Lancaster University Leipzig School of Computing and Communications

Facilitating Graph Exploration

Youssef Gerges Ramzy Mokhtar

Date of Submission (22/03/2024)

Supervisor: Dr Marco Caminati

B.Sc. (Hons) Computer Science

Number of words = 9045
This includes the body of the report only

Declaration of Originality

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work.
I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.
Name: Youssef Gerges Ramzy Mokhtar
Date: 19/03/2024

Abstract

The primary focus of the project is to create aesthetically appealing graph drawings and provide various graph algorithm visualizations. The project utilizes the spring embedder algorithm to visualize graphs. This algorithm simulates physical forces to draw visually appealing graphs. The project is implemented as a website, employing HyperText Markup Language (HTML) and Cascading Style Sheet (CSS) to create the user interface. JavaScript serves as the programming language for implementing the project's features and logic. This project stands out for its unique combination of dynamic physical simulation for graph drawings, alongside animated algorithm visualizers that run concurrently with the graph drawing process.

Acknowledgements

I would like to express my sincere appreciation to Dr. Marco for their great guidance and support throughout this project

Contents

1 Introduction **1**
- 1.1 Project Aims . . . 1
- 1.2 Motivation . . . 1
- 1.3 Expected Knowledge . . . 2
- 1.4 Report Overview . . . 2
- 1.5 Project Links . . . 3

2 Background **4**
- 2.1 Related Work & Project Theme . . . 4
- 2.2 Technology Used . . . 4

3 Graph Aesthetics: Mathematics & Physics Foundation **6**
- 3.1 Notations Used . . . 6
- 3.2 Physical Simulation . . . 6
- 3.3 Force Resolution . . . 8
- 3.4 Physical Simulation Formulation . . . 8
- 3.5 Spring Embedder Algorithm . . . 10

4 Additional Maths Used During Graph Drawing **12**
- 4.1 Connecting Two Circles . . . 12
- 4.2 Edge Weight Following Line Position And Angle . . . 14

5 Code Architecture & Design **16**
- 5.1 Code Architecture . . . 16
- 5.2 Code Design . . . 18
- 5.3 Design Challenges . . . 21

6 Implementation **24**
- 6.1 Website Usage . . . 24
- 6.2 Graph Data Structure . . . 26
- 6.3 Concurrency in JavaScript . . . 28
- 6.4 Concurrency Challenge in JavaScript . . . 29
- 6.5 Concurrency Solution in JavaScript . . . 31

7 Conclusion **34**
- 7.1 Review Of Aims . . . 34
- 7.2 Future Work . . . 34
- 7.3 Self Reflection . . . 35

A Original Project Proposal **39**

B Another Appendix Chapter **45**

List of Figures

3.1 Hooke's Law 7
3.2 Coulomb's law 7
3.3 Formula computing the force on a vertex 8
3.4 force resolution in 2D space 8
3.5 Formulating Hooke's Law & Coulomb's Law 9
3.6 applying force-resolution on the force formula for a vertex 9
3.7 final formulation for the x-axis force component $f_x(v)$ on a vertex 10
3.8 Spring Embedder Pseudocode 11

4.1 Problem 1 Illustration 12
4.2 Problem 1 Using Paythogres 13
4.3 4 Circle Quadrants 14
4.4 Finding angle of the line 15
4.5 Rectangle On Line 15

5.1 Code Architecture 16
5.2 UML Class Diagram 19

6.1 Website Overview 25
6.2 DFS Algorithm Visualizer 26
6.3 Nodes As Text 26

B.1 Initial UML Class Diagram 45
B.2 UI Wireframe 46

Listings

6.1 Graph Class 27
6.2 JavaScript code for drawing a graph 29
6.3 JavaScript code for algorithm visualization 30
6.4 Simple JavaScript asynchronous function 31
6.5 Initial logic to know when should a function be stopped 32
6.6 Code logic to know when should a function be stopped 32

1 Introduction

In discrete mathematics, a graph is a discrete structure consisting of a set of objects known as vertices or nodes. These vertices have relationships with one another, and these relationships are referred to as edges, forming a set of connections between the objects. [6]

Graphs have a long history dating back to the 18th century, and they have been central in computer science and discrete mathematics. They have been used in many areas like traffic routing, social media platforms, operating systems, computer networking, and much more.

1.1 Project Aims

The overall aim of the project is to create a website that displays aesthetically pleasing-looking graphs and offers multiple graph algorithm visualizers. The main goal is to provide the user with the following functionalities:

- Allowing users to input the graph using edge list representation and displaying it visually.
- Offering options to view both directed and undirected versions of the graph.
- Providing animated visualizations for different graph algorithms, such as:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
  - Dijkstra's Shortest Path
- Giving users the ability to adjust the speed of algorithm visualization.
- Displaying a trace of the execution steps for the above-mentioned algorithm.

1.2 Motivation

The main reason I decided to work on this project was curiosity. Before starting this project, I often used websites that showed interesting graph simulations and algorithm visualizations. These websites made me wonder how I could build something similar. This curiosity became the main reason for working on this project - to learn how to implement graph simulations and algorithm visualizations. I also wanted to create a useful tool for competitive programmers, for my use, and for educators who want to explain graphs and graph algorithms visually.

1.3 Expected Knowledge

This report assumes a basic understanding of certain topics. Next to each topic name, you will find link(s) for further exploration if the reader is unfamiliar with any of those topics.

- HTML — (link)
- CSS — (link)
- JavaScript — (link1 | link2)
- Document Object Model (DOM) API — (link)
- Scalable Vector Graphics (SVG), mainly their usage with HTML — (link)
- Graph Representations, specifically Adjacency List, Adjacency Matrix, and Edge List — (link1 | link2)
- Graph Algorithms, specifically DFS, BFS, and Dijkstra — (link1 | link2 | link3)
- Trigonometric functions, specifically sine, cosine, and tangent — (link)
- Pythagoras theorem — (link)
- Object-oriented Programming — (link)
- UML Class Diagrams — (link)
- Few knowledge of some design patterns, specifically the template method design pattern — (link)

Final remarks, the terms "nodes" and "vertices" will be used interchangeably in this report. Additionally, the phrase "classical graph data structure" will be used frequently, signifying that the graph nodes are simple integers and are numbered from 0 to $N - 1$.

1.4 Report Overview

Background This chapter focuses on discussing similar projects and highlighting the different features of this project compared to existing ones. Furthermore, we discuss the technology and languages used for the project implementation.

Graph Aesthetics: Mathematics & Physics Foundation Here, we take a deep dive into the mathematical and physical principles behind the graph drawing algorithm.

Additional Math used during simulation This chapter explores additional mathematical concepts that were used during implementation, such as connecting two circles from their surfaces.

Code Design In this chapter we will switch our focus to the overall design of the software, we will mainly discuss the code architecture, classes, and their interactions, and examine various design challenges along with their solutions.

Implementation In this chapter we will discuss how to run and use the website, and specific implementation details and challenges encountered throughout the development.

Conclusion In this chapter, I will conclude the report with a personal reflection on my performance in this project, my key learnings, areas for improvement, and potential future work to enhance the project's functionalities.

1.5 Project Links

To facilitate ease of access to the project, the following links are provided:

- GitHub Repository: https://github.com/youssef-gerges-ramzy-mokhtar/Facilitating-Graph-Exploration
  The project is hosted on GitHub, where it has been managed since the start.
- Live Website: https://facilitating-graph-exploration.web.app/
  The project is deployed on the internet for convenient access and ease of interaction.
- Complete Project Material: ***Facilitating-Graph-Exploration-Material.zip***
  All project materials are provided in a zip file for offline review.

2 Background

2.1 Related Work & Project Theme

These days, numerous tools are available for creating aesthetically pleasing graph drawings and visualizers for different algorithms. Examples of such tools are:

- Graph Editor - CS Academy [1] – The Graph Editor project, available on the CS Academy website, serves as a tool for visualizing graphs by using physical simulations to generate aesthetically pleasing graph drawings. The platform enables users to manually move graph nodes and offers features such as downloading the graph as a PNG image and generating markup for the resulting graph drawing.
- Graphonline [3] – An open-source project used for displaying graphs and visualization of various algorithms. One unique feature of this project is allowing users to add custom algorithms, providing the ability to visualize them directly on the graph.
- D3 Graph Theory [13] – A project aimed at providing interactive education on graph theory through various graph and algorithm visualizations.
- CS USFCA [16] – A website developed by the University of San Francisco to assist students in understanding data structures and algorithms. It offers a wide range of visualizations for various data structures and algorithms.
- VisuAlgo [4] – A website initially designed for National University of Singapore (NUS) students to visually learn about data structures and algorithms. It provides a wide range of visualizations for different data structures and algorithms, including several visualizers for advanced data structures and algorithms.

The project focuses on both displaying aesthetically pleasing graph drawings and providing different graph algorithm visualizers. The key distinction between this project and existing ones is the integration of dynamic physical simulation for graph drawing alongside support for running different algorithm visualizers on the graph as it is being drawn.

This project combines elements from both the Graph Editor - CS Academy, which utilizes physical simulation to generate graph drawings, and tools like CS USFCA or VisuAlgo, which offer various algorithm visualizers.

2.2 Technology Used

In this section, we will describe the technologies used for the project. The project is developed as a website using the following languages and technologies

1. HyperText Markup Language (HTML) – used to define the content and elements of the website.

2. Cascading Style Sheets (CSS) – used to style the visual attributes of the HTML content and elements.

3. Scalable Vector Graphics (SVG) – used to display the graph on the website.

4. JavaScript (JS) – programming language used to implement all the project logic.

Now you might be wondering why have I chosen to create a website from the start, and have chosen to use web technologies and languages. The are two main reasons for that choice.

The first main reason is that since the project is about graph drawing and graph algorithm visualization, a large portion of the project logic relies on the user interface. With the need to easily access, manipulate, and create new UI elements, the DOM API provides a straightforward way for interacting and modifying HTML elements from the frontend.

The second reason is that I was to some extent comfortable and familiar with how to code using JavaScript.

You might wonder why I choose SVG over other alternatives like Canvas, for example. The main reason is that SVG uses geometric shapes to render graphics, whereas Canvas uses pixels. For our use case, we only need to create simple geometric shapes such as circles and lines to represent the graph. Another advantage is that SVG drawings will never suffer from quality issues like pixelation.

3 Graph Aesthetics: Mathematics & Physics Foundation

There are several techniques to display an aesthetically good-looking graph. For the project, I have chosen force-directed algorithms, specifically the spring embedder algorithm. The reason for choosing force-directed algorithms is that they are generally easy to implement, and they produce very good results with small graphs.

The spring embedder algorithm simulates a graph as a physical entity where nodes act as equally charged particles that repel each other, and edges connecting two nodes act like springs that can be bent and stretched. In the next sections, we will go through how to simulate a graph as the physical entity just described above.

The majority of the mathematical formulas used for the spring-embedded algorithm were drawn from the book 'Graph Drawing: Algorithms for the Visualization of Graphs'[2]

3.1 Notations Used

Here we will provide a concise reference describing the notations used throughout this chapter:

$G = (V, E)$	graph G defined as set V of vertices, and set E of edges
$f(v)$	force exerted on vertex v
$d(u, v)$	euclidean distance between node u and v
$f(u, v)$	hooke's law force exerted by edge $(u, v) \in E$
$g(u, v)$	coulomb's law force exerted by the vertices $u \in V - v$
$P_v = (x_v, y_v)$	(x, y) coordinate of vertex v
f_x	force x-component
f_y	force y-component
$f_x(v)$	force x-component on vertex v
$f_y(v)$	force y-component on vertex v
K_1	spring constant between any edge $(u, v) \in E$
K_2	strength of electric repulsion between any 2 vertices $(u, v) \in V \times V$
L	natural length of the spring between any edge $(u, v) \in E$
adj	adjacent
opp	opposite
hypo	hypotenuse

3.2 Physical Simulation

Given a graph $G = (V, E)$ we want to model a node as a charged particle that repels other nodes, and edges as springs that can stretch and compress based on the spring length.

A spring has an ideal length at which it experiences no force. When the spring is stretched, it attempts to return to its original length. Similarly, when it is compressed, it tries to stretch back to its original length. The strength of the force during stretching or compression primarily depends on the spring's stiffness. We can use Hooke's law to calculate the force on a spring, as shown in Figure 3.1.

$$F = kx$$
$$x = l - b$$

where:

F is the force

k is the spring constant

x the extension of the spring from its ideal length

l spring length

b base length of spring

Figure 3.1: Hooke's Law

For the charged particles, we always assume that they all carry the same charge sign. In other words, all our nodes are treated as either positively charged particles or negatively charged particles. The rationale behind this is to ensure that nodes always repel each other. We can use Coulomb's law to calculate the force on a charged particle, as shown in Figure 3.2 below.

$$F = k\frac{q_1 q_2}{r^2}$$

where:

F is the force

k coulomb's constant

q_1 charge on particle 1

q_2 charge on particle 2

r distance of separation between the 2 particles

Figure 3.2: Coulomb's law

By assuming all nodes have the same charge in our system, where $q_1 = q_2$ we can simplify Coulomb's law to $F = \frac{kq^2}{r^2}$, where $q = q_1$.

Having formalized the laws governing our physical simulation, we can now derive an equation to compute the force acting on any vertex v within the graph. This equation, shown in Figure 3.3, is defined as the sum of all edge forces connected to vertex v, along with the sum of all repulsion forces between vertex v and all other vertices.

$$f(v) = \sum_{(u,v)\in E} f(u,v) \; + \sum_{u\in V-v} g(u,v)$$

Figure 3.3: Formula computing the force on a vertex

3.3 Force Resolution

In this section, we will discuss force resolution. A force can be represented as a vector value in space, with both a magnitude and direction. Force resolution involves splitting a force into two or more components such that the combined effect of these components remains identical to the original force.

As we are simulating our graphs in a 2D space we are mainly concerned with converting a force into two components $x - axis$ and $y - axis$ components. The main advantage of splitting the force into 2 components is that during implementation it will make it easier to know the force exerted along the $x - direction$ and the $y - direction$. Figure 3.4 shows an illustration of force resolution.

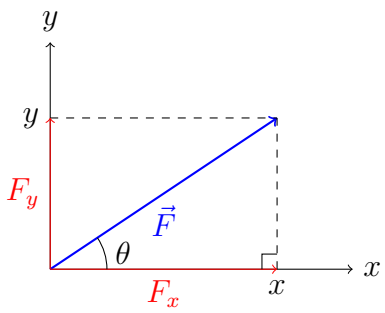


Figure 3.4: force resolution in 2D space

Based on Figure 3.4 we can use trigonometry rules to convert a force into its corresponding x-component and y-component.

$$hypo = F \quad adj = F_x \quad opp = F_y$$

$$\cos\theta = \frac{adj}{hypo} \quad \cos\theta = \frac{F_x}{F} \quad F_x = F\cos\theta$$

$$\sin\theta = \frac{opp}{hypo} \quad \sin\theta = \frac{F_y}{F} \quad F_y = F\sin\theta$$

3.4 Physical Simulation Formulation

In this section, we will try to formulate the force formula in Figure 3.3, so that it can be used for the spring embedder algorithm. In the next section discussing the spring embedder algorithm, it will be more clear why we have used force resolution for the formulation of the force formula on a vertex v.

Let's first start by formulating Hooke's law and Coulomb's law used in the force formula in Figure 3.3 for a vertex in 2D space. Figure 3.5 describes such formulation.

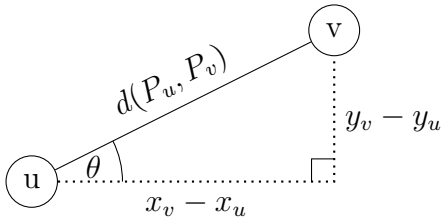
$$f(u,v) = K_1(d(P_u,P_v) - L) \qquad \text{(formulating Hooke's law)}$$

$$K_2 = kq^2 \qquad g(u,v) = \frac{K_2}{d(P_u,P_v)^2} \qquad \text{(formulating Coulomb's law)}$$

$$f(v) = \sum_{(u,v)\in E} K_1(d(P_u,P_v) - L) + \sum_{u\in V-v} \frac{K_2}{d(P_u,P_v)^2} \quad \text{(rewriting force formula in Figure 3.3)}$$

Figure 3.5: Formulating Hooke's Law & Coulomb's Law

Now we would like to decompose the force formula on Figure 3.3 on a vertex to two components the x-component and the y-component, in simple terms we will apply force-resolution on this formula. Figure 3.6 shows how the force components are computed.



$$hypo = d(P_u,P_v) \quad adj = x_v - x_u \quad opp = y_v - y_u$$

$$\cos\theta = \frac{adj}{hypo} \quad \cos\theta = \frac{x_v - x_u}{d(P_u,P_v)}$$

$$\sin\theta = \frac{opp}{hypo} \quad \sin\theta = \frac{y_v - y_u}{d(P_u,P_v)}$$

$$f_x(v) = f(v)\cos\theta \quad f_y(v) = f(v)\sin\theta$$

$$f_x(v) = f(v)\frac{x_v - x_u}{d(P_u,P_v)} \quad f_y(v) = f(v)\frac{y_v - y_u}{d(P_u,P_v)}$$

Figure 3.6: applying force-resolution on the force formula for a vertex

Finally after applying force-resolution on the force formula $f(v)$ in Figure 3.6, we can proceed to expand both force components $f_x(v)$ and $f_y(v)$. Figure 3.7 shows the expansion of force component $f_x(v)$ only, but the same steps can be applied to derive the force component $f_y(v)$

$$f_x(v) = f(v)\cos\theta$$

$$f_x(v) = \left(\sum_{(u,v)\in E} f(u,v) + \sum_{u\in V-v} g(u,v)\right)\cos\theta$$

$$f_x(v) = \left(\sum_{(u,v)\in E} f(u,v) + \sum_{u\in V-v} g(u,v)\right)\frac{x_v - x_u}{d(P_u,P_v)}$$

$$f_x(v) = \left(\sum_{(u,v)\in E} K_1(d(P_u,P_v) - L) + \sum_{u\in V-v} \frac{K_2}{d(P_u,P_v)^2}\right)\frac{x_v - x_u}{d(P_u,P_v)}$$

$$f_x(v) = \sum_{(u,v)\in E} K_1(d(P_u,P_v) - L)\cdot\frac{x_v - x_u}{d(P_u,P_v)} + \sum_{u\in V-v} \frac{K_2}{d(P_u,P_v)^2}\cdot\frac{x_v - x_u}{d(P_u,P_v)}$$

Figure 3.7: final formulation for the x-axis force component $f_x(v)$ on a vertex

3.5 Spring Embedder Algorithm

In this section, our focus will primarily be on explaining the procedure of the spring embedder algorithm. We will make use of the formulas established in previous sections to draw the graph in a nice-looking way which is the primary goal of the algorithm.

The main idea of the algorithm is to go through each node in the graph, calculate the force exerted on that node, move the node a small fraction in the direction of the force, and repeat this process for every node. The algorithm continues to iterate through all nodes, adjusting their positions, until the system achieves a state where the force exerted on all nodes falls below a predefined threshold. Optionally, a maximum iteration limit can be used to prevent indefinite execution if the desired low-force equilibrium is not achieved. It is worth noting that the node positions are initially assigned with random positions.

```
SPRING-EMBEDDER(G, Pos, rate)
1  while TRUE
2      for each vertex u ∈ G.V
3          Force = NODE-FORCE(u, G, Pos)
4          Pos[u].x = Pos[u].x + rate · Force.x
5          Pos[u].y = Pos[u].y + rate · Force.y

NODE-FORCE(u, G, Pos)
 1  Force.x = 0
 2  Force.y = 0
 3  for each edge (u, v) ∈ G.E
 4      f = K1 · (d(u, v) − L)
 5      FORCE-RESOLUTION(u, v, f, Force, Pos)
 6
 7  for each node v ∈ G.V
 8      if u ≠ v
 9          f = K1 / d(u,v)^2
10          FORCE-RESOLUTION(u, v, f, Force, Pos)
11
12  return Force

FORCE-RESOLUTION(u, v, f, Force, Pos)
1  Force.x = Force.x + f · ((Pos[v].x − Pos[u].x)/d(u, v))
2  Force.y = Force.y + f · ((Pos[v].y − Pos[u].y)/d(u, v))
```

Figure 3.8: Spring Embedder Pseudocode

Final remarks, it is important to note that the graph is not in the real physical world. Therefore, we can adjust the constants, such as the electric repulsion constant, used to calculate the force on a vertex, to achieve the most visually appealing graph drawings. Additionally, the algorithm may require an initial calibration phase to determine the best constants that yield good-looking graph drawings. Following the calibration phase during implementation, the following constants were established: $K_1 = 10, K_2 = 1500^2, L = 130$

4 Additional Maths Used During Graph Drawing

This chapter will be relatively short, and we will mainly discuss some additional mathematics that was used during implementation to achieve the following:

1. Connecting two circles by a line from their surfaces.
2. Having the edge weight follow the line position and angle.

4.1 Connecting Two Circles

In this section, we will illustrate the math used to connect two circles, representing two nodes in the graph, by a line from their surfaces. This task would be straightforward if the lines were connecting the two circles from their centers, as the circle is defined based on its (x, y) center coordinates. Without further ado, let's define the problem formally. We can formulate the problem as follows:

Problem 1 *Given 2 circles C_1 and C_2 both with radius r and both are centered at (x_1, y_1) and (x_2, y_2) respectively. We would like to find the point (x, y) such that it satisfies all the following conditions:*

- *points (x_1, y_1), (x_2, y_2), and (x, y) all are connected in a straight line.*
- *point (x, y) lies on the circumference of circle C_1.*

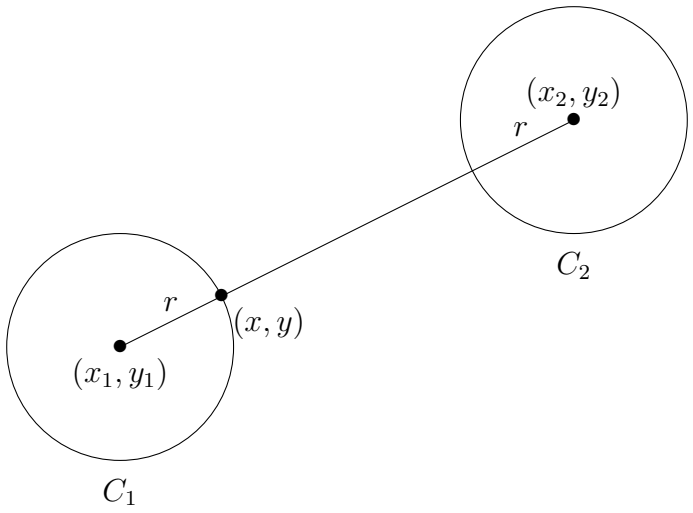
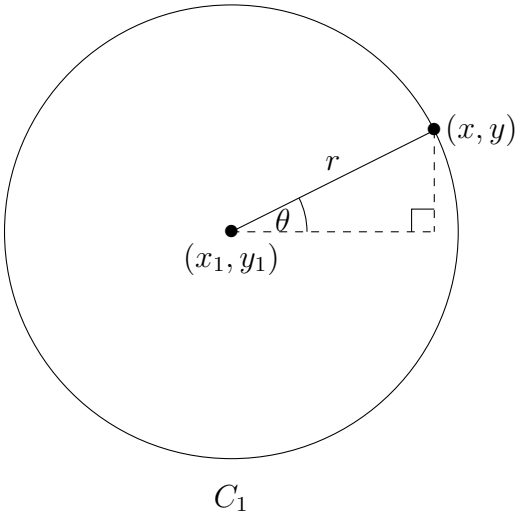


Figure 4.1: Problem 1 Illustration

Now let's focus on getting the x-distance and y-distance between points (x_1, y_1) and (x, y), and to do that we can simply use trigonometry rules as illustrated in Figure 4.2



$$hypo = r$$

$$\sin\theta = \frac{opp}{hypo} \qquad opp = |y - y_1| \qquad r\sin\theta = |y - y_1|$$

$$\cos\theta = \frac{adj}{hypo} \qquad adj = |x - x_1| \qquad r\cos\theta = |x - x_1|$$

Figure 4.2: Problem 1 Using Paythogres

Finally, you might be wondering how could we find the value of θ in the equation we derived in Figure 4.2. The solution is quite simple we will once again use trigonometry rules, but this time instead we will find the angle using the points (x_1, y_1) and (x_2, y_2). If you imagine a right angle triangle formed by those two points in Figure 4.1 you will notice that θ is the same one derived by forming a right-angle triangle using the points (x_1, y_1) and (x, y).

$$\tan\theta = \frac{opp}{adj} \qquad opp = |y_2 - y_1| \qquad adj = |x_2 - x_1|$$

$$\theta = \tan^{-1}\left(\frac{opp}{adj}\right) \qquad \theta = \tan^{-1}\left(\frac{|y_2 - y_1|}{|x_2 - x_1|}\right)$$

After figuring out how to compute θ this mainly wraps up the math behind getting the (x, y) coordinate on the surface of a circle. Simply all that we need to do after computing the *opp* and the *adj* in Figure 4.2 is to add or subtract the *opp* and *adj* to the point (x_1, y_1) based on which quadrants the point lies on inside the circle. Final remark you can imagine a circle with 4 quadrants as shown in Figure 4.3 where the following rules apply:

- x-coordinate rules:
  - if x-coordinate lies on the right of x_1, then $x = x_1 + adj$
  - if x-coordinate lies on the left of x_1, then $x = x_1 - adj$
- y-coordinate rules:
  - if y-coordinate lies above y_1, then $y = y_1 + opp$
  - if y-coordinate lies below y_1, then $y = y_1 - opp$

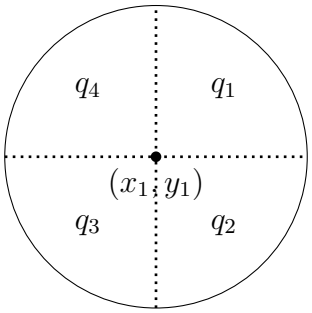


Figure 4.3: 4 Circle Quadrants

4.2 Edge Weight Following Line Position And Angle

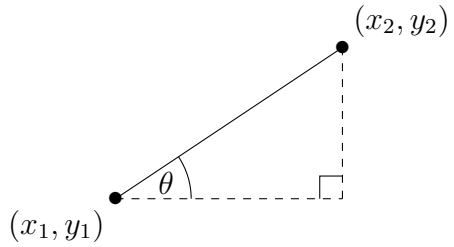
In this section, we will demonstrate the math used to have an edge weight follow the line position and angle, regardless of how the line is positioned. For this section, you could think of an edge weight placed on top of a line as some text, and you can even think of that text as a rectangle that should be placed on the line. Without further ado, let's define the problem formally as follows:

Problem 2 *Given a line defined by the points* (x_1, y_1) *and* (x_2, y_2)*, and a rectangle centered at coordinate* (x_r, y_r)*. Our goal is to rotate and position this rectangle in the middle of the line.*

Now this problem can be divided into two sub-problems:

1. Finding the angle required to rotate the rectangle such that it has the same angle as the line.
2. Finding the coordinate at the line that represents its midpoint, so that we can update the rectangle's center coordinate (x_r, y_r) to be positioned at the middle of the line.

Let us focus on the first subproblem in this subproblem all we need to do is find the angle made by the line and rotate the rectangle by this angle, and we can find the line angle simply as shown in Figure 4.4



$$\tan\theta = \frac{opp}{adj} \qquad opp = |y_2 - y_1| \qquad adj = |x_2 - x_1|$$

$$\theta = \tan^{-1}\left(\frac{opp}{adj}\right) \qquad \theta = \tan^{-1}\left(\frac{|y_2 - y_1|}{|x_2 - x_1|}\right)$$

Figure 4.4: Finding angle of the line

Now let's shift our focus to the second sub-problem which is finding the midpoint coordinate of the line. That can be done easily by averaging the sum of both coordinates, as following

$$x_{mid} = \frac{x_1 + x_2}{2} \qquad y_{mid} = \frac{y_1 + y_2}{2}$$

Figure 4.5 shows the result after rotating the rectangle and centering it at the midpoint coordinate of the line.

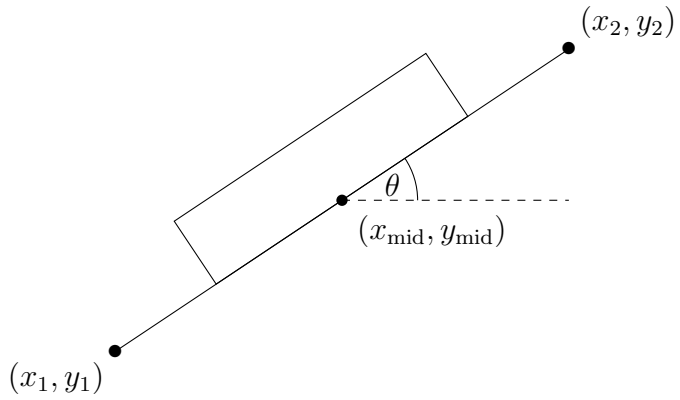


Figure 4.5: Rectangle On Line

5 Code Architecture & Design

In this chapter we will shift our focus to the design aspect of the project, we will focus on:

5.1 Code Architecture High-level description of how the software works, and the interactions happening between the different components within the software.

5.2 Code Design More detailed view of how the code works, we will review the UML class diagram of the software, and how the classes interact with each other.

5.3 Design Challenges In this section we will dive into the design challenges encountered during development, and we will examine the design choices made to overcome those challenges.

5.1 Code Architecture

In this section, we will go through the big picture of how the code works. We will mainly discuss the major software components, and describe the role of each component and its relation to other components. Additionally, note that the terms "component" and "module" will be used interchangeably throughout this chapter.

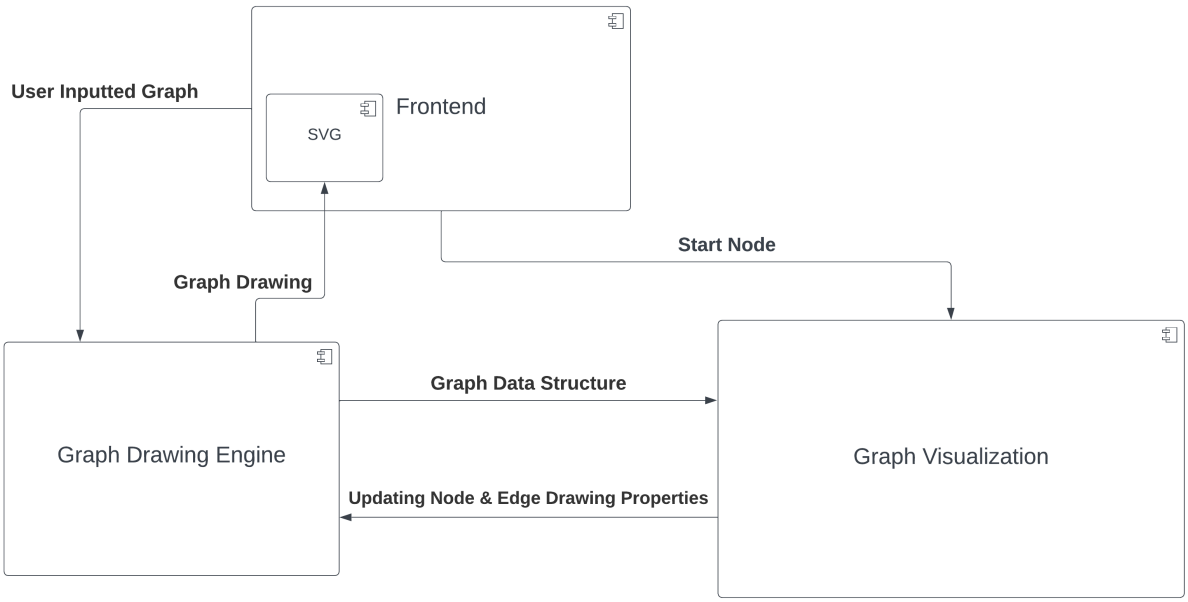


Figure 5.1: Code Architecture

Let's now describe the role of each component in Figure 5.1

Frontend The frontend is responsible for handling displaying the user interface (UI), and handling all the interactions made by the user through the UI. For example, reading graph input from the user.

SVG Scalable Vector Graphics (SVG), is a sub-component of the Frontend, mainly used for displaying the Graph Drawing to the user in the frontend.

Graph Drawing Engine The graph drawing engine is one of the most important components in our software, and it is mainly responsible for three tasks:

1. Drawing the graph directly into the SVG in the frontend.
2. Storing a classical graph data structure representing the graph input from the frontend.
3. Mapping the graph input from the frontend into the classical graph data structure, and vice versa.

Graph Visualizer The graph visualizer's main role is to visualize different graph algorithms, by changing the visual attributes of the graph drawing at each step of the algorithm visualization.

Let's now go through the interactions made between the components shown in Figure 5.1

Frontend The frontend has several interactions with other components, but the main interactions are:

1. Reading graph input from the user as text and parsing this text into a suitable graph representation that is then passed to the Graph Drawing Engine for drawing.
2. Reading the start node, and passing it to one of the available algorithm visualizers in the Graph Visualizer module.

SVG Mainly responsible for displaying the graph drawing from the Graph Drawing Engine module.

Graph Drawing Engine Receives the parsed graph from the frontend module. As nodes in the graph received by the frontend are represented as text, the graph drawing engine maps the graph received from the frontend to a classical graph. It then draws the graph it is storing into the SVG module in the frontend. Also, sends the classical graph data structure to the Graph Visualization module to be used as input to the different available graph algorithms.

Graph Visualization The graph visualizer mainly receives the start node from the frontend, which will be used as input to one of the available graph algorithms. Next, the graph visualizer will request the classical graph data structure from the Graph Drawing Engine to run the graph algorithm. The graph visualization module will also change the visual properties of the nodes and edges in the graph drawing engine to visualize the execution steps of the graph algorithm.

That mainly concludes our discussion on the code architecture. We have just seen a high-level overview of how the software is structured and working. In the upcoming sections, we will dive into more details of the design of the software.

5.2 Code Design

In this section, we will describe the code design in more depth, by providing the UML class diagram of the project. We will also describe the classes and their interactions.

First before going into explaining the code design you need to keep in mind the following points:

- The UML Class diagram does not match exactly the code and that is for two main reasons. First, JavaScript as a language is not fully an object-oriented language which means a UML class diagram might not always be able to describe specific parts of the code. Second, I tried to make the UML class diagram as generic as possible and not tied to any specific language or technology. Plus it would be very hard to make a UML class diagram matching every bit of the code it will be very large and contain a lot of information which will make it hard to understand the high-level design of the code.

- JavaScript as a language only supports dynamic arrays, so in the UML class diagram the keyword **Array** is used to denote a dynamic array while the bracket symbol **[]** is used to indicate an array of fixed size. The main reason for such a convention is to make the UML class diagram as generic as possible.

- In JavaScript, object literals and arrays containing different data types are referred to as Tuples in the UML class diagram, and this is mainly to show a collection of heterogeneous values.

- In the UML class diagram some classes have been removed, as they are not part of the core functionalities of the software.

- All class constructors have been removed from the UML class diagram. The main reason was that all the class constructors in this project would mainly initialize the data members of their respective classes.

- Last but not least I have used a class called SVGElement in the UML class diagram. Although this class is provided by the DOM API, it was hard to remove it because SVG is the main technology used to display the graph. Also, the class name was very descriptive and intuitive.

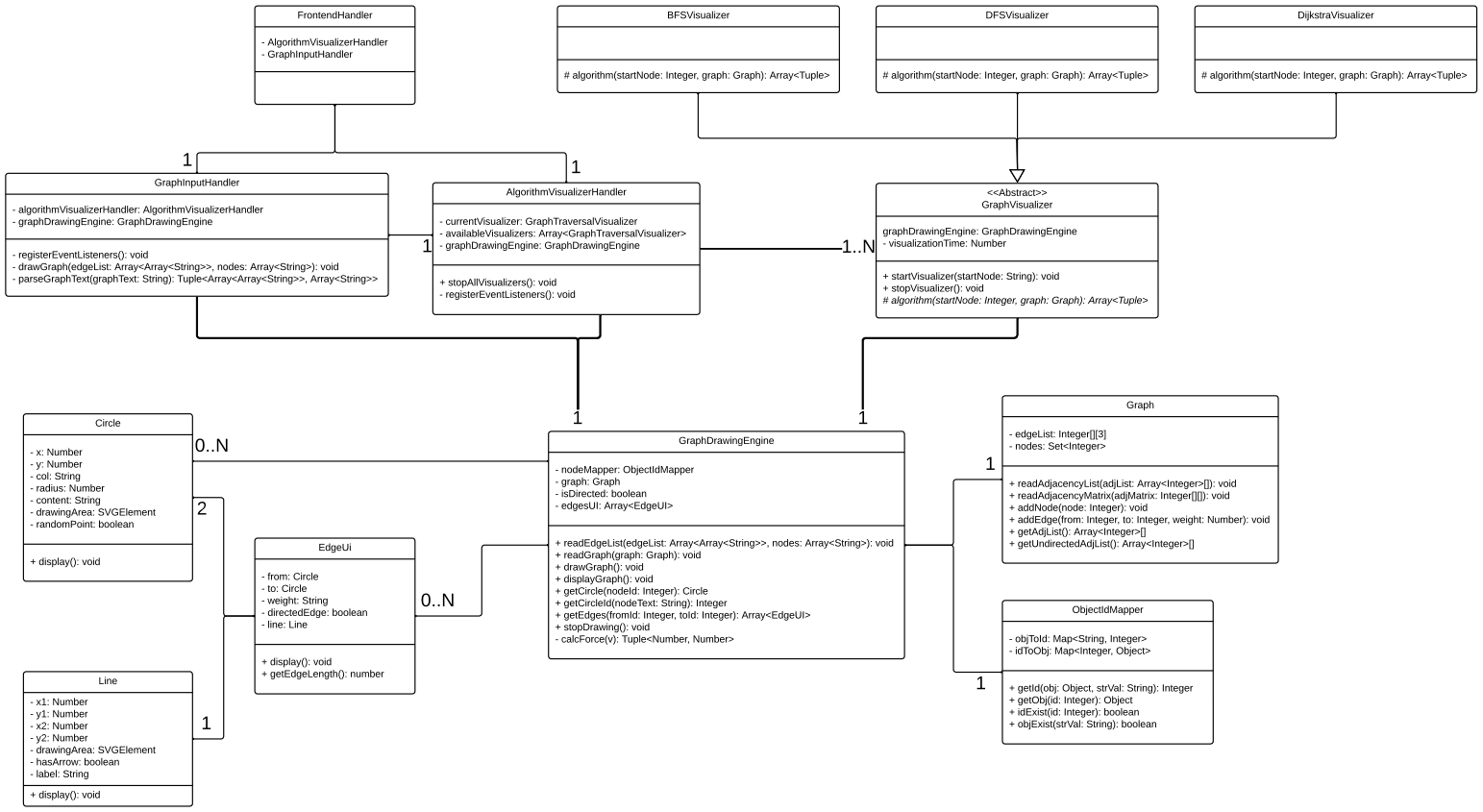


Figure 5.2: UML Class Diagram

Now let's go through explaining each class

Circle This class represents a circle containing text. It mainly has the *display()* method which displays the circle into the *drawingArea*, which is the SVG container displayed in the frontend to the user.

Line This class represents a line with a label. Optionally, the line could have an arrow at its end. It mainly has the *display()* method to display the line into the *drawingArea*.

EdgeUi This class is responsible for displaying a line between 2 circles to simulate an edge in the graph. The edge also has a weight and can be set to be directed or undirected.

Graph This class encapsulates a classical graph data structure, and internally uses an edge list representation. It provides methods for adding nodes and edges to the graph, as well as numerous methods for reading and retrieving the graph in different representations, such as adjacency lists and adjacency matrices.

ObjectIdMapper Class responsible for mapping objects to integer IDs. The method *getId(obj)* is responsible for returning the ID associated with the object or creating a new ID if the object does not have a mapping. The method *getObj(id)* is responsible for returning the object mapped to the given ID.

GraphDrawingEngine The GraphDrawingEngine class is responsible for drawing the graph into the SVG Container in the frontend. It has the following main methods:

- *readEdgeList(edgeList, nodes)* This method receives an edge-list representation of the graph from the frontend, where nodes are represented as strings. It maps the graph received from the frontend to a classical graph, creates a *Circle* instance for each node, and creates an *EdgeUI* instance for each edge.
- *drawGraph()* This method is responsible for drawing the graph into the SVG container in the frontend by running the spring-embedder algorithm.
- *displayGraph()* This method is responsible for refreshing the SVG container and displaying all the nodes and edges in the graph.
- *getCircle(nodeId)* Returns the circle instance mapped to the given node ID.
- *getCircleId(nodeText)* Returns the node ID mapped with the given node string.
- *stopDrawing()* stops the drawing process.
- *calcForce(v)* returns a pair of numbers representing the x-component and y-component forces on vertex v.

GraphVisualizer An abstract class to represent any graph algorithm visualizer. It has the following main methods:

- *startVisualizer(startNode)* This method receives the start node and starts the visualization process. During the visualization process, it directly updates the visual attributes of the nodes and edges stored inside the *GraphDrawingEngine*.
- *stopVisualizer()* This method stops the running of the current visualizer.
- *algorithm(startNode, graph)* An abstract method that receives the start node and a *Graph* instance, and returns an array of the steps taken during the algorithm execution.

BFSVisualizer Extends the *GraphVisualizer* class and overrides the *algorithm(startNode, graph)* method to execute the BFS algorithm.

DFSVisualizer Extends the *GraphVisualizer* class and overrides the *algorithm(startNode, graph)* method to execute the DFS algorithm.

DijkstraVisualizer Extends the *GraphVisualizer* class and overrides the *algorithm(startNode, graph)* method to execute Dijkstra's algorithm.

GraphInputHandler This Class handles the input of graph data from the user interface in text format and parses it into an edge list representation. This parsed data is then used by the *GraphDrawingEngine* for drawing the graph.

AlgorithmVisualizerHandler This Class handles inputting the start node and the chosen algorithm visualizer from the user interface. The start node will then be passed to the chosen visualizer to start the visualization of the algorithm steps.

FrontendHandler This class stores and instantiates all the frontend handlers, specifically the *GraphInputHandler* and *AlgorithmVisualizerHandler*.

Let's wrap up this section with some final remarks about the UML class diagram. First, you might have noticed that the *GraphInputHandler* has a single instance of the *AlgorithVisualizerHandler*. The main reason was to stop any graph algorithm visualizer currently running, once the user inputs a new graph.

Secondly, you may have noticed that both the *GraphInputHandler* and *AlgorithmVisualizerHandler* have an instance of the *GraphDrawingEngine* class. The reason is that the *GraphInputHandler* passes the graph input from the user to the *GraphDrawingEngine* to display the graph. Similarly, the *AlgorithVisualizerHandler* instantiates all the visualizers and passes the *GraphDrawingEngine* to each visualizer instance.

Thirdly, you might have noticed that inside the *ObjectIdMapper* we are mapping a string to an ID instead of an object to an ID. The main reason for that is a limitation in JavaScript as the Map data structure in JavaScript uses direct equality with object references. Therefore, even if two objects have the same content, if they have different addresses, they are not considered equal. For that reason, we are using a string value to represent an object instead.

Fourthly, you might wonder why we use a classical graph data structure inside the *GraphDrawingEngine*. You might suggest that we could represent the graph explicitly as a set of *Circle* instances for nodes and *EdgeUi* instances for edges. While this approach would have a direct mapping to the graph drawn in the SVG container in the frontend, I decided to map *Circle* instances to numeric IDs representing the nodes, and *EdgeUi* instances as edges between these *Circle* IDs.

This decision was made early during implementation, and it was intended to decouple the graph drawn in the frontend from the graph used by all the graph algorithms, such as BFS, DFS, Dijkstra, and even the Spring-Embedder algorithm. While these algorithms could be modified to run with our user-defined graph data structure, I wanted to provide a standard graph data structure that any graph algorithm could use without needing knowledge about our specific user-defined graph data structure used for drawing in the frontend.

5.3 Design Challenges

In this section, we will discuss design challenges arising during implementation and how they were overcome to improve the overall design of the software. Although most of the design decisions discussed in this section will be reflected in the Code Design Section 5.2, I wanted to show the rationale and reason for achieving certain design choices that were presented in the previous section and the challenges that led to those designs.

Challenge 1, you might have noticed in the architecture diagram 5.1 that the *GraphDrawingEngine* draws the graph directly to the SVG Container in the frontend. However, since the SVG Container is part of the HTML displayed on the website, we rely on the DOM API provided by the browser to manipulate the HTML elements in JavaScript. The fundamental problem here is that we do not want the *GraphDrawingEngine* class to be tightly coupled with any specific technology used by the frontend. Simply stated we do not want the *GraphDrawingEngine* to rely on the DOM API to perform the drawing in the frontend.

The solution for that problem was to abstract the interaction with the SVG container through high-level classes that encapsulate the usage of the DOM API. Those classes are the *Circle* and *Line*, they are designed to be abstract enough to be used by the *GraphDrawingEngine*. Internally those classes use the DOM API for creating their corresponding SVG elements and adding those elements to the SVG container in the frontend. Thus all that the *GraphDrawingEngine* needs, is to just call the method *display()* in any of those classes to display those elements in the frontend.

Challenge 2: Before describing the challenge, let's first review how the algorithm

visualization process works. The description provided here is a high-level overview, avoiding extensive implementation details, as this chapter primarily focuses on software design.

Initially, the visualizer executes the graph algorithm normally. At specific points during the algorithm's execution, the visualizer modifies the visual properties of some nodes or edges in the graph to illustrate the current step within the algorithm.

The visualizer achieves this by requesting a *Circle* instance for a node or an *EdgeUi* instance for an edge from the *GraphDrawingEngine*. After applying a visual change, the visualizer re-displays the graph using the *GraphDrawingEngine*.

To simulate the animation of the algorithm's steps, the algorithm pauses for a fixed time before applying the next visual change to the graph.

Having provided a high-level overview of how the visualizer functions, let's now describe the problem. Initially, all the logic described above was implemented in a single method. This means that if a new graph algorithm visualizer were to be added in the future, a new method would need to be created including:

1. the graph algorithm itself.
2. changing the visual properties at specific points in the algorithm.
3. pausing to allow the animation of the algorithm steps.
4. re-displaying the graph at each step using the *GraphDrawingEngine*.

From the above list, it is clear that a new graph algorithm visualizer only needs to implement the graph algorithm itself and the specific points in the algorithm that will be visualized.

One might suggest splitting each specific task into a separate method. While this would improve the code organization, the method running the graph algorithm would still need to be concurrent. It would have to pause at each step to simulate animation. Even if the logic for pausing is implemented separately, it would need to wait until the pausing method finishes before continuing with the algorithm. This still results in a complex method that handles multiple tasks.

The solution achieved is to run the complete graph algorithm without any visualization or re-drawing. Instead, the method implementing the graph algorithm will return an array containing all the visualization steps. Afterward, another method will be responsible for going through each step in the array and performing all the tasks described above.

Thus, adding new graph algorithm visualizers becomes very straightforward. The new graph algorithm needs to implement the standard graph algorithm, along with the specific points to be visualized during the algorithm execution. Once the algorithm finishes execution an array containing all the steps is returned. This also allows for extended functionalities, like stepping forward and backward in the visualization process.

In the UML class diagram in the previous section, you can observe that the class *GraphVisualizer* has an abstract method called *algorithm(startNode, graph)*. This method is overridden by any new subclass of the algorithm visualizer class and is expected to return an array containing all the steps taken during the algorithm execution.

The *startVisualizer(startNode)* method inside the *GraphVisualizer* calls the abstract method *algorithm(startNode, graph)*. It is responsible for all the common tasks described above by going through each step in the array returned by the specific visualizer. The technique just used is known as the template method design pattern. I will not describe how the steps array looks like as that is an implementation detail. However, for those

interested in understanding how this is implemented, please refer to the **traversals.mjs** file, which contains all the code logic for the visualizer.

6 Implementation

In this chapter, we will discuss a variety of topics, covering instructions on how to run the code and how to navigate the project website. Also, we will discuss additional implementation details such as the internal representation of the graph data structure, and finally, we will address the concurrency challenges faced during implementation.

6.1 Website Usage

In this section, we will mainly focus on how to run the code, and how to use the website functionalities.

First, let's examine how to run the code. These steps assume you have the *live-server* [5] package installed globally using *npm* [12] on your machine. Alternatively, you can use any similar package that supports creating a local server. However, please note that it is the reader's responsibility to run the code if a different method is chosen. It is worth mentioning that we use a local server to prevent Cross-Origin Resource Sharing (CORS) [8][15] errors, as our JavaScript code is defined across multiple files.

Follow these steps to run the code:

1. Open a new command line and navigate to the folder ***"Facilitating-Graph-Exploration-Material/4. Implementation"***.
2. Type ***live-server*** in the command line.
3. Enter the following link in your browser ***http://localhost:8080/***.

Now let's understand how to use the website. You can get an overview of the website in Figure 6.1

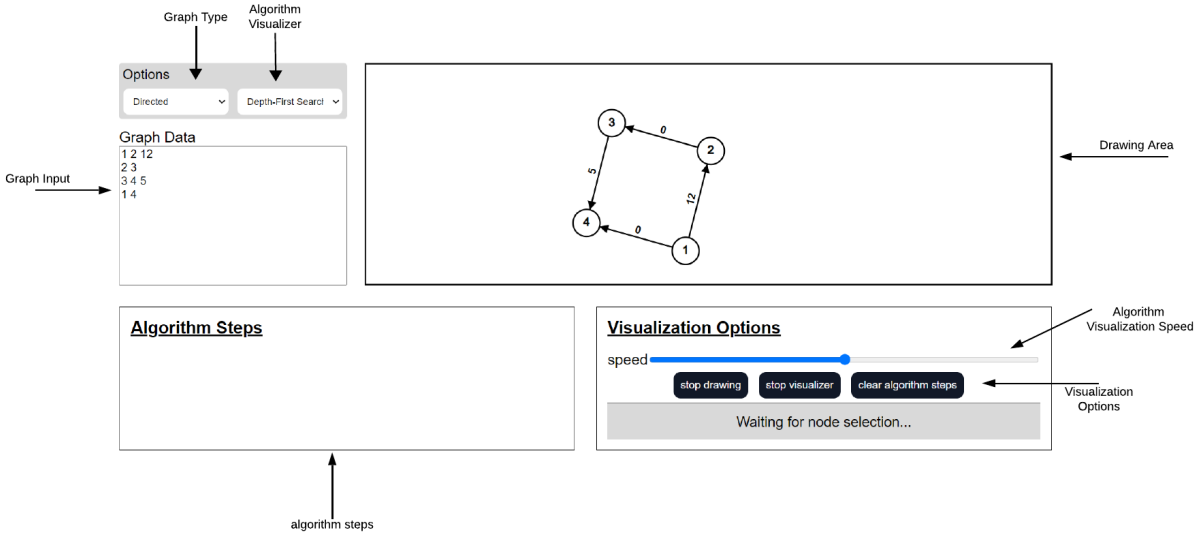


Figure 6.1: Website Overview

Now let's go through each part of the website:

1. *Graph Input* – That is the text area where users can input their graph. Each line represents an edge in the graph, formatted as (u, v, w), indicating an edge from node u to node v with weight w. If the weight of an edge is not specified, it is assumed to be zero.

2. *Graph Type* – Here users have the option to display the graph either as directed or undirected.

3. *Algorithm Visualizer* – Here, users can select the algorithm to visualize on the graph.

4. *Drawing Area* – The area that shows the graph drawing, and the algorithm visualization steps.

5. *Algorithm Visualization Speed* – An input slider to adjust the speed of the algorithm visualization.

6. *Visualization Options* – Several options for the user, including:
   - *stop drawing*, stops the graph drawing.
   - *stop visualizer*, stops the algorithm visualizer.
   - *clear algorithm steps*, clears the algorithm steps area.

7. *algorithm steps* – Displays the trace of execution steps for the algorithm visualizer while it is running.

To run any algorithm visualizer on the graph, the user needs to click on any of the graph nodes. This node will then be used as the start node for the algorithm. An example of running DFS on the graph is shown in Figure 6.2

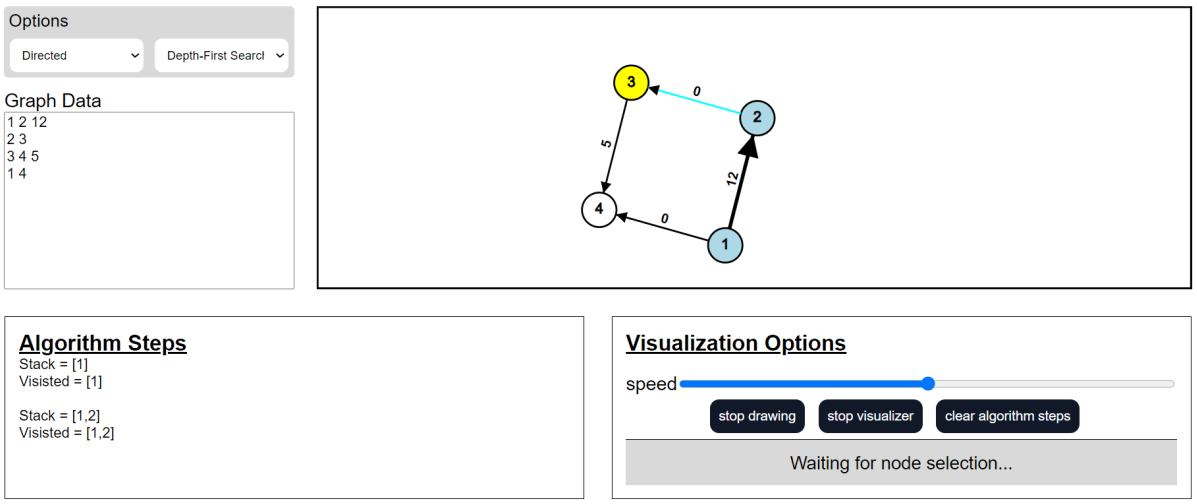


Figure 6.2: DFS Algorithm Visualizer

Final remarks:

- While there is an option to display undirected graphs, it is important to note that when running the algorithm visualizers, they operate on the directed graph version inputted by the user.
- Nodes can be labeled with text, as shown in Figure 6.3.

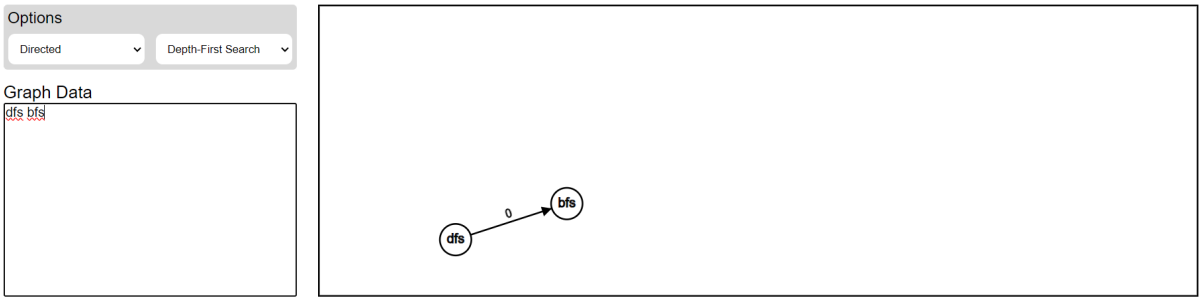


Figure 6.3: Nodes As Text

6.2 Graph Data Structure

In this section, we will go through the internal details of the *Graph* class, which is responsible for storing the graph data structure. This section will be relatively short as we will not cover all the methods provided by the *Graph* class. Instead, we will focus on the internal data structure used, and the most important methods provided by the *Graph* class. Before starting, you can find the *Graph* class code in the file ***"graph/graph-visualizer.mjs"***.

The *Graph* class internally represents the graph as an edge list. We have an array containing the weighted edges of the graph, along with a set containing the graph nodes.

The *Graph* class offers a versatile set of methods supporting various graph representations, primarily adjacency lists and adjacency matrices, as well as different graph types such as

directed, undirected, weighted, and unweighted. Specifically, the *Graph* class can read both weighted and unweighted adjacency lists and adjacency matrices. When reading unweighted graphs, edge weights are assumed to be zero. Furthermore, the *Graph* class can return the graph in adjacency list representation, which may be weighted directed, weighted undirected, unweighted directed, or unweighted undirected.

Listing 6.1 shows a subset of the main methods and internal data members of the *Graph* class.

```
export class Graph {
  constructor() {
    // edgeList: array containing a tuple {from: val, to: val, weight
       : val}, we allow multi-edges
    this.edgeList = [];

    // nods: set of all nodes
    this.nodes = new Set();
  }

  readAdjacencyList(adjList) {
    const n = adjList.length;
    const weightedAdjList = this.#initializeAdjList(n, Array);

    for (let u = 0; u < n; u++)
      for (const v of adjList[u])
        weightedAdjList[u].push({to: v, weight: 0});

    this.readAdjacencyListWithWeights(weightedAdjList);
  }

  readAdjacencyListWithWeights(adjList) {
    for (let u = 0; u < adjList.length; u++)
      this.addNode(u);

    for (let from = 0; from < adjList.length; from++)
      for (const neighbour of adjList[from])
        this.addEdge(from, neighbour.to, neighbour.weight);
  }

  addNode(node) {
    if (typeof node !== "number")
      throw new Error("Invalid Arguments Number Expected");

    this.nodes.add(node);
  }

  addEdge(from, to, weight = 0) {
    if (typeof from !== "number" || typeof to !== "number" || typeof
       weight !== "number")
      throw new Error("Invalid Arguments Number Expected");

    if (from !== to) // not a self-loop edge
      this.edgeList.push({from, to, weight});
```

```
43
44      this.addNode(from);
45      this.addNode(to);
46    }
47
48    getAdjList() {
49      return this.#removeAdjListWeights(this.getAdjListWithWeights());
50    }
51
52    getAdjListWithWeights() {
53      const adjList = this.#initializeAdjList(this.nodes.size, Array);
54
55      for (const {from, to, weight} of this.edgeList)
56        adjList[from].push({to, weight});
57
58      return adjList;
59    }
60
61    clear() {
62      this.edgeList = [];
63      this.nodes.clear();
64    }
```

Listing 6.1: Graph Class

6.3 Concurrency in JavaScript

In this section, I will provide a brief overview of how concurrency works in JavaScript. Although I will briefly explain concurrency in JS, I highly recommend watching the YouTube video titled *'What the heck is the event loop anyway? | Philip Roberts | JSConf EU'* [14]. It offers a great presentation to get a better understanding of how concurrency works in JavaScript.

The first important aspect to understand about JavaScript (JS) is that it operates as a single-threaded programming language, but at the same time, it supports concurrency. This might initially seem confusing – how can a language be both single-threaded and concurrent?

Unlike multi-threading languages, where each thread has its call stack, allowing multiple functions to run simultaneously at the same time, each with its stack frame in separate stacks, JS operates differently. In JS, functions have access to only a single stack, where they can store their stack frames.

Now, the important question is how JS could be concurrent, and the simple answer is through the help of Web APIs. Web APIs are functions provided by the web browser, and those functions generally run in their own threads. This means that during the execution of those Web APIs, the JS code can technically run other tasks by utilizing the call stack it has.

Now you might wonder, once the Web API finishes its work, how does it return its values to be executed in JS? The answer is through callbacks. But what are callbacks? Callbacks are functions passed to other functions as parameters. In JS, you can pass any function to any other function as a normal value. Now, imagine we have a Web API

called ***doUsefulStuff(callback)***. Once this function finishes its work, it calls the callback function provided to it. This callback function is defined in the JS code.

You might ask, once the callback is called, does it run immediately and create a stack frame in the call stack? The answer is no. This is because there could already be function call frames added to the call stack, which implies that we cannot simply add the callback stack frame to the call stack. The solution is to use the ***Event Loop*** and the ***Callback Queue***.

Now, what are those two additional things? Once the callback has been called by a Web API, it will not be added to the call stack immediately. Instead, it will be added to a callback queue, waiting to be executed, and have its stack frame entry added to the call stack. The event loop is responsible for two things: first checking that the call stack is empty and no functions are running; second, it takes the first callback function waiting in the callback queue and runs the function. Once the function runs, it will now create a stack frame in the call stack and start execution normally.

That is how JS achieves concurrency while being a single-thread programming language. Understanding the concepts presented in this section might be challenging, particularly because JavaScript operates quite differently from other programming languages. In the next section 6.4, we will discuss the implementation challenges arising from using concurrency in JS.

6.4 Concurrency Challenge in JavaScript

In this section, we will describe an implementation challenge encountered during the project, arising from using concurrency in JavaScript. One of the most striking things in JS compared to other languages is the lack of control over callback functions in the callback queue. In languages like Java or C, we can create threads and kill them at any point in time. In JS we do not have such control simply once a callback has been added to the callback queue it must execute and we can never stop it.

Before diving into the challenge details, let's first describe when we use concurrency in the project. Concurrency is used for both the drawing process and the algorithm visualization process. The reason for that is that after each frame in the animation process, we need to stop the execution to allow a smooth animation. However, we also do not want to block other functions from running to prevent the website from freezing. To achieve animation without interrupting the rest of the execution of the website we implement the drawing process and the algorithm visualization process asynchronously. You can find simplified code for the drawing process in Listing 6.2 and the algorithm visualization process in Listing 6.3.

```
async drawGraph() {
  // we draw the graph for a fixed number of iterations/frames
  for (let i = 0; i < 250; i++) {
    // we loop over all the vertices in the graph
    for (let v = 0; v < undirectedAdjList.length; v++) {
      displayGraph();
      updateNodePositions(v);
      await sleep(1);
    }
  }
}
```

Listing 6.2: JavaScript code for drawing a graph

```
async startVisualizer(startNode) {
  // resets the graph visual attributes
  resetGraph();

  const nodeId = graphDrawingEngine.getCircleId(startNode);
  const algorithmSteps = _algorithm(nodeId, this.graphDrawingEngine.
     getGraph());

  for (const step of algorithmSteps) {
    updateGraphVisualAttributes(step);
    // display graph directly after update
    graphDrawingEngine.displayGraph();
    await sleep(visualizationTime);
  }
}
```

Listing 6.3: JavaScript code for algorithm visualization

There are a few things to note about the function *drawGraph()* in Listing 6.2. Firstly, the *drawGraph()* function will enter the callback queue 250 times. Whenever we reach the *sleep()* statement, the function will be stopped and the stack frame of the function will be removed from the call stack and stored by the JS runtime engine [9] running JS. Next, the function will be added to the callback queue. Once the *sleep()* call finishes running, the event loop will re-execute the function again. [1]

Secondly, it is important to note that the data used by the *drawGraph()* function is stored globally. This leads to an interesting observation: if we have an ordered sequence S containing all the animation frames needed for the graph drawing process, where $S[i]$ represents the ith frame in the animation, then the following fact holds. When two *drawGraph()* calls are made, they run concurrently. The first call generates the $S[0]$ animation frame and then stops, and is added to the callback queue. Then, another *drawGraph()* call is made, which generates $S[1]$. This is possible because all the data required by the function is stored globally, allowing the changes made by the first function call to be used by the second one. These changes primarily involve updating the node positions, as shown in line 10 of Listing 6.2. As a result, a total of 500 frames are generated by both function calls, with the sequence of frames produced by the first call being $S[0]$, $S[2]$, $S[4], ...$, while the frames generated by the second call being $S[1]$, $S[3]$, $S[5], ...$ Therefore, roughly double the amount of time is required to complete the execution of both function calls. It is worth noting that data inconsistency never occurs when writing to the global data due to the single-threaded nature of JavaScript.

The challenge arises when multiple asynchronous functions are running concurrently, but we only require the latest one to execute, with all previous calls needing to be terminated. You might wonder why we need to stop previous asynchronous function calls, and there are two main reasons.

If more than a single *startVisualizer(startNode)* function runs concurrently, both functions execute different visualizers simultaneously, which is not intended. For instance,

[1] sleep() is a custom function created using the setTimeout() function [11], which is one of the functions provided by the Web APIs.

running both the BFS and DFS visualizer simultaneously would not be relevant to the user. In such cases, it is necessary to stop all previous calls made to the visualizer function to start executing the new visualizer function call.

Secondly, there are performance reasons, especially in the case of *drawGraph()*. While it may technically function properly with multiple concurrent function calls, this approach can lead to performance issues. The presence of numerous unnecessary concurrent functions stored in the callback queue can cause delays in executing other callbacks added later to the callback queue.

In the next section 6.5 we will go through the solution used to stop asynchronous function calls whenever a new asynchronous function is called.

6.5 Concurrency Solution in JavaScript

In this section, we will describe how we have overcome the concurrency challenges described in Section 6.4. During our discussion in this section, we will focus mainly on describing the solution using the simple asynchronous function described in Listing 6.4. We assume that we have made only two calls to the function *foo()*, and we will name them $foo_1()$ and $foo_2()$.

```
async foo(num) {
  for (let i = 1; i <= num; i++) {
    console.log(i);
    await sleep(1000); // sleeping for 1 second
    console.log(i-1);
  }
}
```

Listing 6.4: Simple JavaScript asynchronous function

Now imagine that we have made an initial function call to $foo_1(10)$, and after some time we have made a call $foo_2(5)$, and assume that when we have made the second call $i = 3$ in $foo_1(10)$. First, a thing that you should realize is that if we have made a call $foo_2(5)$ and it started execution that means that the call stack was empty, and if the call stack was empty that implies that $foo_1(10)$ was actually still sleeping, or it might have finished sleeping and still waiting in the callback queue to be re-executed again.

Now we somehow need to add some boolean condition in *foo(num)* to check if it can continue or not, and we should not have the value determining if the function should continue or not in the global scope because this value is different from one function to the other. So, somehow we want to add a local variable inside *foo()* stating if the function can continue execution or not.

Now you might wonder how will the local value stored inside $foo_1(10)$ be changed once a call is made to $foo_2(5)$, and the solution is to wrap the local boolean value inside *foo(num)* in an object, and the reason is that objects in JS are updated by reference, and whenever a call to *foo(num)* is made it will somehow set the boolean value in the previous function call and then create a new object for itself to be used by the next call. Now you might ask where will the object reference used by the function be stored, and the answer is that we will store an object reference in the global scope, and we will update the global object state, and then will create a new object and make the global reference point to the next object instead of the old object and that means that the old object reference is not

accessible anymore except its local function where it was stored in. Listing 6.5 shows the code to achieve what has just been described above. [2]

```
1  // initial object not used by any function
2  let lock = {stopped: false};
3  async function foo(num) {
4    lock.stopped = true; // set current lock to stop
5    const myLock = {stopped: false}; // create new object
6    lock = myLock; // assign global lock with myLock
7
8    for (let i = 1; i <= num; i++) {
9      console.log(i);
10     await sleep(1000); // sleeping for 1 second
11     console.log(i-1);
12   }
13 }
```

Listing 6.5: Initial logic to know when should a function be stopped

As you might have figured out already the code in Listing 6.5 is not yet complete and simply we are not checking the stopping condition in *foo(num)*. The crucial question now is where should we place the condition check? Should it be the first line in the for loop, the last line, or some other point? To answer this question we need to think at which point $foo_2(5)$ might start execution. As we have already mentioned $foo_2(5)$ will only start execution once the call stack is empty, which implies that $foo_1(10)$ is still sleeping or is waiting in the callback queue to be executed. At some point, $foo_2(5)$ will sleep and be removed from the call stack, and at this point $foo_1(10)$ will start running back again. At this exact moment, after $foo_1(10)$ resumes execution, we need to check if the lock object has been set to be stopped by another function call.

If you tried to think of any other position where we could place the conditional statement it would be invalid. For example, if we decided to place such a check before the *await sleep(1000)*, then that would be incorrect. This is because when the function continues execution after sleeping, it will print $i-1$ which should not happen given that we already have stopped the function call. If we tried placing the condition below line 11 in Listing 6.5 that would still be incorrect for the same reason. We would still print $i-1$ even though we have already stopped. Now that we know where is the most accurate place to add the condition, we can write the completed code as shown in Listing 6.6

```
1  // initial object not used by any function
2  let lock = {stopped: false};
3  async function foo(num) {
4    lock.stopped = true; // set current lock to stop
5    const myLock = {stopped: false}; // create new object
6    lock = myLock; // assign global lock with myLock
7
8    for (let i = 1; i <= num; i++) {
9      console.log(i);
10     await sleep(1000); // sleeping for 1 second
11     if (myLock.stopped == true)
12         return;
```

[2] We do not need to worry about race conditions when modifying the object state simply because JS is single-threaded

```
13      console.log(i-1);
14    }
15  }
```

Listing 6.6: Code logic to know when should a function be stopped

Final remark, I have abstracted the technique used in this section, through a class called *SingleAsync*, to be re-used at any place without the need to rewrite the same logic again and again. The *SingleAsync* works exactly as described in this section.

That concludes our discussion about concurrency in JavaScript, the challenges encountered during implementation, and how we have overcome those challenges. If readers are interested in examining the actual code implementation discussed, they can find it below:

1. *startVisualizer(startNode)* – class *GraphVisualizer* in ***"graph/algorithm-visualizers/traversals.***
2. *drawGraph()* – class *GraphDrawingEngine* in ***"graph/graph-visualizer.mjs"***.
3. *sleep()* – in ***"utils/utils.mjs"***.
4. *SingleAsync* – in ***"utils/utils.mjs"***.

7 Conclusion

7.1 Review Of Aims

The primary focus of the project was to visually represent graphs in an aesthetically pleasing manner and to visualize various graph algorithms. Throughout the paper, we have utilized the spring-embedder algorithm for graph drawing and gained an understanding of the underlying physics and mathematics used by the algorithm. Additionally, we encountered several design challenges, such as ensuring that the algorithm visualizers were implemented in a way that allows for future extensibility. Moreover, we discussed implementation challenges, including those related to concurrency, which arose during the process of drawing graphs and running algorithm visualizers on them.

7.2 Future Work

From a features delivery point of view, the project contains all the minimum viable product (MVP) features that have been outlined in the project proposal, as detailed in the appendix A.

Many additional features can be included in this project. The following are some examples:

1. Allow additional algorithm visualizers like Bellman-Ford, Floyd-Warshall, and Khan's algorithm.

2. Allowing users to save the graph drawing as an image with different formats like JPG, PNG, and SVG.

3. Supporting curved edges during the drawing of the graph. This will improve the appearance of nodes connected in both directions, such as those with edges (u, v) and (v, u).

4. Allow users to change the graph drawing properties, like the node size, node color, edge color, and so on.

5. Allow users to modify the colors used during the algorithm visualization process.

6. Implementing lazy loading of the graph drawing, where only differences from previous inputs are rendered instead of rendering the entire graph each time a change is made.

7. Extending the graph drawing engine to be used as an independent library with a fixed application programming interface (API) by any tool that requires a graph drawing service.

8. Allow users to move the nodes in the graph drawing manually.

9. Implementing responsive design [10] for the website to ensure optimal viewing across screen sizes.

10. Having a service on the website for generating different non-isomorphic groups of graphs, possibly utilizing a tool like Nauty and Traces [7].

7.3 Self Reflection

In this section, I will endeavor to reflect on my key learnings, assess my satisfaction with the work completed, and consider areas for potential improvement.

Let's begin by discussing the key learnings from this project. Primarily, I believe there are three significant takeaways. Firstly, learning the spring embedder algorithm stands out as the most crucial learning experience in this project. This algorithm presented the most challenge throughout the project. Its complexity arises from its reliance on various math and physics principles, such as Hooke's law, Coulomb's law, force resolution, basic graph theory, and basic mathematics like trigonometry. Viewing the graph drawing results after implementing this algorithm marked a significant milestone for me, as it fulfilled one of my primary motivations for working on this project: understanding how other software creates visually appealing graphs.

The second key learning revolves around being able to write concurrent code in JavaScript. There have been many challenges, such as making the graph drawing process and graph algorithm visualization process concurrent. Additionally, managing previous asynchronous function calls was challenging with a language that does not provide much control over asynchronous function calls. However, from those challenges, I believe I gained a very good understanding of how concurrency in JavaScript works.

The third key learning involved gaining better experience with designing software, such as splitting the code into classes and modules. Throughout development, I faced several design challenges, including separating the user-inputted graph from the classical graph data structure. Another example of a design challenge was making the development of new graph algorithm visualizers more extensible by using the template method design pattern and returning an array of steps from the visualizers.

Now, an important question: Am I satisfied with what has been achieved? The answer is a clear yes. I have implemented every single feature outlined in the MVP requirements. While I always wanted to do more, reality presented us with time constraints. However, there were several areas where improvements could have been made, as detailed below

- I should have paid more attention to improving the User Interface (UI), as most of the time was dedicated to the code logic. Additionally, the HTML and CSS code was not as well-organized as the JavaScript code handling the website's logic. Another thing was including the "About" and "Help" links on the website without implementing them.

- Better design decisions could have been made, such as completely decoupling the code responsible for drawing the graph from the frontend. This could have been achieved by having the graph drawing engine return a sequence of graph frames that the frontend can use to draw the graph. This approach would have deferred the drawing process to the frontend itself rather than the graph drawing engine.

- I could have made better use of JavaScript's support for functional programming. A notable scenario where functional programming could have been advantageous

is in implementing the template method pattern for the classes handling graph algorithm visualization. Rather than relying on the abstract class *GraphVisualizer* to apply visual changes to nodes and edges based on an array of steps, delegating this responsibility to the subclasses by storing the function responsible for applying visual changes in each step of the algorithm process would have been a more straightforward approach.

- I wish I had allocated more time to learn more algorithms for displaying graphs. It would have been a valuable addition to expand my knowledge in the field of graph drawing, beyond the scope of implementation itself.

Bibliography

[1] CS Academy. CS Academy Graph Editor. https://csacademy.com/app/graph_editor/. Online: accessed 19-March-2024.

[2] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*, chapter 10. Force-Directed Methods, pages 303 – 309. Prentice-Hall, 1999.

[3] Graph Online. https://graphonline.ru/en/. Online: accessed 19-March-2024.

[4] Halim, S. and Halim, F. VisuAlgo. https://visualgo.net/en. Online: accessed 19-March-2024.

[5] live-server contributors. live-server. https://www.npmjs.com/package/live-server, 2024. Online: accessed 19-March-2024.

[6] Discrete Mathematics and Its Applications. *Discrete Mathematics and Its Applications*, chapter 10. Graphs, pages 641 – 641. McGraw-Hill, 7th edition, 2011.

[7] B.D. McKay and A. Piperno. Practical graph isomorphism, *ii*. *Journal of Symbolic Computation*, 60:94–112, 2014. https://doi.org/10.1016/j.jsc.2013.09.003.

[8] Mozilla Developer Network contributors. Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, 2024. Online: accessed 19-March-2024.

[9] Mozilla Developer Network contributors. Engine. https://developer.mozilla.org/en-US/docs/Glossary/Engine, 2024. [Online; accessed 19-March-2024].

[10] Mozilla Developer Network contributors. Responsive design. https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design, 2024. Online: accessed 19-March-2024.

[11] Mozilla Developer Network contributors. setTimeout() global function. https://developer.mozilla.org/en-US/docs/Web/API/setTimeout, 2024. Online: accessed 19-March-2024.

[12] npm contributors. npm. https://www.npmjs.com/, 2024. Online: accessed 19-March-2024.

[13] Pandey, A. D3 Graph Theory. https://d3gt.com/. Online: accessed 19-March-2024.

[14] P. Roberts. What the heck is the event loop anyway? | philip roberts | jsconf eu. https://www.youtube.com/watch?v=8aGhZQkoFbQ, 2014. Online: accessed 19-March-2024.

[15] stackoverflow contributors. JavaScript module not working in browser? https://stackoverflow.com/questions/52139811/javascript-module-not-working-in-browser, 2024. Online: accessed 19-March-2024.

[16] University of San Francisco. CS USFCA. https://www.cs.usfca.edu/~galles/visualization/Algorithms.html. Online: accessed 19-March-2024.

A Original Project Proposal

Facilitating Graph Exploration

Abstract:

Graphs have a long history dating back to the 18th century, and they've been fundamental in computer science. Graphs have been used in a wide range of applications from traffic routing, social media platforms, operating systems, computer networking, and much more.

The main motive of the project comes from the need to provide a solution for researchers, and students who require accessible tools for exploring graphs. The main goal of the project is to create a user-friendly website that combines two essential functionalities: graph visualisation and generation. During the development stage, the focus will primarily be on a subset of features known as the Minimum Viable Product (MVP) features. These features include inputting graphs in different formats, visualising them, and visualising various graph algorithms. The project will mainly use web languages such as HTML, CSS, and JavaScript for development.

1. Introduction:

The proposed project is mainly focused on 2 areas:

1.1 Graph & Graph Algorithm Visualisations

This part of the proposed project aims to create user-friendly visualisations for representing graphs and to make it easier to understand how graph algorithms work. One frequent use case of such a feature is that it is so often that programmers will go solve coding problems where a graph is given in the form of an Adjacency Matrix, Adjacency List, etc. and you want to quickly draw this graph and the classical approach is to manually use pen and paper, but with the help of the graph visualiser people can quickly create and view those graphs.

1.2 Graph Generations

This part of the proposed project aims to generate all sets of graphs with specific properties to be used for experimentation, benchmarking, and various other research applications.

2. Background & Motivation:

There have been previous projects that have similar ideas to this proposed project examples are:

1. CS Academy Graph Editor [3]: Used for graphically displaying Graphs
2. The Combinatorial Object Server [1]: Used to Generate unlabelled non-isomorphic graphs with a given number of nodes with various additional properties using the nauty library [2]
3. Graph Online [4]: Used for graphically displaying Graphs and running algorithm visualisations on Graphs
4. D3 Graph Theory [5]: Used to facilitate interactive learning of graph theory
5. CS USFCA [6]: Used for Algorithm and Data Structure Visualisations

All the above are examples of projects similar to the proposed project, but up to this day there has not been a service that supports both Graph Generation and Visualization at the same place and that is the main focus of the proposed project.

3. The Proposed Project:

3.1 Aims & Objectives:

As discussed in the introduction this project is concerned with facilitating the exploration of graphs. In this section, we will dive deeper into the exact requirements of the project. We will divide the requirements of the project into 3 categories:

A. **MVP Features:** Minimal Viable Product will contain all the set of requirements that is so critical to the project, and those requirements alone are enough for conveying the goals of the project
B. **Extra Features:** Features that would be nice if were included in the project, but if they were not included that would not be a big problem
C. **Miscellaneous Features:** Features that would be nice to have, but out of the scope of this project

A. MVP Features:

1. Allow the user to input a graph in any representation like Adjacency List, Adjacency Matrix, etc.., and display the drawing of the Graph on the website
    a. These features should be able to draw both undirected and directed graphs
2. Allow the user to visualise the following algorithms on the graph:
    a. Depth-First Search (The user has to specify the start and end nodes)
    b. Breadth-First Search (The user has to specify the start and end nodes)
    c. Dijkstra's Shortest Path (The user should specify the start node)
    d. For all the above visualisation the user should have the ability to modify the speed of the visualisation process

B. Extra Features:

1. Allow the user to visualise the following algorithms on the Graph:
    a. Bellman-Ford Shortest Path (The user should specify the start node)
    b. Floyd Warshall Shortest Path (The user should specify the start node)
    c. Khan's Algorithm
    d. Cycle Detection using DFS
2. Should create another webpage on the website that allows users to generate all combinations of undirected non-isomorphic graphs based on the following properties:
    a. Number of Nodes (required field)
    b. Minimum number of edges
    c. Maximum number of edges
    d. Minimum degree
    e. Maximum degree
3. The user should be able to specify the type/family of the graph he wants to generate including the following:
    a. Connected
    b. Biconnected
    c. Bipartite

4. The user should be able to choose the output format of the generated graphs including the following:
    a. Adjacency List
    b. Edge List
    c. Adjacency Matrix
5. All the output of the Generated Graphs should be displayed in the Browser
6. Allow the user to generate all Isomorphic undirected graphs, as the number of isomorphic undirected graphs with **n** nodes is exponential based on the following formula $2^{n(n-1)/2}$ there would be a limit to the number of nodes used
7. Allow the user to click on any of the generated graphs to display and visualise it

C. Misc Features:

1. Allow for additional graph types/families including the following:
    a. Triangle Free
    b. 4 Cycle Free
    c. Allow the users to download the generated graphs as a text file
2. Allow the user to download the graph visualisation as an image including the following image formats:
    a. jpg
    b. png
    c. SVG
3. The website should be able to support random generation and visualisation of mazes and should be able to run DFS & BFS on the maze by treating it as a graph

3.2 Methodology:

Mainly in the Methodology, we will discuss the following points:

a. Software Development Approach Used
b. Software Development Life Cycle (SDLC) of the Project
c. Main Languages and Technologies used in the Project

A. Software Development Approach Used:

The Software Development approach used for this project is the waterfall model, where all requirements are gathered before the development, and that is simply the case as the project has a well-defined set of requirements with a specific goal. By choosing this approach for development I will not be expecting a change in the requirements. One advantage of this approach compared with other approaches is that you have no ambiguity as all the requirements are defined from the start, and a design for the system as a whole can be created for best aiding those set of requirements without needing to think about designs that handle frequently changing requirements.

B. SDLC of the Project:

As we are using the waterfall model we are expected to pass by every stage of the SDLC only once, except for the design and the implementation phases as for each requirement category will have its own design and implementation. The Life Cycle of the project will be as follows:

1. **Requirements Analysis:** This part of the life cycle has already been done in the Aims & Objectives section

2. **Design:** The output of this stage should be a simple diagram representing the architecture of the project I might provide a draft Class Diagram at this stage, but there is a very high chance it will change in the future during the implementation phase
3. **Implementation:** This is the most fun phase at least for me, the expected outcome of this phase is a fully functional project with all MVP Features, and any additional features
4. **Testing:** This is the last phase in the Project Life Cycle, black-box testing will be used for the project and the expected outcome at this phase is a document containing different test cases and their results

C. Main Languages and Technologies used in the Project:
The Project will be implemented as a website, so the main Languages and Technologies used in the project are:

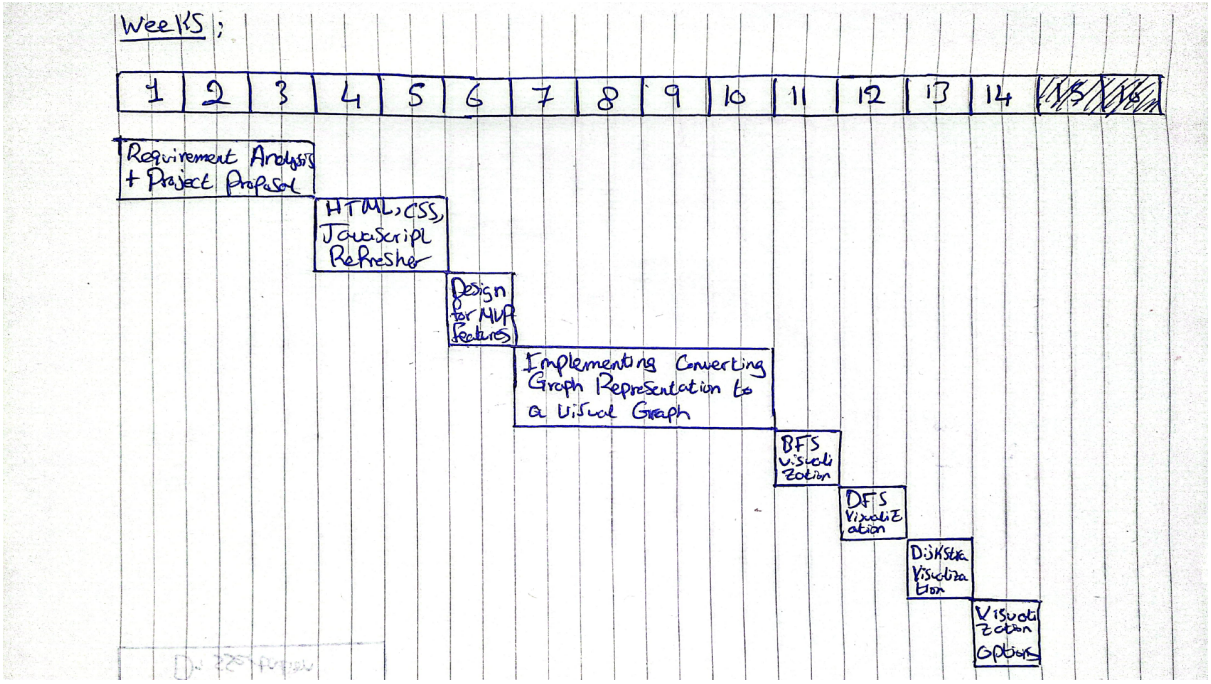
1. HTML
2. CSS
3. JavaScript/TypeScript

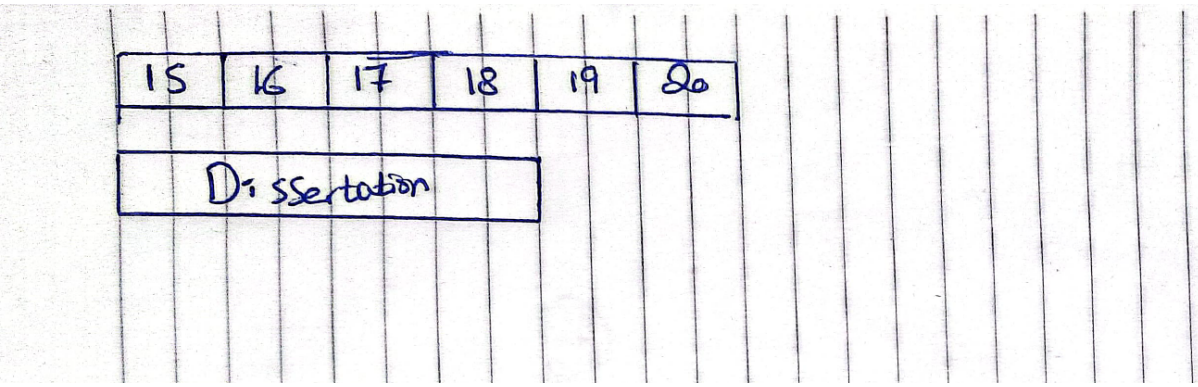
Also, additional frameworks and Libraries may be used, but at this point, it is not determined yet, as a point of reference the following might be used:

1. Frontend: React.js
2. Backend: Express.js
3. Drawing: d3.js or p5

4. Program of Work:

In this section we will discuss the timeline of the project, the timeline provided here is just a draft for reference, and it will only consider the MVP Features, and changes to the timeline during the duration of the project are expected and normal.





5. References:

1. Combinatorial Object Server
   http://combos.org/nauty
2. Nauty and Traces
   https://pallini.di.uniroma1.it/
3. CS Academy Graph Editor
   https://csacademy.com/app/graph_editor/
4. Graph Online
   https://graphonline.ru/en/
5. D3 Graph Theory
   https://d3gt.com/
6. Algorithm and Data Structure Visualisations, University of San Francisco
   https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

B Another Appendix Chapter

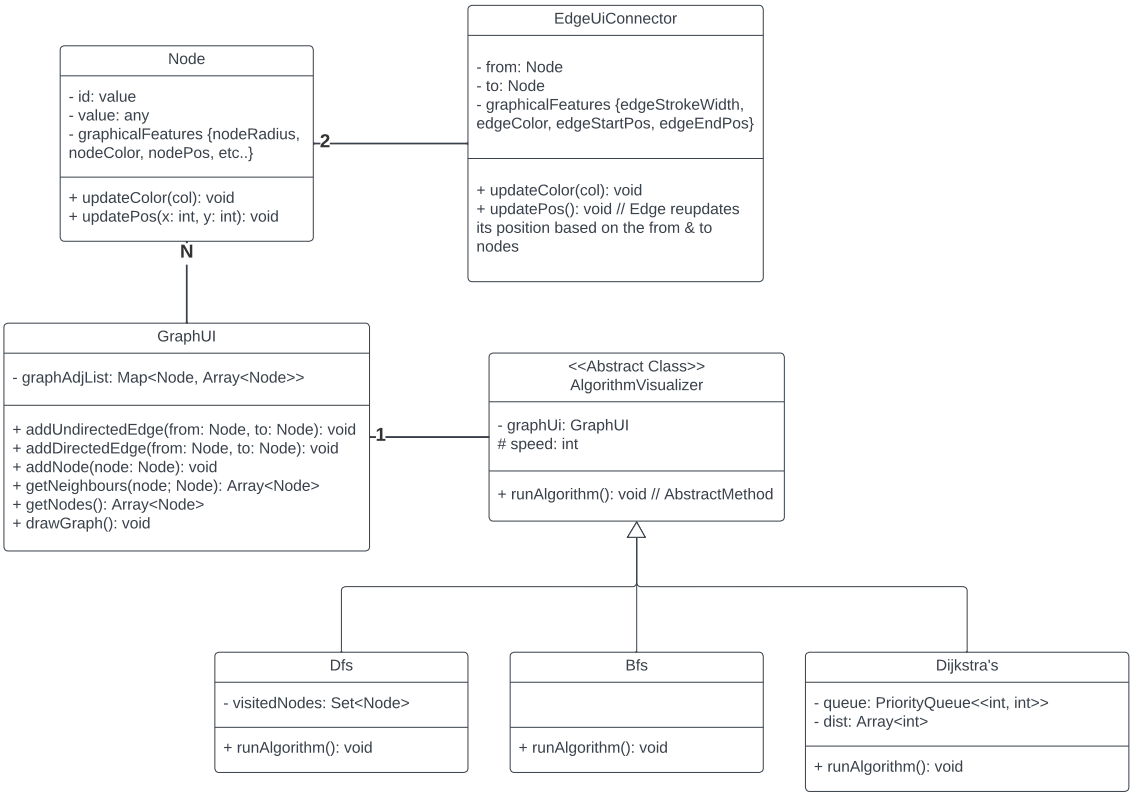


Figure B.1: Initial UML Class Diagram

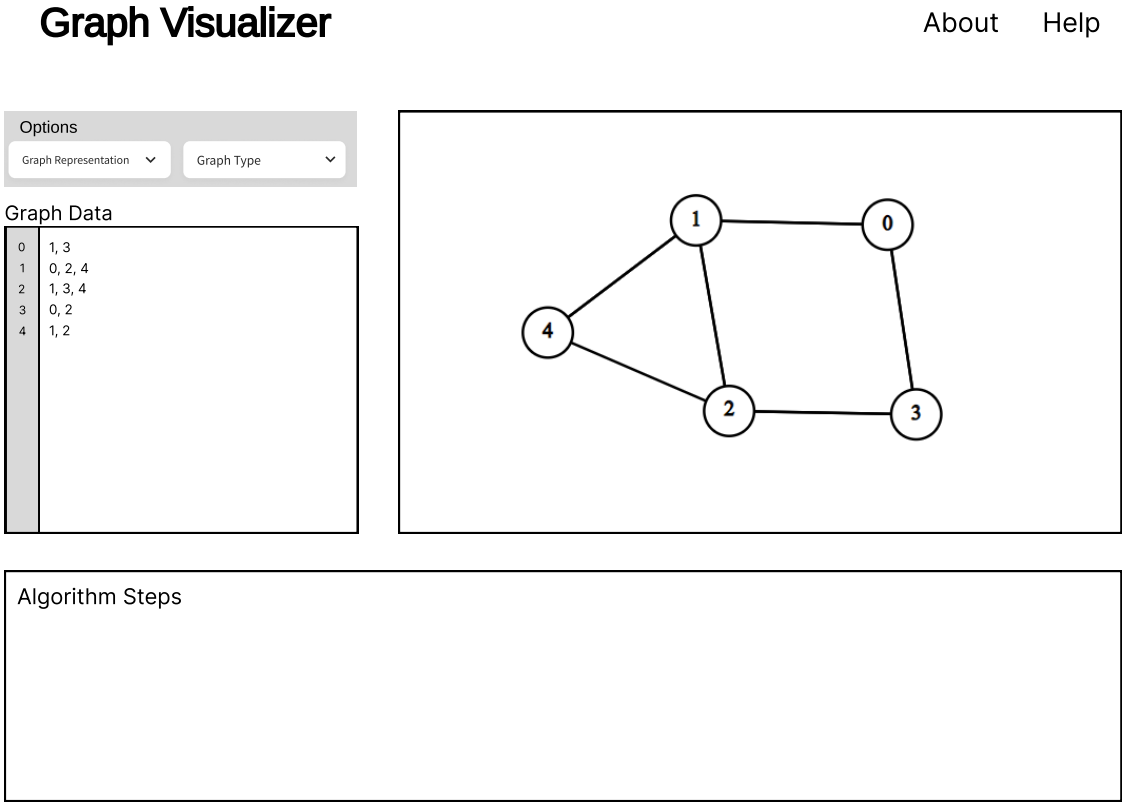


Figure B.2: UI Wireframe