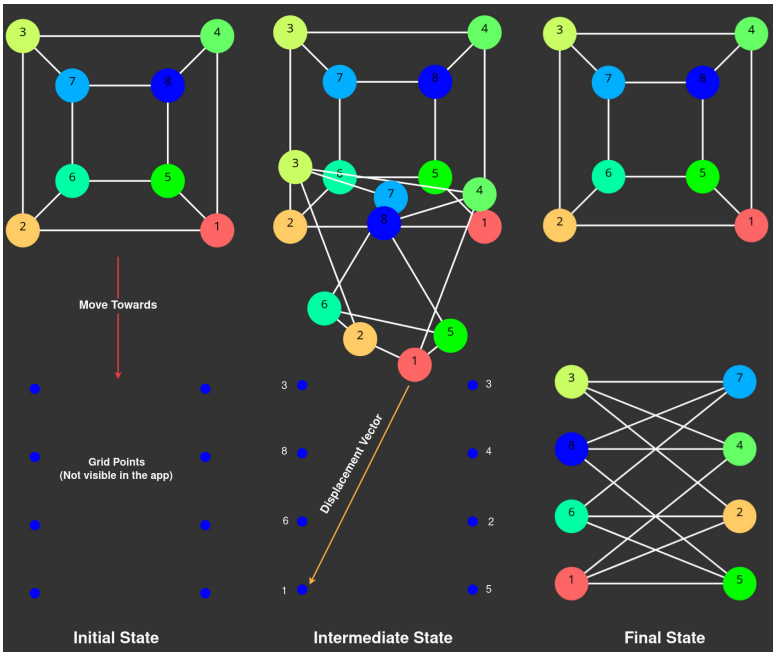

Figure 5.3: *Morphing a graph. This example from the Graph Isomorphism topic shows how shape transformation is done with the help of a targed grid. The target grid is shown as blue points. In the picture in the middle, a diplacement vector is shown between the current position of vertex 1 and it's final position. Veloicty of this vertex is calculated by deviding the displacement vector with the available time left in the animation.*

5.4.2 Re-formation of the Graphs

At each tick of the animation clock the graph under transformation, is built again, with vertices having the same name and color as the original but new positions. The edges need to be re-constructed again as the vertex positions have been renewed. This is something which is expected in the functional programming paradigm where nothing is changed in place and new data structures are created with application of a function. This is true not just for animations, it is true for user-interaction or anything which requires visual (Geometric or Color) modification of the graph.

The re-formation of the vertices and the edges are quite explicitly shown in the Elm function shown in Listing 5.5. The function takes a graph and a grid and produces a new graph situated at the new grid with new vertices and edges. The edges formed in the new graph are connected to the same vertices (vertices with the same names, actually) as the original ones.

```
1  morphGraph : Graph -> Grid -> Graph
```

```
morphGraph graph grid =
    let
        updatedVertices =
            List.map2 updatePositionVertex graph.vertices grid

        createEdge =
            updateEdge updatedVertices

        updatedEdges =
            List.map createEdge graph.edges
    in
    Graph updatedVertices updatedEdges
```

Listing 5.5: *Changing the shape of a graph for animation. The first line, in the code describes the Type Signature of the function morphGraph. It says that the function takes a Graph and a Grid as input and produces a Graph as an output. The input Graph's vertices adopt the positions listed in the Grid.*

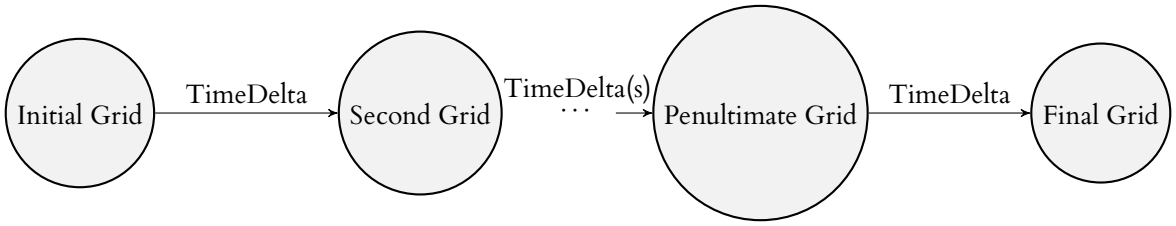

Figure 5.4: *An FSM shows, how the animation progresses with each tick of the elm run-time clock, depicted as the message TimeDelta. With each such message the position of each vertex, takes their respective position in the following grid.*

5.4.3 Drawing of Graphs

Drawing graphs is done using SVG elements. The vertices are drawn as color filled circles while the edges are drawn as straight line segments between the positions of the related vertices. The edges are drawn first and the vertices later so that vertices appear on top of the edges and the edges seem to be appearing out of the surface of the vertices.

5.5 Explanation Panel

The Explanation Panel is the right half of each page concerned with a topic. It consist of the title of the topic and suggestions on how to interpret and interact with the animations and user interactions.

The content of this panel depends on the state of the animation or the user interaction associated with the topic.

The functions which populate the explanation panel with different elements have therefore a subset of the *Model* data-structure as it's input. According to the state of the program appropriate advises, instructions and buttons are spawned on the panel.

The functions responsible for Explanation Panel in the case of user interaction such as the ones in Graph Coloring and minimum vertex cover run a check on the state of the program to know if the user is doing his task correctly. For example in graph coloring, if two adjacent vertices are colored the same color, there appears a message in the explanation panel warning about the same.

5.6 Navigation and Control

There are several buttons and keyboard shortcuts to go from one topic to the next and to play/pause and restart animations. These have been implemented by generating appropriate messages which are caught by the update function which in turn updates the model (state of the program). The updated model of the program is reflected on the screen by the view function.

5.6.1 Home Page

For the layout of the Home page, see it's design in ??. The icons contain miniaturized version of the graphs for the corresponding topics. The miniaturized versions of the graph are borrowed from the implementation of the various animations and user interactive tasks in the app. The miniatization is implemented by giving the SVGs small spaces to occupy on the page. As the graphics is scalable, it adopts the size of the parent. Clicking on the Graph Isomorphism icon, for example triggers the *GoToIsomorphism* message which is used by the *update* function to change the state of the model to contain the *Isomorphim* topic.

5.6.2 URL Management

Although there is only one HTML page rendered (this application being a Single Page Application), when the user navigates from one topic to another he can see the URL path after the Website's name change to reflect the pseudo-page he is on. This change in the URL is brought by the function *pushUrl* at appropriate situations. The change in the URL is detected by the run-time to produce the message *UrlChanged* which in turn is caught by the Update function to re-populate the page with new topic.

5.6.3 Navigation Bar

The navigation bar consists of buttons to go to the previous and the next topics. When the message *PreviousTopic* or the message *NextTopic* is generated by the buttons respectively, the update function catches it to change the state of the program (a data structure known as Model) to load it with the details of the previous/next topic.

5.6.4 Keyboard Shortcuts

Keyboard shortcuts provide a fast way to test various functionalities while developing the application. These functionalities have not been taken away even after development therefore they still can be used to trigger events in the application. The key presses messages are registered by the elm run-time which in turn triggers the update function to update the model. The updated model is now rendered by the view function. Information about the keyboard shortcuts can be availed on the screen by pressing the information icon present in the navigation bar. Below is an example list of a few key-bindings.

- **p**: Toggle between pause and play animation. (Can be used instead of the Play/Pause button; Generates the *AnimationToggle* message).

- **r**: Restart animation. (Can be used instead of Restart Button; Generates the *AnimationStartOver* message).

- **n**: Go to the next topic. (Can be used instead of the navigation button; Generates the *NextTopic* message).

- **N**: Go to the previous topic. (Can be used instead of the navigation button; Generates the *PreviousTopic* message).

- **t**: Next animation (Can be used in case of Max k Cut and Tree width to go to the next animation; Generates the *NextAnimation* message)

5.6.5 Screen Compatability

So that the various visual sections of the application occupy their appropriate place on screens of different sizes, such as screens of of various laptops and smart phones, the width and height of the screen is used to give the height and width to each section. The state of the program stores data of the type *DeviceType*, which makes the program aware of the screen size of the device being used. This data is used to give proper font size to various text elements used in the app. Such measures, have made the application friendly to mobile devices and changes in the browser window size caused by the user on a personal pc.

5.7 Implementation of Topics

The above sections have discussed the basic building blocks which can be used to implement the individual topics. The building blocks include graphs, animations page-layout, navigation and control.

5.7.1 Graph Isomorphism

This topic is elucidated by two user interactive animations. Refer to subsection 4.5.1 for the details.

Isomorphic Definition Animation The first animation consists of two Graphs which are the same. In the beginning the two graphs are superimposed with each other positionally. Each graph is constructed by merging two polygons and connecting their vertices in a wheel like structure. As the animation begins one graph leaves it's position and transforms to another shape. This is achieved by instantiating the two graphs with same position and form. An empty grid is also provided as the target set of positions for the vertices of the second graph.

When the user presses the play button the message *AnimationToggle* is generated. When the update function recieves this message, it cheks for the state of the program which is set at *IsomorphicTransition*. It starts the animation if it is not already in progress.

When the animation is started, with each tick of the animation clock, the vertices of the second graph move towards the target grid points with a constant velocity. For the kinematics involved in this transition see *Morphing Geometry of a Graph* in subsection 5.4.1.

The animation is also user interactive, when the user hovers over a vertex, or selects it by pressing the corresponding numerical key on the keyboard, a Boolean variable associated with the vertex is turned *True*. A function filters the list of vertices to find the selected vertex. Another function finds the edges incident on the vertex and also the vertices adjacent to the selected vertex by filtering over the list of edges in the graph. These are shown differently in a highlighted color in both the graphs in the display section. The explanation section also uses this data to explain how the selected vertex is connected to the same adjacent vertices in both the graphs.

Animated Quiz for Graph Isomorphism The second activity gives the user three graphs. The first graph is a reference graph. The user is asked to choose which one out of the two remaining graphs is isomorphic to the reference graph. The right answer is hard coded, and when the user selects his answer, the program checks if the answer chosen is right one and changes the appeartance of the display and the explanation panel. The state of the game consists

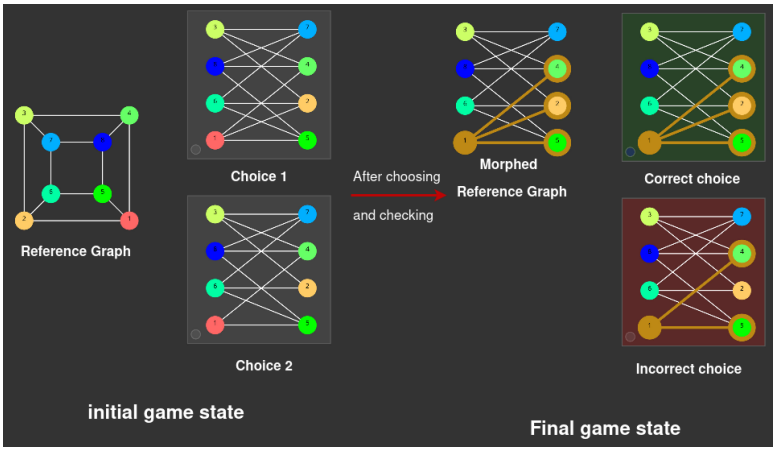

Figure 5.5: *Graph Isomorphism Quiz.*

of two variables: *Choice* and *CheckState*. When the user has not made any choice yet *Choice* is at *NoChoice* state. When the choice is made, *Choice* is set to *FirstGraph* or the *SecondGraph*. When the user presses the button to check the answer *CheckState* transforms from *NoCheck* to the *Check* state. There are case statements in the code which take the correct measure depending on the state of the game.

5.7.2 Max k Cut

The Max k Cut topic has two animations back to back related to firstly the Max 2 Cut and then Max 3 Cut . See subsection 4.5.2 for explanation of the animations. The first animation has a graph constructed by merging two polygons. The edges are so defined that the graph is nearly bipartite (see subsection 2.2.1) save for one edge (see subsubsection 4.5.2). The target grid for the animation is made of the combination of the same polygons but set apart vertically. The kinematics of the animation is executed the same way as is done for Graph Isomorphism.

The user can draw a predetermined line segment which seperates the two sets. The line segment is superimposed by intersection points of the line segments and the edges between the two sets. An intersection point is calculated by finding intersection of two line segments. A linear algebra library function was used for this task.

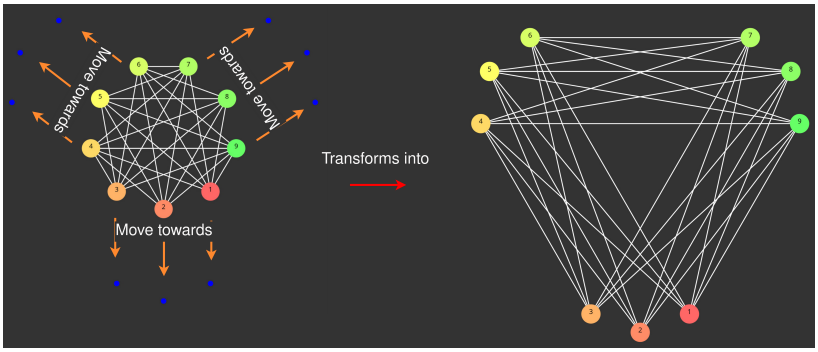

Figure 5.6: *Arrangement of animation of Max 3 Cut. This animation transforms a graph into a shape such that the three sets of vertices of the graph making a Max 3 cut are seperated from each other distinctively. The three sets move away from each other in directions which are* 120° *apart.*

In the case of Max 3 Cut (see subsubsection 4.5.2), the animation begins with a tripartite graph. The graph is based upon a nonagon shaped grid. The destination grid, in this animation is shape of an equilateral triangle, with the grid points near the vertices of the said triangle. The graph consists of three sets of vertices which settle down at the vicinity of the three vertices of the equilateral triangle at the end of the animation. The kinematics of the motion of the vertices is explained in subsection 5.4.1.

5.7.3 Graph Coloring

This section has two user interactive tasks to color graphs in the way they should be colored for a graph coloring problem. The graph in the first task is composed of two concentric square graphs such that each vertex has three adjacent vertices. The user task is to color the graph using two different colors. See subsection 4.5.3 for further explanation of the task.

The graph in the second task is composed of two concentric pentagonal graphs such that each vertex has three adjacent vertices. The user task is to color the graph using three different colors. See subsection 4.5.3 for further explanation of the task.

When the user chooses a color from the color palette, the chosen color is updated in the state of the program. When the user clicks a vertex, the color of the vertex is changed with the color stored in the state of the program.

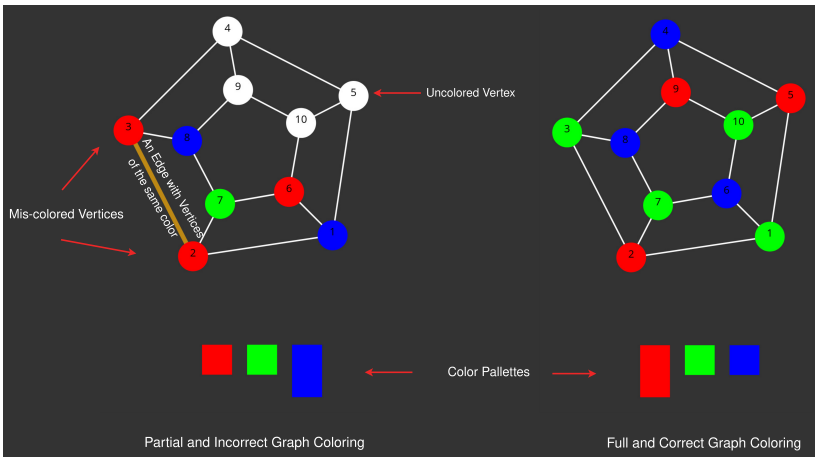

Figure 5.7: *Graph Coloring Task. The graph on the left has a pair of incorrectly colored vertices. On the right there is a fully and correctly colored graph, with all the adjacent vertices are colored differently from each other.*

There is a function which makes a check while the user task is going on, whether two adjacent vertices have similar color by doing a linear search on the edges. If they are the information is used in the display to mark that edge differently and a warning appears on the explanation panel based on this search.

If in a linear search for finding adjacent vertices of the same colors doesn't have an output and if all the vertices have been colored, then the function which populates text on the explanation panel shows a message that the task is complete.

5.7.4 Vertex Cover

The user interaction for this topic asks the user to choose the minimum number of vertices such that all the edges in the Graph belong to the set of vertices which are incident on the chosen vertices. See subsection 4.5.4.

The user is given two tasks similar in nature. In the first task the graph in the Vertex Cover is based on composition of two square grids. In the second task the graph is based on composition of two hexagonal grids. When the user clicks on a Vertex, there is a Boolean value associated with the Vertex which becomes *True*. There is a function which filters the list of edges to find which of them have at least one of their vertices *selected*. Such edges are illuminated, for the user to note that the edge has been covered.

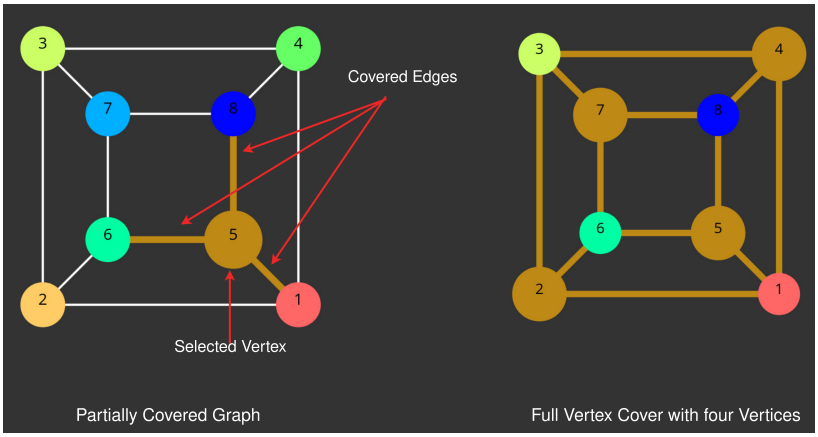

Figure 5.8: *Vertex Cover task. The graph on the left shows the selected vertex 5 and the edges incident on it in golden color. A vertex can be selected by either clicking the vertex or pressing the corresponding number key on the keyboard. The edges shown in golden color are the ones which are covered, they are colored so automatically when a vertex related to them is selected. The graph on the right has all it's edges covered.*

There is another function which filters out the edges whose one of the vertices have been selected, and yet another which counts the number of vertices which have been selected. When the all the edges have been covered, the program checks for the number of vertices selected to do this by filtering the list of vertices. In the case of the particular example in this project, the graph can be covered using four vertices. Therefore if the number of vertices selected at the end of the session are greater than four, then the explanation panel is populated with a message that the task needs to be attempted again for proper execution.

Next-task button loads the program with the new task by changing the state of the program.

5.7.5 Tree Width

Tree width topic is implemented in a series of animations. In the first animation the vertices of the graph of this topic are arranged circularly albeit with edges such that there is a cellular structure in present inherently in the graph. This edges were set up in this graph by coding their connectivity in a list of tuples. This is also a graph-transition just like graph isomorphism. The target grid for this transition is a regular lattice which reveals the cellular nature of the graph. The kinematics of the transition is the same as that of graph isomorphism (see subsection 5.7.1) and Max k Cut (see subsection 5.7.2). For more information on the kinematics of such shape transitions see subsection 5.4.1.

The second part of the animation highlights one piece of the cellular structure (see **??**). The piece is also marked by drawing a blue dot at the center of the it. The position of the blue dot is found by taking the centroid of the constituent vertices of the piece.

In the third part the centroid of all the pieces are found and marked as blue. These blue dots are connected by special lines in the last part of the animation to form the tree structure inherent in the original graph.

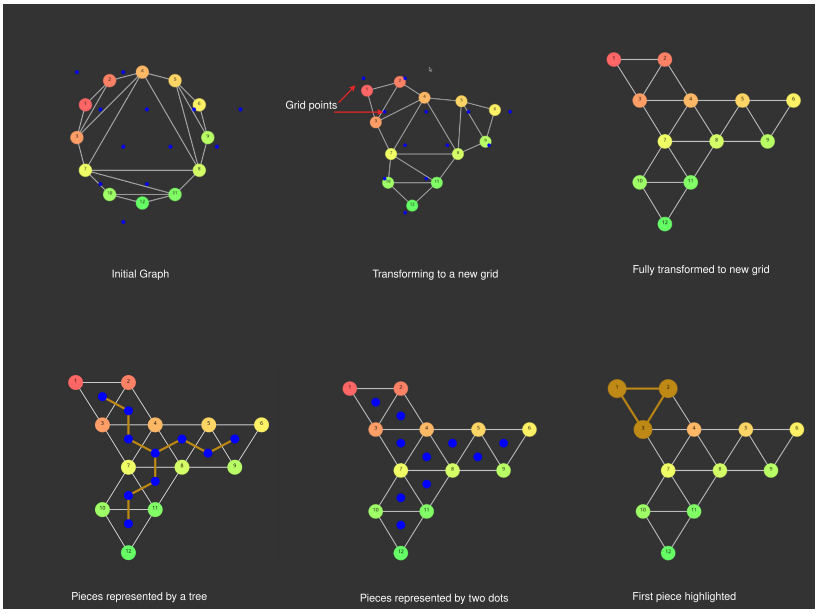

Figure 5.9: *Tree width Animations. From the top left to bottom left. Tree width is explained in a series of animations. The first animation turns the original graph to a cellular like structure by moving towards a pre-determined grid. After the transformation a piece is defined as a subset of a graph. The next graph shows how graph can be decomposed into several such pieces. The last picture shows how the pieces can be connected together by a tree.*

The animation series is followed step by step by explanation panel which responds to the various states of the program.

5.8 Modules

The program is divided into ten modules.

The *Main* module is of the central utility. It's function is to be a starting point for the program and also to act as a porter to load the functionality of individual graph theory topics to the state of the program whenever the navigation commands demand so.

The five topics included in the project own a module of their own. They are *Isomorphism*, *Maxkcut*, *GraphColoring*, *VertexCover* and *TreeWidth* modules. These modules are imported by the main function to be used on the screen. Henceforth these will be collectively known as the *Topic Modules*.

The *Graph* module contains the data types and functions for representing, constructing, drawing and animating the graphs. This module contains the most geometry. This module is depended upon by all the *topic modules*.

The *Explanation* module contains the text data as a $0-$ *nary* functions (functions which don't take input) used by topic modules for the explanation of the respective topics. These only contain the definitions of the concerned graph theory problems.

The *Messages* module contains the messages which are generated with system clock ticks or user interaction with the DOM elements. Such messages are defined as *Algebraical Data Types* (see section 5.2) and are imported by almost all other modules. This along with the *Main* and *Graph* modules form the central pillar of the program and are the minimum requirement of the

program to function.

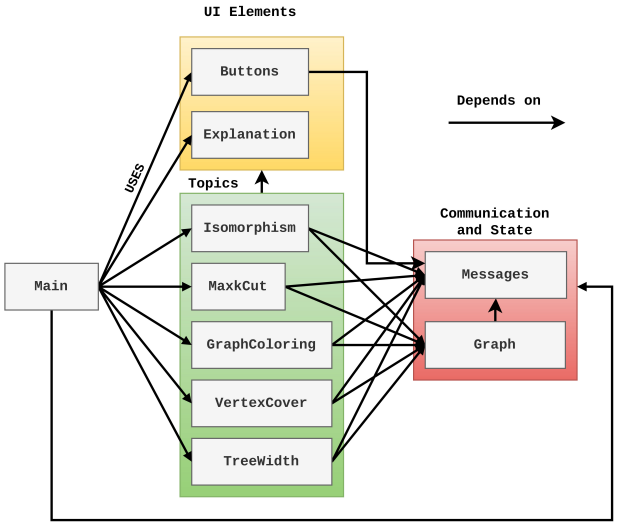

Figure 5.10: *Modules in the Program. Arrows from the blunt end to the pointy end depict dependency. The main module contains the main application which uses different modules. The most noteworthy of them are the modules related to the different graph theory topics. The graph theory topics uses topics like graph and messages. While graph is used to define and draw graphs, messages is important for keeping track of the events.*

5.9 Software Engineering Practices

In the following subsections, how a few good practices which were adopted to make the project of this size manageable and organized are explained.

5.9.1 Version Control

The program along with the documentation and the dissertation report were version controlled in a single GitHub repository. A git repository, gives the developer confidence for fearless refactoring and feature enhancement by the way of creating separate branches for separate issues. Therefore a different branch was created for editing documentation, and it was made sure that modification in code files was not done on this branch. Similarly, a branch for code refactoring didn't modify the documentation part hence separating concerns. Furthermore a branch was conveniently discarded if an adventure in a radical change in code went wrong.

5.9.2 Continuous Deployment

The web application concerning the project is deployed on the World Wide Web using Netlify. It is a service which lets one host a front end website. It is connected to the master branch of the GitHub repository of the project. Every time a new version is pushed to the master, netlify service fetches the new version of the app from the repository, builds it according to the build script present in the GitHub repository and deploys the web application on a specific URL. The build and deployment can also be done manually by going to the dashboard of the Netlify website. The build and deployment process primarily consists of running a script to transpile Elm code to a JavaScript file and copy the output along with the boiler plate HTML to the publish directory.

If it is needed to deploy the application anew instantaneously (something which is required to test the website for different screen sizes) a *curl* command with a specific URL called a *build hook* is executed on the local machine. It deployes the website from the code in the master branch in the repository.

5.9.3 Documentation

Documentation for the project was done in a continuous fashion in a variety of ways, such as a wiki for code implementation and a guide for future work, time logging for project management and report writing for the submission.

The wiki was informed heavily from the code comments made to explain the functions. Time logging brought a sense of discipline in the whole endeavor. It also helped in estimating time required to complete the tasks which lay in future and asses the learning curve.

6 Evaluation

7 Conclusion

7.1 Reflections

7.1.1 Experience of using Functional programming for Front End Programming

Here is a reflection on the experience of using functional programming for this project. This experience will also be compared to past experience in programming in imperative languages.

A web app today is one of the primary means of human computer interaction (HCI). To model something as complex as a HCI a programming paradigm which can model this complexity efficiently is needed.

An imperative programming language, however *high-level* it might be, is much closer to the Von-Nueman machine in terms of how a computer program is reasoned about by the programmer. There is a tape and it has to be read one step at a time. The programmer implements most of the logic by control mechanisms and less in logic itself. Therefore the program becomes far more complex than the actual complexity it has to represent.

Functional programming on the other hand was much more efficient in modelling the domain to be represented. It implemented logic more in terms of logic and less in terms of control structures. Such that while reading the program one saw mostly logic and not boiler-plate code.

Using Elm in representing the world inside the application and the signals which are received from the outside world was simple and elegant. The functional *type system*, not only provides run-time safety, it also gives a structure for organizing, extending and refactoring of the project which continued to grow in size and complexity. The application was made in a bottom up style of designing. Animations were developed before pages, and pages were done before the whole of the app.

While types made modelling the domain efficient. Functions made the rules of physics in the web application explicit. Pattern matching on the inputs of the functions was used extensively in the code to take logical decisions. Another important method was pipeling the output of a function to the input of another function. Reading a series of pipelined functions, was easy to reason with as it seems like data is being processed on an assembly line with functions acting on it in turn.

A Appendices

7 Bibliography

V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(S7), Apr. 2013. doi: 10.1186/1471-2105-14-s7-s13. URL https://doi.org/10.1186/1471-2105-14-s7-s13.

J. Fairbank. *Programming Elm: Build Safe and Maintainable Front-End Applications*. Pragmatic Bookshelf. 2019. ISBN 1680502859. URL http://gen.lib.rus.ec/book/index.php?md5=597c62b902710b0f282678b951ce8638.

R. Feldman. *Elm in Action*. Manning Publications, 2020. ISBN 9781617294044. URL https://books.google.co.uk/books?id=KNt8vQEACAAJ.

D. Galles. Data structure visualizations. https://www.cs.usfca.edu/galles/visualization.

C. Gros. *Complex and adaptive dynamical systems*. Springer International Publishing, Cham, Switzerland, 4 edition, Apr. 2015.

S. Halim. Visualgo. https://visualgo.net/en.

P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of haskell. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, June 2007. doi: 10.1145/1238844.1238856. URL https://doi.org/10.1145/1238844.1238856.

J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 01 1989. ISSN 0010-4620. doi: 10.1093/comjnl/32.2.98. URL https://doi.org/10.1093/comjnl/32.2.98.

K. Karvonen. The beauty of simplicity. In *Proceedings on the 2000 Conference on Universal Usability*, CUU '00, page 85–90, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581133146. doi: 10.1145/355460.355478. URL https://doi.org/10.1145/355460.355478.

J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2006.

L. Lau. Algmatch. https://liamlau.github.io/individual-project/.

M. Newman. *Networks: An Introduction*. Oxford University Press, 03 2010. ISBN 9780199206650. doi: 10.1093/acprof:oso/9780199206650.001.0001. URL https://doi.org/10.1093/acprof:oso/9780199206650.001.0001.