Lancaster University Leipzig School of Computing and Communications

# Facilitating Graph Exploration

Youssef Gerges Ramzy Mokhtar

Date of Submission (e.g., 24/02/2024)

Supervisor: Dr Marco Caminati

B.Sc. (Hons) Computer Science

Number of words = 0
This includes the body of the report only

## Declaration of Originality

Put some text similar to the following in here:

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:

## Abstract

Put your abstract here. You should create a short abstract (200 words at maximum) which is on a page by itself. The abstract should be a very high-level overview: for example 1–2 sentences on the aims of the project, 1–2 sentences on the kind of design, implementation, or empirical work undertaken, and 2–3 sentences summarising the primary contribution or findings from your work. The abstract appears in the front matter of the report: after your title page but before the table of contents.

If you want to dedicate to someone in particular

## Acknowledgements

General acknowledgements . . .
your supervisor, your family, your firends, . . .

# Contents

# List of Figures

# List of Tables

# 1 Introduction

There report assumes basic knowledge in the following topics, also I have bundled a list of resources that I think might be useful for understanding the below mentioned topic in the Additional Reading Chapter

- HTML
- CSS
- JavaScript
- DOM API
- Scalable Vector Graphics (SVG), mainly how are they used with HTML
- Graph Representations, specifically Adjacency List, Adjacency Matrix, and Edge List
- Trigonometric functions, specifically sine, cosine, and Tanget
- Right angle triangles, and Pythagoras theorem
- UML Class Diagrams
- OOP
- Few knowledge of some design patterns, specifically the Template Method design pattern
- Basics of Graph Theory
- Graph Algorithms, specifically DFS, BFS, and Dijkstra
- Graph Representations, specifically Adjacency List, Adjacency Matrix, and Edge List

Note the words nodes and vertices will be used interchangeably through this report. - state the terminologies used throughout the paper4 . The term classical graph data structure, is where all nodes are just simple intelgers and are numbered from 0 to N-1 - adjacney Matrix - adjacency list - Edge list

# 2 Background

# 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 algorithm 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 like charged particle that repel each other, and edges between 2 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.

go through this section review it and finish the todo's

reference the book name

## 3.1 Notations Used

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

refernce the book name

| | |
|---|---|
| $G = (V, E)$ | 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 $u$ & $v$ |
| $f(u, v)$ | hooke's law force exerted by edge $(u, v) \in E$ |
| $g(u, v)$ | coulomb's law force exerted by the vertices $(u, v) \in V \times V$ |
| $P_v = (x_v, y_v)$ | $(x, y)$ 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 |
| $hypo$ | hypothesis |

## 3.2 Physical Simulation

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

For a spring it will have an ideal length which is a length that it doesn't have any force on it whenever the spring is stretch it will always attempt to bend back to the original length and whenever it is bent it always tries to stretch back to its original length, the strength of the force when stretching or bending of the spring mainly depends on the spring stifness. So to model such a spring we can simply use Hooke's law, defined in Figure 3.1, to calculate the force on the spring.

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

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 charged particles we always assume they all have sign for the charge, in other word we always assume that all our nodes are treated as either a positive charged particle or a negative charged particle and the rationale behind that is to always make nodes repel/push each other, and we always use the same charge for all the charged particles. We can use Coulomb's law to calculate the force on a charged particle defined in Figure 3.2 below

$$F = k\frac{q1q2}{r^2}$$

where:

$F$ is the force

$k$ coulomb's constant

$q1$ charge on particle 1

$q1$ charge on particle 2

$r$ distance of separation between 2 particles

Figure 3.2: Coulomb's law

By assuming all nodes have the same exact charge in our system, where $q1 = q2$ we can simplify coulomb's law to $F = \frac{kq^2}{r^2}$, where $k$ is equal to the charge of a node. Now after

After formalising the laws that our physical simulation adhere to we can now finally derive an equation to calculate the force on any vertex $v$ in the graph, which can be defined as the sum of all edge/spring forces connected to vertex $v$ plus the sum of all repulsion force between vertex $v$ and all other vertices

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

Figure 3.3: Formula computing the force on a vertex

## 3.3 Force Resolution

In this section we will talk about force resolution. First a force can be viewed as a vector value in space with a magnitude and a direction. Simply force resolution is splitting a force into two or more components such that the effect of those force components have the same exact effect as the initial force. As we are simulating our graphs in a 2d space we are mainly concerned with converting a force into 2 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 in the $x - direction$ and the force in the $y - direction$

reference some math website talking about trigonmetry and pythagoras and force resolution

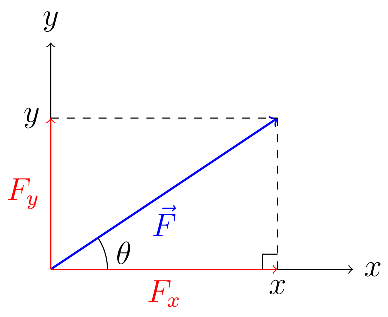


Figure 3.4: force resolution in 2d space

Based on figure 3.4 we can use Pythagoras theorem, and 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

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

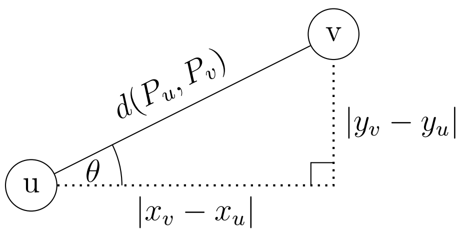
$$K_2 = kq^2 \qquad \text{(formulating coulomb's law)}$$
$$g(u,v) = \frac{K_2}{d(P_u,P_v)^2}$$

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

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

Now we would like to convert the force formula on figure 3.3 on a vertex to 2 components the x-component and the y-component, simply said we will apply force-resolution on this formula, figure 3.6 shows how the force components are computed.

add tags to this figure



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

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

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

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

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

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 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 exact steps can be applied to achieve the force component $f_y(v)$

this equation is wrong there is no abs the sign determines the direction

add tags here

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

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

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

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

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

Figure 3.7: final formulation for force formula components, $f_x(v)$ and $f_y(v)$, for a vertex

## 3.5 Spring Embedder Algorithm

That chapter was extensive, filled with a lot of math and physics content to treat a graph as a phyiscal entity. In this section, our focus will primarily be on explaining the procedure of the spring embedder algorithm. We'll make use of the formulas established in previous sections to draw the graph in a nice looking way which is the primay goal of the algorithm.

The big picture of the algorithm is to go through each node in the graph, calculate the force exerted on that node, and 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.

check the algorithm matches the equations + write some comments within the pseudocode

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

NODE-FORCE($u, G, Pos$)
 1  $Force.x = 0$
 2  $Force.y = 0$
 3  for $each\ edge\ (u,v) \in G.E$
 4      $f = K_1 \cdot (d(u,v) - L)$
 5      FORCE-RESOLUTION(U, V, F, FORCE, POS)
 6
 7  for each node $v \in G.v$
 8      if $u \neq v$
 9          $f = \frac{K_1}{d(u,v)^2}$
10          FORCE-RESOLUTION(U, V, F, FORCE, POS)

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

Figure 3.8: Spring Embedder Pseudocode

Final remarks, the graph is not in a real physical world which means we can adjust the constants, such as electric repulsion constant, used to calculate the force on a vertex to acehive the best looks of the graph drawing. Aditionally the algorithm might need an initial callibration phase until you find the best constants that yields good looking graph drawings. After the callibration phase the following constants were used in the implementation: $K_1 = 10, K_2 = 1500^2, L = 130$

# 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:

You are saying that you use paytho-gres the-orm, but you are not

1. Connecting 2 circles by a line from their surfaces
2. Edge weight following the line position and angle

## 4.1 Connecting Two Circles

In this section we will see the math used to connect 2 circles, which represent 2 nodes in the graph, by a line from their surfaces. This problem would be very easy if the lines were connecting the 2 circles from their centers as a circle based on its $(x, y)$ center coordinates. So without any further talking let's define the problem in a formal way, we can formulate the problem as following:

**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 $(x, y)$ on 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 $C_1$*

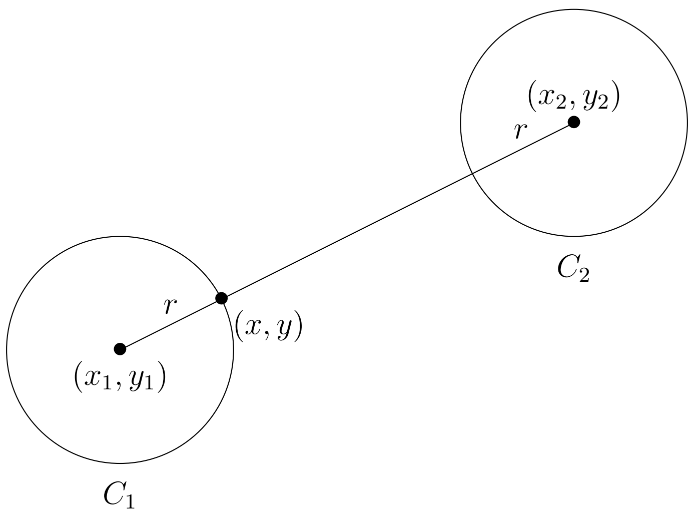
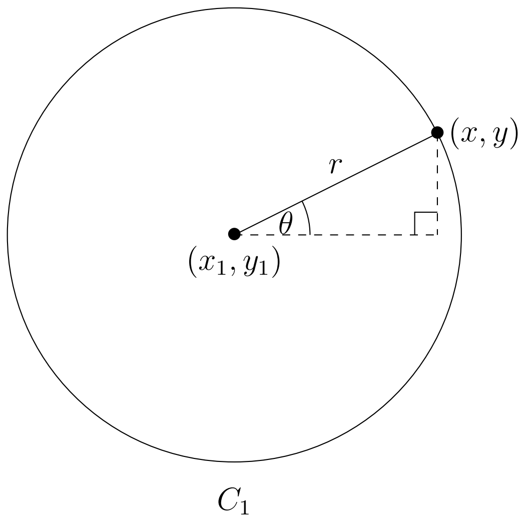


Figure 4.1: Problem 1 Illustration

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 paythogres theorem which is 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 thinking how could we find the value of $\theta$ in the equation we derived in Figure 4.2, and the solution is quite simple we will once again use paythogres theorem 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 righat angle triangle formed by those 2 points in Figure 4.1 you will notice that $\theta$ is the same exact 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 $\theta$ this mainly wraps up the math behind getting the $(x, y)$ coordinate on the surface of a circle, and simply all what you need to do after computing the *opp* and the *adj* in Figure 4.2 you just add or substract the *opp* and *adj* to 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-coordiante 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 $y = y_1 - opp$

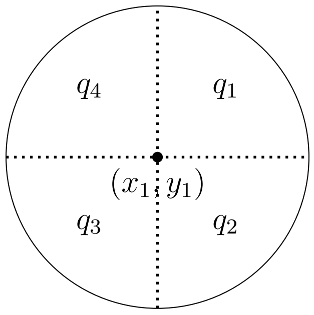


Figure 4.3: 4 Circle Quadrants

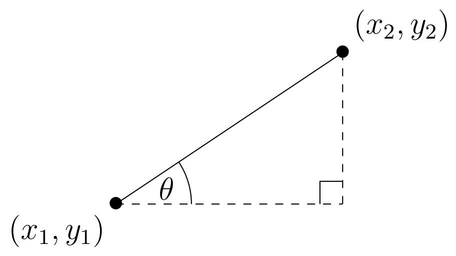
## 4.2 Edge Weight Following Line Position And Angle

In this section we will see the math used to have an edge weight follow the line position and angle no matter 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. So without any further talking let's define the problem in a formal way, we can formulate the problem as following:

**Problem 2** *Given a line defined by the points $(x_1, y_1)$ and $(x_2, y_2)$, and a rectangle where it center have the coordinate $(x_r, y_r)$. We would like to rotate and position this rectangle on the middle of the line.*

Now this problem can be divided into 2 sub-proplems, 1st sub-problem is what angle do we need to rotate the rectangle to have the same angle as the line. 2nd what is the coordinate at the line that represents its middle so we can update the rectange center coordinate $(x_r, y_r)$ to be at the mid of the line.

Let's focus on the 1st sub-problem and in this sub-problem all what 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 show 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 2nd sub-problem which is finding the mid-coordinate of the line and that can be done easily by find 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 end result after rotating the rectangle and cnetering it at the mid-coordinate of the line.

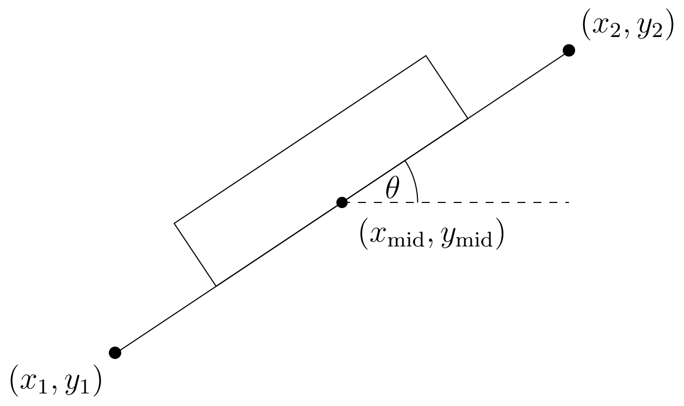


Figure 4.5: Rectangle On Line

# 5 Code Architecture & Design

In this chapter we will focus on the design aspect of the project, we will mainly discuss the following sections

ask Dr Marco how to align the item descriptions

**5.1Code Architecture** High-level description of how the software works, and the interactions happening between the different components in the project

**5.2Code Design** More detailied view of how code works, we will mainly see a UML class diagram of the software, and how are classes related to each other

**5.3Design Challenges** We will talk a dive into design challenges I faced during development that led to different design choices to overcome those challenges

remove BFS, DFS, and Dijkstra from the Architecture Img

## 5.1 Code Architecture

In this section, we will go through the Big Picture of how the code is working. We will mainly discuss the major components, and describe the role of each component and the relation of each component to other components. Also, note that the words component and module will be used interchangeably throughout this chapter

try to improve the sentence a bit

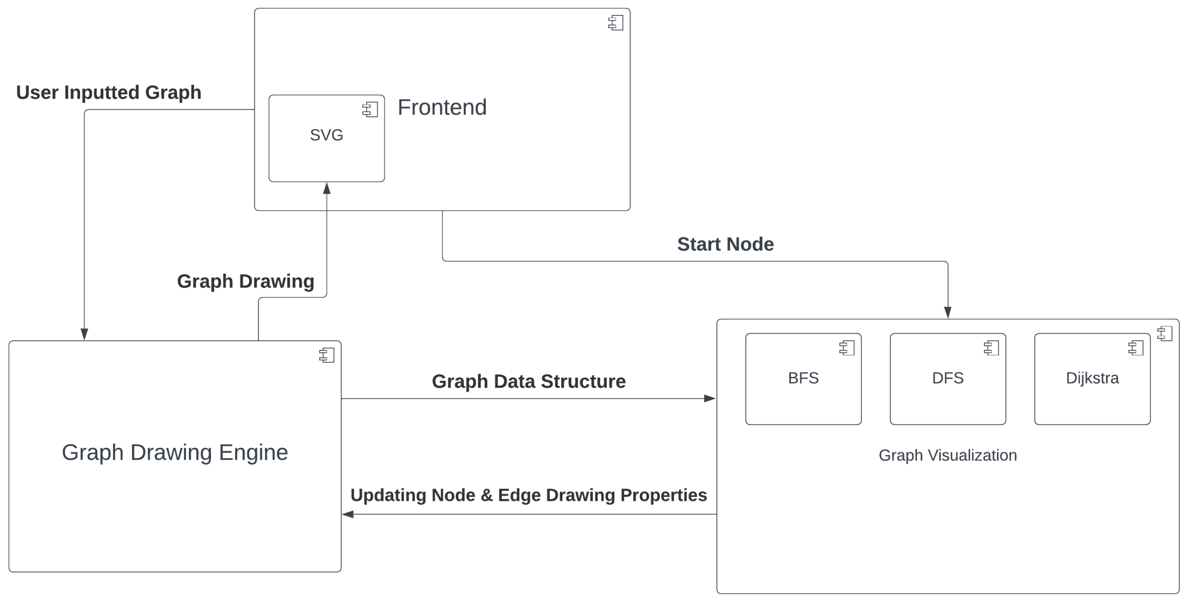


Figure 5.1: Code Architecture

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

**Frontend** The frontend is responsible for handling displaying the user interface (UI) to the user, 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 and it is 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 3 things

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 through each step of the algorithm visualization

Let's now go through the interactions made between components with each other in Figure 5.1

**Frontend** The frontend has several interactions with other components, but the main 2 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, through a user click on the graph drawing, and passing the start node to one of the available algorithm visualizers in the Graph Visualizer module

**SVG** mainly displays the graph drawing from the Graph Drawing Engine module

**Graph Drawing Engine** Receives the parsed graph user input from the frontend module, and as nodes in the graph received by the frontend are represented as text the graph drawing engine will map the graph received from the frontend to a classical graph. Draws the graph it is storing into the SVG module in the frontend and sends the classical graph data structure to the Graph Visualization module to be used as input to the different available graph algorithm

**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, and the graph visualizer will then request the classical graph data structure from the Graph Drawing Engine to be able 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.

Let's finish the chapter with some final high-level remarks to better understand how the modules work. The first thing to be aware of is that the graph drawing engine draws the graph asynchronously into the SVG in the frontend. The reason behind that is that the

maybe consider remov- ing the 2 bottom chapters and adding them to the Imple drawing can be thought of as a sequence of frames. There are small time delays between each frame to give the illusion of a real simulation of a graph drawing, and as there might be a lot of frames the process of drawing has been made to be asynchronous, as the process of drawing the graph is generally slow, to allow the remaining of the software to work normally without the need to wait for the drawing process to finish.

In the Graph-Visualizer module, we offer a range of visualization options including Breadth-First Search (BFS), Depth-First Search (DFS), and Dijkstra's algorithm. When a user selects a specific algorithm and a starting node, the frontend passes the start node to the chosen visualizer. Subsequently, the visualizer retrieves the relevant classical graph data structure from the Graph Drawing Engine module and runs the algorithm on the start node and classical graph data structure as inputs. Following this, it initiates the visualization process by changing the visual attributes of the graph within the graph drawing engine module to illustrate each step of the algorithm. This will become more clear in the code design section 5.2

That mainly concludes our discussion about Code Architecture. We have just seen a very high-level overview of how the software is structured and working and in the upcoming sections we will dive more into more details of the design of the software.

el in the UML stand for Element

## 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 and describing the classes and its relations with other classes.

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

- The UML Class diagram doesn't match exactly the code and that is for 2 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 provided as generic as possible and not be tied to a 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 confusing to understand rather than showing the high-level design of the code

- JavaScript as a language have only support for dynamic arrays/lists, so in the UML class diagram the keyword **Array** is used to show a dynamic array while the bracket symbol **[]** is used to show an array of fixed size. The same reason for such convention is once again to make the UML class diagram as generic as possible

- Object literals, and arrays containing different data types in JavaScript are referred to as Tuples in the UML Class Diagram, and this mainly to show a collection of heterogeneous values

- In the UML class diagram I have removed some classes that are not part of the core functionalities

- All class constructors have been removed from the UML Class Diagram and the main reason for was that all the class constructors would initalize the data members of the class

- 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, but it was hard to remove it because SVG is the main technology used to display the graph and the class name was very descriptive and intuitive

there is a problem

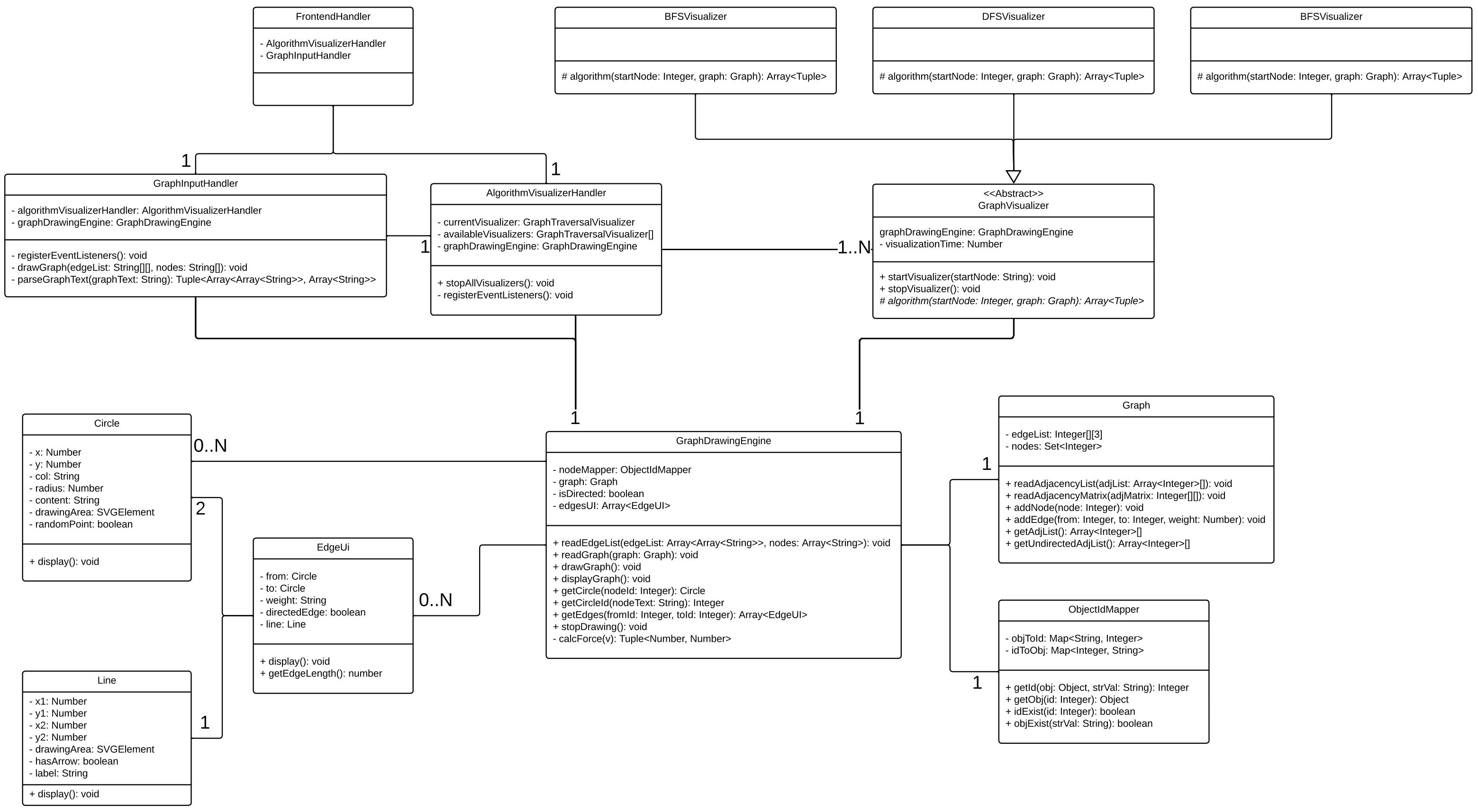

FrontendHandler
- AlgorithmVisualizerHandler
- GraphInputHandler
BFSVisualizer
# algorithm(startNode: Integer, graph: Graph): Array<Tuple>
DFSVisualizer
# algorithm(startNode: Integer, graph: Graph): Array<Tuple>
BFSVisualizer
# algorithm(startNode: Integer, graph: Graph): Array<Tuple>
1
1
GraphInputHandler
- algorithmVisualizerHandler: AlgorithmVisualizerHandler
- graphDrawingEngine: GraphDrawingEngine
- registerEventListeners(): void
- drawGraph(edgeList: String[][], nodes: String[]): void
- parseGraphText(graphText: String): Tuple<Array<Array<String>>, Array<String>>
AlgorithmVisualizerHandler
- currentVisualizer: GraphTraversalVisualizer
- availableVisualizers: GraphTraversalVisualizer[]
- graphDrawingEngine: GraphDrawingEngine
+ stopAllVisualizers(): void
- registerEventListeners(): void
1
1..N
<<Abstract>>
GraphVisualizer
graphDrawingEngine: GraphDrawingEngine
- visualizationTime: Number
+ startVisualizer(startNode: String): void
+ stopVisualizer(): void
# algorithm(startNode: Integer, graph: Graph): Array<Tuple>
1
1
Circle
- x: Number
- y: Number
- col: String
- radius: Number
- content: String
- drawingArea: SVGElement
- randomPoint: boolean
+ display(): void
0..N
2
GraphDrawingEngine
- nodeMapper: ObjectIdMapper
- graph: Graph
- isDirected: boolean
- edgesUI: Array<EdgeUI>
+ readEdgeList(edgeList: Array<Array<String>>, nodes: Array<String>): void
+ readGraph(graph: Graph): void
+ drawGraph(): void
+ displayGraph(): void
+ getCircle(nodeId: Integer): Circle
+ getCircleId(nodeText: String): Integer
+ getEdges(fromId: Integer, toId: Integer): Array<EdgeUI>
+ stopDrawing(): void
- calcForce(v): Tuple<Number, Number>
Graph
- edgeList: Integer[][3]
- nodes: Set<Integer>
+ readAdjacencyList(adjList: Array<Integer>[]): void
+ readAdjacencyMatrix(adjMatrix: Integer[][]): void
+ addNode(node: Integer): void
+ addEdge(from: Integer, to: Integer, weight: Number): void
+ getAdjList(): Array<Integer>[]
+ getUndirectedAdjList(): Array<Integer>[]
1
EdgeUi
- from: Circle
- to: Circle
- weight: String
- directedEdge: boolean
- line: Line
+ display(): void
+ getEdgeLength(): number
0..N
ObjectIdMapper
- objToId: Map<String, Integer>
- idToObj: Map<Integer, String>
+ getId(obj: Object, strVal: String): Integer
+ getObj(id: Integer): Object
+ idExist(id: Integer): boolean
+ objExist(strVal: String): boolean
1
Line
- x1: Number
- y1: Number
- x2: Number
- y2: Number
- drawingArea: SVGElement
- hasArrow: boolean
- label: String
+ display(): void
1

Figure 5.2: UML Class Diagram

Now let's go through explaining each class

**Circle** Class representing a circle containing text. It mainly have the *display()* method which displays the circle into the *drawingArea* which is the SVG container displayed in the frontend to the user

**Line** Class representing a line with a label. The line could also optionally have an arrow at its end. It mainly have the *display()* method to dislay the line into the *drawingArea*

**EdgeUi** Class responsible for displaying a line between 2 circles to simulate an edge in a graph. The edge also have a weight, and can be set to be directed or undirected

**Graph** This class encapsulates a classical graph data structure, interally 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 matrix

**ObjectIdMapper** Class that maps an object to an integer ID. *getId(obj)* method responsible for returning the ID associated with the object or creating a new ID if the object doesn't have a mapping. *getObj(id)* method responsible for returning the object mapped to the id

**GraphDrawingEngine** The GraphDrawingEngine is the class 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 rperesented as strings, and this method maps the graph received from the frontend to a classical graph and creates a circle instance for each node and creates an EdgeUI instance for each edge
- *drawGraph()* this method is the one responsible for drawing the graph into the SVG container in the frontend by running the Spring Embedder algorithm
- *displayGraph()* this method is responsbile for refreshing the SVG container and displaying all the nodes and edges in the graph
- *getCircle(nodeId)* returns the circle instance mapped to the node ID
- *getCircleId(nodeText)* returns the node ID mapped with the 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. The graph has mainly 3 method

- *startVisualizer(startNode)* which receives the start node and starts the visualization process and during the visualization process it directly make updates to the visual attributes to the nodes and edges stored inside the GraphDrawingEngine
- *stopVisualizer()* stops the running of the current visualizer
- *algorithm(startNode, graph)* an abstract method that receives the start node and the classical graph and returns an array of the steps takes during the algorithm execution

**BFSVisualizer** Extends the GraphVisualizer class. The class runs the bfs algorithms by overriding the *algorithm(startNode, graph)* method

**DFSVisualizer** Extends the GraphVisualizer class. The class runs the dfs algorithms by overriding the *algorithm(startNode, graph)* method

**DijkstraVisualizer** Extends the GraphVisualizer class. The class runs dijkstra's algorithms by overriding the *algorithm(startNode, graph)* method

**GraphInputHandler** Class that 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** Class that handles inputting the start node and the chosen algorithm visualizer from the user interface, the start node will then be passed to 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 seen that the *GraphInputHandler* has has a single instance of the *AlgorithVisualizerHandler* and the main reason was to stop any visualizer that are running once the user inputs a new graph as any old graph that have been drawn will be replace with the new graphs inputted by the user.

Second you can see that both the *GraphInputHanlder* and *AlgorithVisualizerHandler* have an instance of the *GraphDrawingEngine* class and the reason is that the *GraphInputHandler* passes the graph input from the user to the *GraphDrawingEngine* to display the graph, and the *AlgorithVisualizerHandler* instantiates all the visualizers and passes the *GraphDrawingEngine* to each visualizer instance.

Third 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 so although 2 objects might be equal in content but have different addresses and not be equal, and for that reason we are using a string value to represent an object instead.

Fourth you might have thought why are we using a classical graph data structure inside the *GraphDrawingEngine*, you might say we can explicitly represent the graph as a set of *Circle* instances representing the graph nodes and as a set of *EdgeUi* instances representing the graph edges, although you are right this can be done and it will have a direct mapping to the graph drawn in the SVG container in the frontend, but I have took the decision to map the *Circle* instances to numeric IDs representing the nodes, and the *EdgeUi* instances as edges between the *Circle* IDs. This decision was taking very early during implementation and the rationale behind that is to decopule the graph drawn in the frontend from the graph used by all the graph algorithms like BFS, DFS, Dijkstra and even the Spring-Embedder algorithm. As you will find that most of the graph algorithms run on the classical graph data structure, although all those algorithms can be modified to run with our user-defined graph structure, but I wanted to provided something standard to be used by any graph algorithm without need to have knowledge about my user-defined graph data structure used for drawing in the frontend.

still some points in the below comment

## 5.3 Design Challenges

In this section we will mainly talk about design challenges arising during implementation and how they were overcome to lead to better designs. Although most of the design decision disucssed in this section will be reflected in the Code Design Section 5.2, but I wanted to show the rationle and reason for achieving certain designs 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. Given that the SVG Container is part of the HTML displayed on the website we rely on the DOM API provided by the browser to manipulated the html elements from the JavaScript, and the fundamental problem here is that we don't want the *GraphDrawingEngine* class to be coupled tighly with any certain technology used by the fronten simply stated we don't want the *GraphDrawingEngine* to rely on the DOM API to perofrm the drawing in the frontend. And the solution for that problem was to abstract the interaction with the SVG Container through high-level classes that wrap the usage of the DOM API, and those classes are the *Circle* and *Line*, those classes are abstract enough to be used by the *GraphDrawingEngine* and internally those classes use the DOM API, and are responsbiel for creating their corresponding SVG Elements and adding those elements to the SVG Container in the frontend, and all what the *GraphDrawingEngine* needs to do is 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 the algorithm visualization process. 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 the 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, I would need to create a method that includes:

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's 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, the method running the graph algorithm would still need to be concurrent. It would have to pause at each step to simulate animation, and even if the sleeping logic is implemented separately, it would need to wait until this method finishes before continuing with the algorithm. This still results in a complex method that handles multiple tasks.

The solution is to run the complete graph algorithm without any visualization or redrawing. The method implementing the graph algorithm will return an array containing all the visualization steps. Another method will be responsible for going through each step in the array and performing all the tasks described above.

Thus, when a new graph algorithm visualizer needs to be created, it simply has to implement the regular graph algorithm, include the specific points to be visualized, and return an array containing all the steps after the algorithm execution. This also allows for extended functionality like stepping forward and backward in the visualization process itself.

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

The *startVisualizer(startNode)* method inside the *GraphVisualizer* calls the abstract method *algorithm(startNode, graph)* and is responsible for all the common tasks described above by going through each step in the array returned by the specific visualizer. This is known as the Template method design pattern. I will not describe how the each step looks like as that is an implementation detail, but if you are curious of how that is implemented you can have a look at the **traversals.mjs** file which contains all the code logic for the visualizers.

include the below last challenge if you think it is worth including, discuss with Dr. Marco

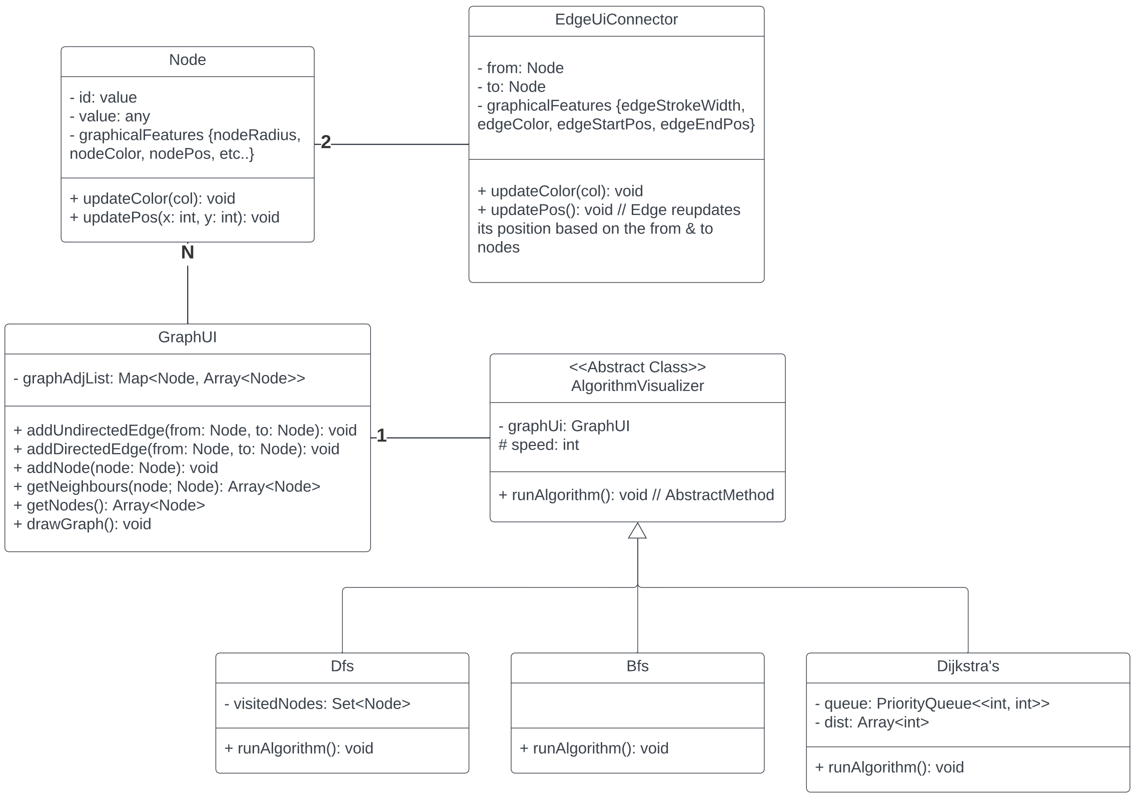
include the below section about design failures if you think it is worth including, dicuss with Dr. Marco

# Bibliography

# A Original Project Proposal

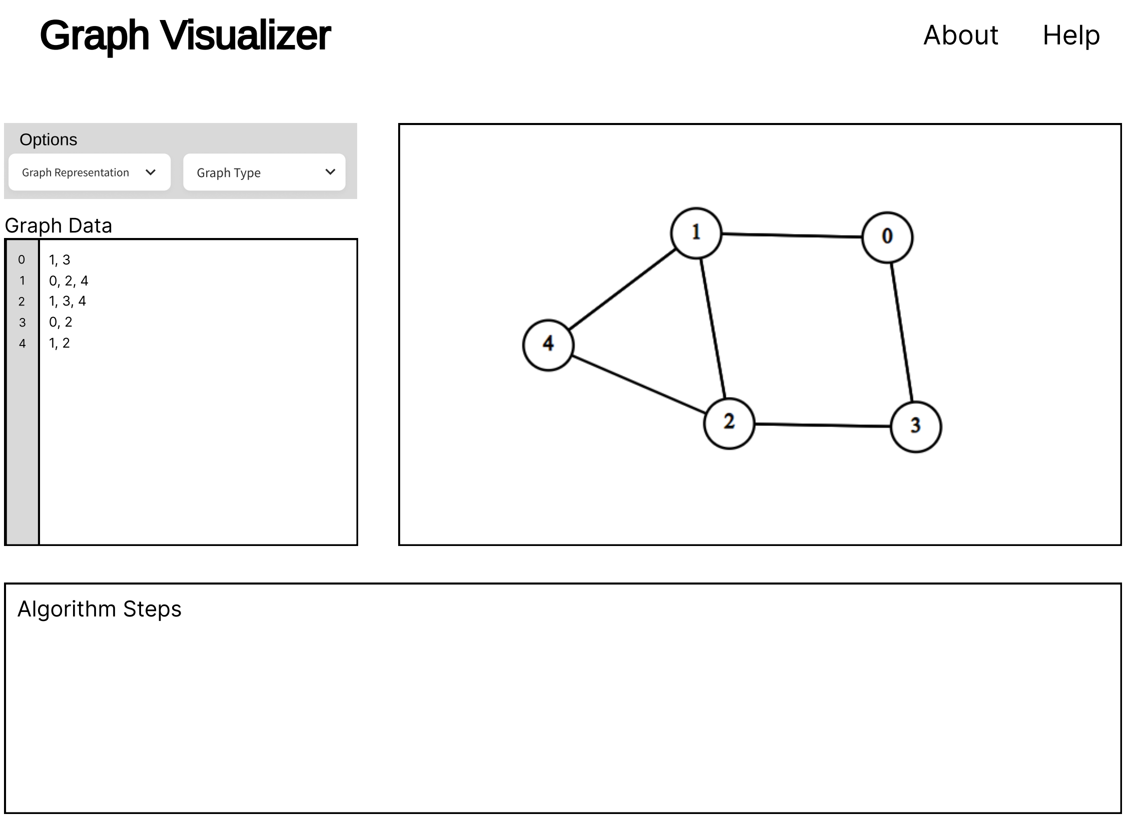
Put a copy of your proposal here

# B Another Appendix Chapter



Node
- id: value
- value: any
- graphicalFeatures {nodeRadius, nodeColor, nodePos, etc..}
+ updateColor(col): void
+ updatePos(x: int, y: int): void
2
N
EdgeUiConnector
- from: Node
- to: Node
- graphicalFeatures {edgeStrokeWidth, edgeColor, edgeStartPos, edgeEndPos}
+ updateColor(col): void
+ updatePos(): void // Edge reupdates its position based on the from & to nodes
GraphUI
- graphAdjList: Map<Node, Array<Node>>
+ addUndirectedEdge(from: Node, to: Node): void
+ addDirectedEdge(from: Node, to: Node): void
+ addNode(node: Node): void
+ getNeighbours(node; Node): Array<Node>
+ getNodes(): Array<Node>
+ drawGraph(): void
1
<<Abstract Class>>
AlgorithmVisualizer
- graphUi: GraphUI
# speed: int
+ runAlgorithm(): void // AbstractMethod
Dfs
- visitedNodes: Set<Node>
+ runAlgorithm(): void
Bfs
+ runAlgorithm(): void
Dijkstra's
- queue: PriorityQueue<<int, int>>
- dist: Array<int>
+ runAlgorithm(): void

Figure B.1: Initial UML Class Diagram



Graph Visualizer

About Help

Options

Graph Representation: Graph Type

Graph Data

```
0   1, 3
1   0, 2, 4
2   1, 3, 4
3   0, 2
4   1, 2
```

Algorithm Steps

Figure B.2: UI Wireframe