



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF ALGORITHMS AND APPLICATIONS

Visualization of the path-finding algorithms on graphs

Supervisor:

Maloschikné Harrach Nóra

Assistant professor

Author:

Mykyev Atabek

Computer Science BSc

Budapest, 2024

Thesis Registration Form

Student's Data:

Student's Name: Mykyev Atabek
Student's Neptun code: FYQWAV

Course Data:

Student's Major: Computer Science BSc
I have an internal supervisor

Internal Supervisor's Name: *Maloschikné Harrach Nóra*

Supervisor's Home Institution: **Department of Algorithms and Applications**
Address of Supervisor's Home Institution: **1117, Budapest, Pázmány Péter sétány 1/C.**
Supervisor's Position and Degree: *Assistant professor, PhD in Mathematics*

Thesis Title: Visualization of the path-finding algorithms on graphs.

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis of your planned thesis.)

In this thesis, the primary objective is to provide a comprehensive understanding of how diverse path-finding algorithms operate on graphs, unraveling the details of each step in their execution.

The purpose of this project extends beyond mere algorithmic exploration - it serves as an educational tool. Through detailed visualizations, the thesis aims to show the inner workings of various path-finding algorithms. Each algorithm is dissected, and its functionality is expounded upon, fostering a nuanced comprehension of the algorithmic processes involved.

Furthermore, the research extends to recommending optimal algorithms such fit to specific graph characteristics. These suggestions are not arbitrary but are grounded in a diligent analysis of factors such as graph type, density, and other pertinent algorithm-related considerations. By offering nuanced insights into algorithm selection, the thesis aims to empower users to make informed decisions based on the unique attributes of their graphs.

In essence, this thesis strives to be a comprehensive resource, shedding light on the dynamic interplay between path-finding algorithms and graphs while simultaneously serving as an instructive guide for those navigating the complicated landscape of algorithmic decision-making.

Budapest, 2023. 11. 30.

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Thesis goal	4
2	User Documentation	5
2.1	Graphs	5
2.1.1	Undirected graphs	5
2.1.2	Directed graphs	6
2.2	Algorithms	7
2.2.1	Breadth-first search	7
2.2.2	Depth-first search	7
2.2.3	Dijkstra	8
2.2.4	The Bellman-Ford	9
2.3	Installing the tool	9
2.3.1	Required software	10
2.4	Manual on the tool	11
2.4.1	Main pages	11
2.4.2	Adding node	12
2.4.3	Removing node	13
2.4.4	Adding edge	14
2.4.5	Removing edge	15
2.4.6	Setting the weight for the edge	16
2.4.7	Choosing from prepared graphs	18
2.4.8	Canceling action	19
2.4.9	Choosing the algorithm	19
2.4.10	Running the algorithm	20
2.4.11	Directed graph page	23

3	Developer Documentation	24
3.1	Architecture	24
3.1.1	Use cases	25
3.1.2	Component diagram	26
3.1.3	Class diagram	27
3.1.4	Graph representation	28
3.2	Technical details	30
3.2.1	Libraries	30
3.3	Implementations	33
3.3.1	API endpoints	33
3.3.2	Graph management	33
3.3.3	Algorithm management	43
3.4	Testing	46
3.4.1	Tests structure	48
3.4.2	Tests implementation	49
3.4.3	Results	53
4	Conclusion	55
	Bibliography	56
	List of Figures	58
	List of Codes	60

Chapter 1

Introduction

1.1 Motivation

In the vast and developing world of algorithms and data structures, one could find quite a lot of different topics.

The Graphs could be considered as one of the most complex of them all! You might find it challenging when you first encounter the concept and theory behind them. “Graphs are one of the unifying themes of computer science—an abstract representation that describes the organization of transportation systems, human interactions, and telecommunication networks. That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer. More precisely, a graph $G=(V,E)$ consists of a set of vertices V together with a set E of vertex pairs or edges.” [1].

We truly grow, when we face the challenges and seize the opportunity, no matter how hard it may seem to achieve! So the true motivation behind this very thesis is to dive into the difficult world of graphs myself and to guide others by showing how exactly certain things work with graphs. And I do such, by visualizing the graph and the algorithm being applied to the graph. It is always great to have a tool by your hand, which could show you how exactly certain parts might look like in practice when you learn the theory.

1.2 Thesis goal

So abstract yet, so practical in the real world! Different applications of graph data structure is wide-spread around the world: in networking schema, circuits, maps and etc. With all of the aforementioned applications, we might encounter many kinds of challenges and obstacles, one of which, is what my whole thesis is about - Path Finding. Let's say you have a certain node A and the other node B, having that all of the nodes in graph are connected to each other with edges, we would like to find a path from our node A to node B through those connections.

That problem carries a name - Pathfinding or as Jon and Éva call it in their book, s-t connectivity. "Suppose we are given a graph $G=(V,E)$ and two particular nodes s and t. We'd like to find an efficient algorithm that answers the question: Is there a path from s to t in G? We will call this the problem of determining s-t connectivity." [2]

Something so simple and intuitive to the human being is not an easy goal for the program to achieve. Traversal of the elementary unweighted graph would not be of trouble, but what if you have a massive network consisting of thousands nodes and edges, where every edge has some weight. A trivial traversal through all of the possible paths would be terribly ineffective in this case. Pathfinding algorithms reach the goal in the most efficient ways, so it is crucial to know them for everyone who is going to deep dive into the world of graphs.

The main goal of the thesis is to help people understand the ways algorithms traverse through the graph by visualizing them. Handing to the learner the tool which might help them to better understand what really happens when the algorithm is being executed.

Chapter 2

User Documentation

In the following chapter, one could find everything he needs in order to start using Visualizing Tool. Firstly, the theory required to fully grasp all of the concepts is elaborated. Then, directions on the tool installation are provided. In conclusion, all of the parts regarding the usage of the application are explained in details.

2.1 Graphs

Reciting the aforementioned definition - “*Graphs* are one of the unifying themes of computer science—an abstract representation that describes the organization of transportation systems, human interactions, and telecommunication networks. That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer. More precisely, a graph $G=(V,E)$ consists of a set of vertices V together with a set E of vertex pairs or edges.” [1]. There are also two types of graphs that are worth mentioning.

2.1.1 Undirected graphs

Undirected graphs consist of the edges which are all bidirectional. Implying that if you can get from node A to node B, you will definitely be able to get from node B to node A, using the same edge.

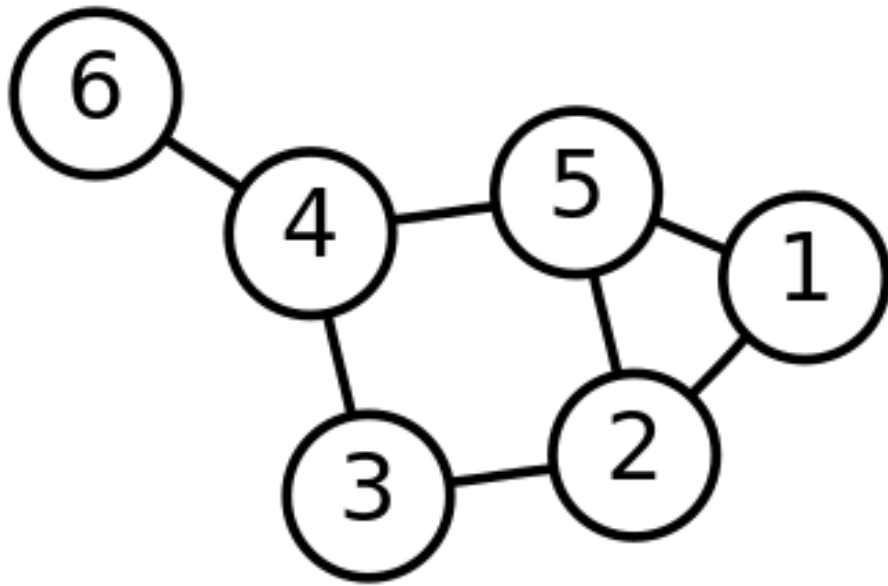


Figure 2.1: Undirected Graph

2.1.2 Directed graphs

In case of directed graphs, the edges have certain direction that is represented by the pointing arrow. The arrow starting at the node A, pointing at the node B, implies that you might get from A to B using the edge, but in case of directed graphs, it does not imply, that you can get from B to A.

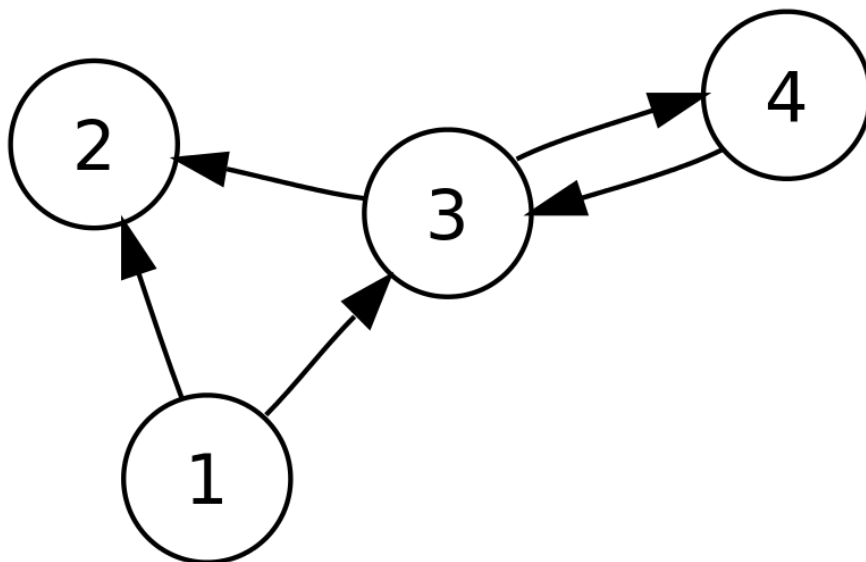


Figure 2.2: Directed Graph

2.2 Algorithms

There are four algorithms which you are able to execute on the graphs in the tool. Such are: Breadth-first search, Depth-first search, Dijkstra and The Bellman-Ford algorithms.

2.2.1 Breadth-first search

Breadth-first search or simply BFS, is an elementary algorithm that could be used for the s-t connectivity problems. This algorithm is perfectly explained in the Cormen's book - Introduction to Algorithms.

“Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning- tree algorithm and Dijkstra's single-source shortest-paths algorithm use ideas similar to those in breadth-first search. Given a graph $G=(V,E)$ and a distinguished source vertex s , breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s . It computes the distance from s to each reachable vertex, where the distance to a vertex v equals the smallest number of edges needed to go from s to v . Breadth-first search also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s , the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G , that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.“ [3]

The property of the graph which you might want to consider while executing the algorithm on the graph is whether it's weighted or not. In case, the graph is weighted it simply does not take it into account and only gives the path with minimal number of edges from A to B.

2.2.2 Depth-first search

Depth-first search or DFS, is also an elementary algorithm which also used for the pathfinding problems. Once again, Cormen explained the algorithm in well formed terms in his already mentioned book, Introduction to Algorithms.

“As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered

vertex v that still has unexplored edges leaving it. Once all of v 's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search." [3]

The same concept applies to DFS as it does to BFS, it gives an optimal solution in case of unweighted graphs. Worth mentioning that it also does not give out the "shortest" path but the first lexicographical which means that if you have paths $1 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 4 \rightarrow 3$ it will give out the first one as the resulting path, as 2 is less than 4.

2.2.3 Dijkstra

Dijkstra's algorithm for finding the shortest paths in the weighted graph is one of the most favored in its field of purpose. It is plain and efficient, so it is widely used in different applicable areas. Skiena clearly elaborated definition of the algorithm in his book 'The Algorithm Design Manual'.

"Dijkstra's algorithm is the method of choice for finding shortest paths in an edge and/or vertex-weighted graph. Given a particular start vertex s , it finds the shortest path from s to every other vertex in the graph, including your desired destination t . Suppose the shortest path from s to t in graph G passes through a particular intermediate vertex x . Clearly, this path must contain the shortest path from s to x as its prefix, because if not, we could shorten our s -to- t path by using the shorter s -to- x prefix. Thus, we must compute the shortest path from s to x before we find the path from s to t . Dijkstra's algorithm proceeds in a series of rounds, where each round establishes the shortest path from s to some new vertex. Specifically, x is the vertex that minimizes $\text{dist}(s, v_i) + w(v_i, x)$ over all unfinished $1 \leq i \leq n$, where $w(i, j)$ is the length of the edge from i to j , and $\text{dist}(i, j)$ is the length of the shortest path between them. This suggests a dynamic programming-like strategy. The shortest path from s to itself is trivial unless there are negative weight edges, so $\text{dist}(s, s) = 0$. If (s, y) is the lightest edge incident to s , then this implies that $\text{dist}(s, y) = w(s, y)$. Once we determine the shortest path to a node x , we check all the outgoing edges of x to see whether there is a better path from s to some unknown vertex through x ." [1]

The Dijkstra algorithm works for non-negative weighted graphs, because as soon

as it meets a negative edge it is stuck in the infinite loop. For the sake of user experience, all of the negative edges, in case of using Dijkstra, are treated as 0-weighted edges. In that instance, it won't find the optimal solution but it will not stay in the loop forever either, so that is an understandable sacrifice. In case of the unweighted graphs, Dijkstra finds a solution, but is not necessarily the best in this case, as you might achieve the same result by using easier algorithm such as BFS.

2.2.4 The Bellman-Ford

The Bellman-Ford algorithm is very crucial in cases when you need to find the path in the weighted graph with negative values. Though, it is more complex and resource consuming, the broader set of graphs could be resolved with the algorithm.

By Cormen, "it solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G=(V,E)$ with source vertex s and weight function $w : E \rightarrow \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights. The procedure BELLMAN-FORD relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s,v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source." [3]

The algorithm is a great choice in cases when you have negative weights in the graph, but whenever there is a graph with no negative weights, The Bellman-Ford loses a lot due to it being executed on every edge of the graph and repeating the step until no further relaxations are possible.

2.3 Installing the tool

The tool might be used by installing some software and running couple of commands in order to start it on your local machine. It is a web application, though self-hosted app would not need any connection to the web.

2.3.1 Required software

There are two parts playing role in the application. Front end, the client-side application which plays role in everything related to view of the program and Back end, the server-side application which computes every operation that is being requested from the client.

For the back, you would need to follow the next steps:

- 1 Install Python 3.12 [4]
- (Optional) Install PyCharm [5]
- 2 Go into the directory - app
- 3 Run the command for the installation of the packages

```
1 pip install -rrequirements.txt
```

- 4 Go back to the previous directory

- 5.1 After that you can run the application using the next command

```
1 python app/main.py
```

If you are having an error, you might want to check whether your current PYTHONPATH environmental variable is set appropriately, to the directory of the whole project. There is an example of its usage in the github ci workflow file.

- 5.2 In case of using PyCharm, go to the main.py file and execute the main by clicking the run button.

For the front, you would need to follow the next steps:

- 1 Install Node.js [6] (v20.x.x LTS)
- 2 Go into the directory - app-view
- 3 Run the command for the installation of the packages

```
1 npm install
```

- 4 After that you can run the application using the next command

```
1 npm start
```

5 The window will open in your default browser, if it doesn't then open the website on the URL - `http://localhost:3000/`.

2.4 Manual on the tool

There are two main capabilities of the tool that this manual will mostly be about: graph and algorithm control. So, let's cover those parts in details for the sake of understanding on how to use the tool.

2.4.1 Main pages

The tool meets you with a nice welcome page and a navigation bar. The learner has such options as undirected and directed graphs.

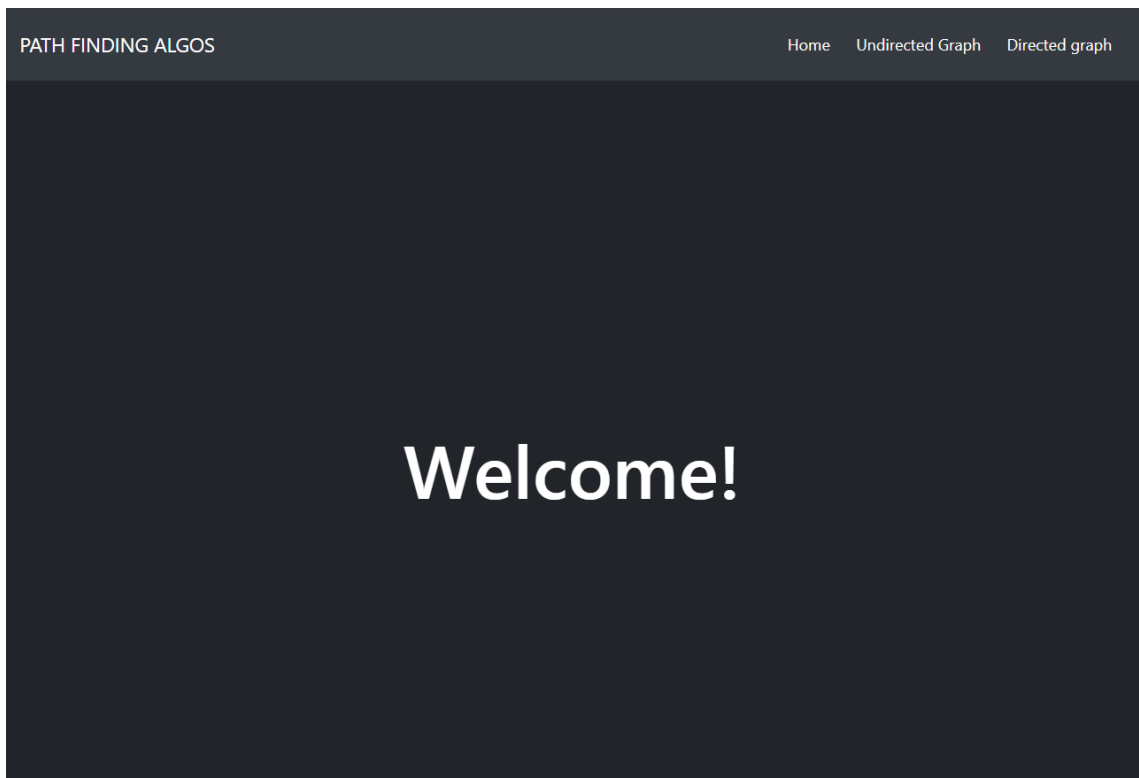


Figure 2.3: Welcome Page

On the figure 2.4 one might see the empty graph page. The initial states of the undirected and directed pages are indistinguishable as long as the user won't create a

graph. Further on the manual, the undirected graph page will be used to demonstrate the functionality of the tool. Division into five main components could be derived from the page. These are communication panel, graph canvas, algorithm description panel, graph control panel and algorithm control panel. The communication, canvas and description panels are separated by gray borders and the remaining panels are located below them.

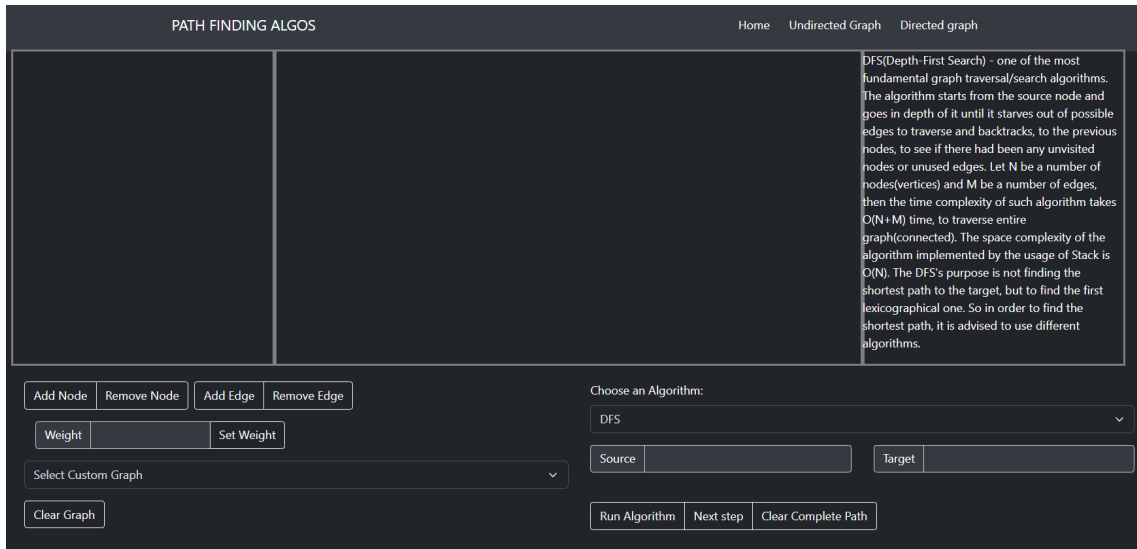


Figure 2.4: Empty Graph Page

2.4.2 Adding node

The node addition does not require anything but the click of a single button on the graph control panel side of the page. "Add Node" button is used to create a new node on the page. New node is going to pop out in the random location on the canvas. And a message in the communication panel will appear saying "Added node!" which indicates that the node has been added to the graph successfully.

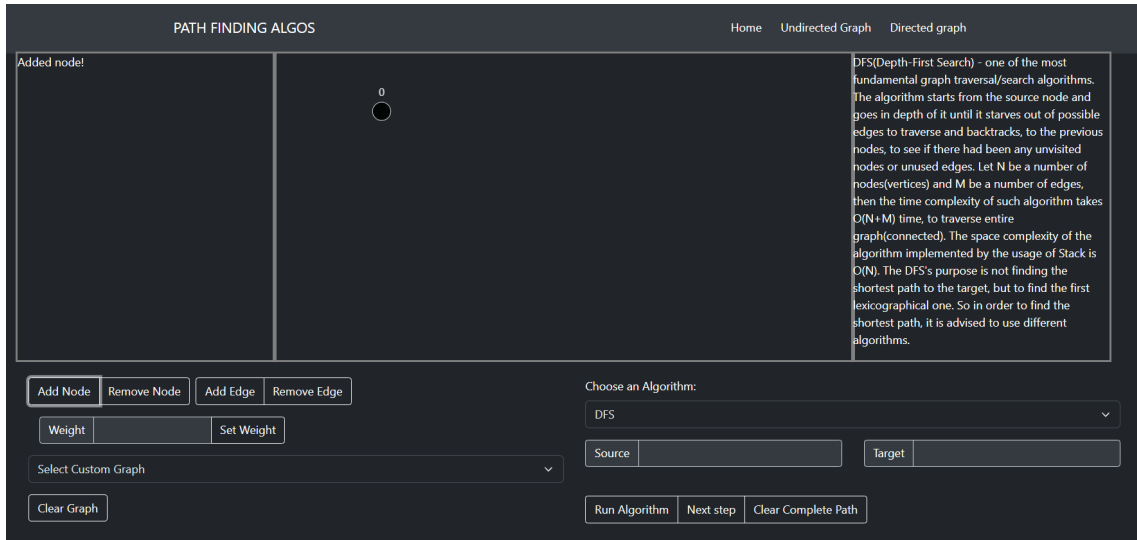


Figure 2.5: Adding node to graph

2.4.3 Removing node

In order to remove the node, you must have at least one node in the graph. "Remove node" button in the graph control panel let's us to choose any node in the graph that we would like to delete.

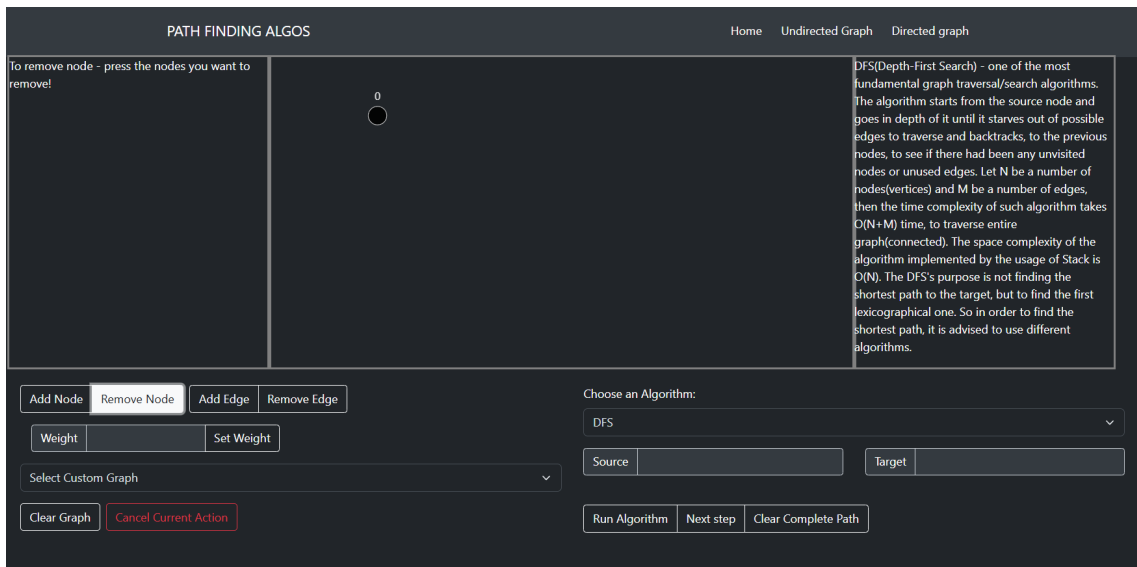


Figure 2.6: Removing node from graph

After pressing such button and clicking on any node of the graph, the chosen node will be removed and the relevant message will appear. All of the nodes, which had the bigger index will be decreased by one and in case there is at least one node with the larger index in the graph, it will take the place of the removed one after

deletion. Such behavior is caused by the limitation of library used for drawing of graph on the screen.

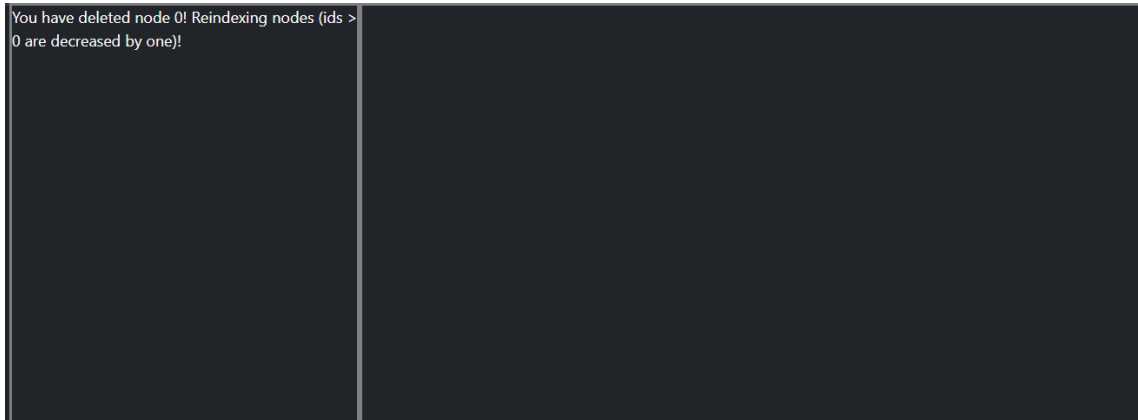


Figure 2.7: Removed node from graph

2.4.4 Adding edge

The addition of the edge is achieved by the click of the button "Add Edge" and further steps of choosing the nodes between which the edge shall appear.

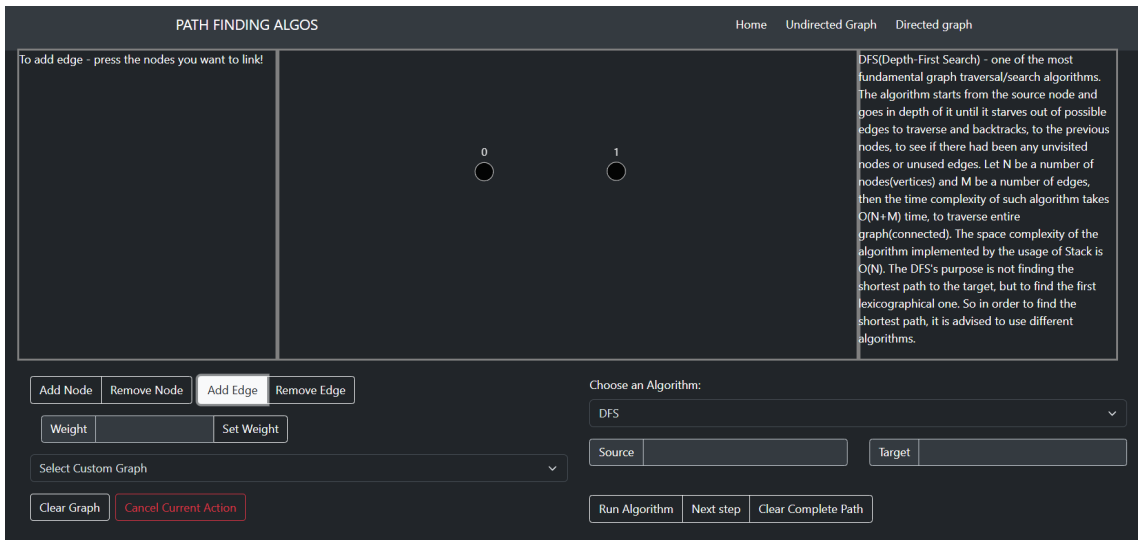


Figure 2.8: Adding edge to graph

After you have clicked on the first node you will see the message appear, indicating that you have successfully chose out-node, the node from which edge will lead into the other node(in case of directed graphs).



Figure 2.9: Selecting first node

Finally, if you have chosen a proper node to be the in-node, then the edge will be created between them. Additionally, The message will indicate the success of the operation.

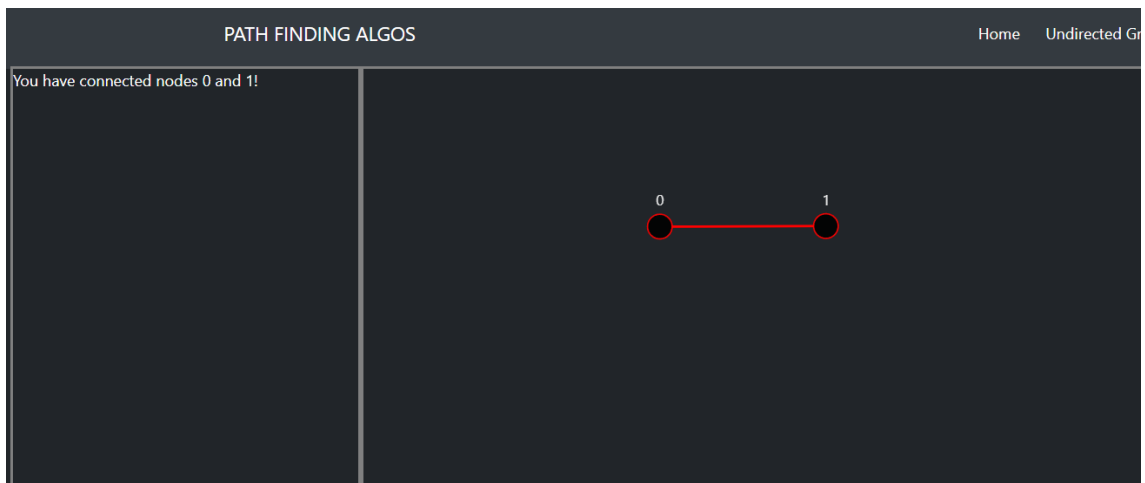


Figure 2.10: Selecting second node

2.4.5 Removing edge

Removal of the edge appears to be quite similar to the operation of node removal. One must click on the "Remove edge" button to see that the communication panel is telling them to choose any edge they want to delete.

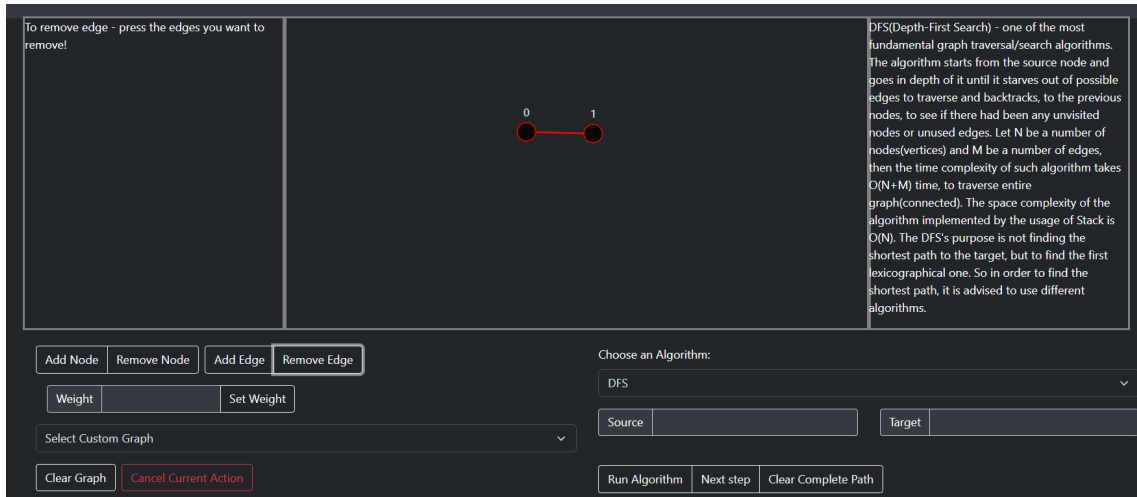


Figure 2.11: Removing edge from graph

By following the instructions of the communication panel seen on the figure 2.11, the successful removal of the chosen edge is achieved. Also, worth mentioning, that the removal of the edge may be executed by the removal of the node as well, in case the deleted node has any incoming or outgoing edges.

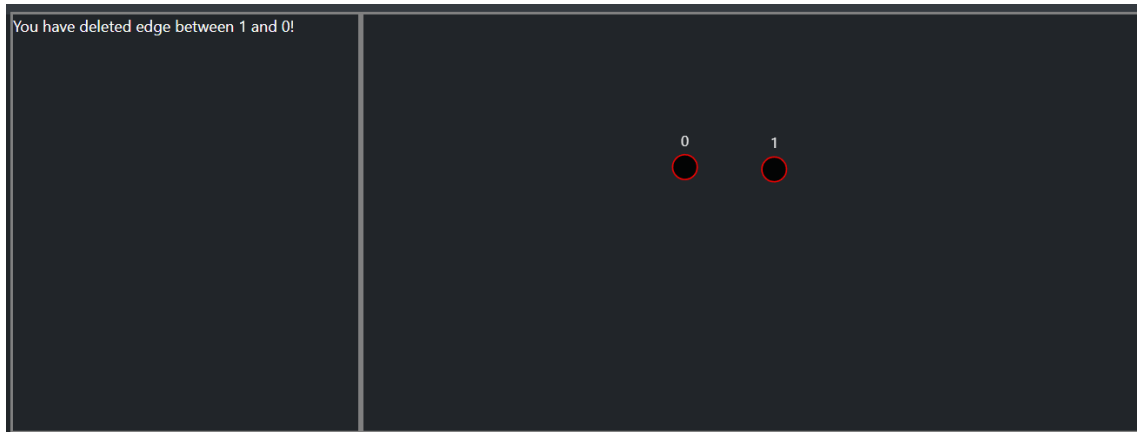
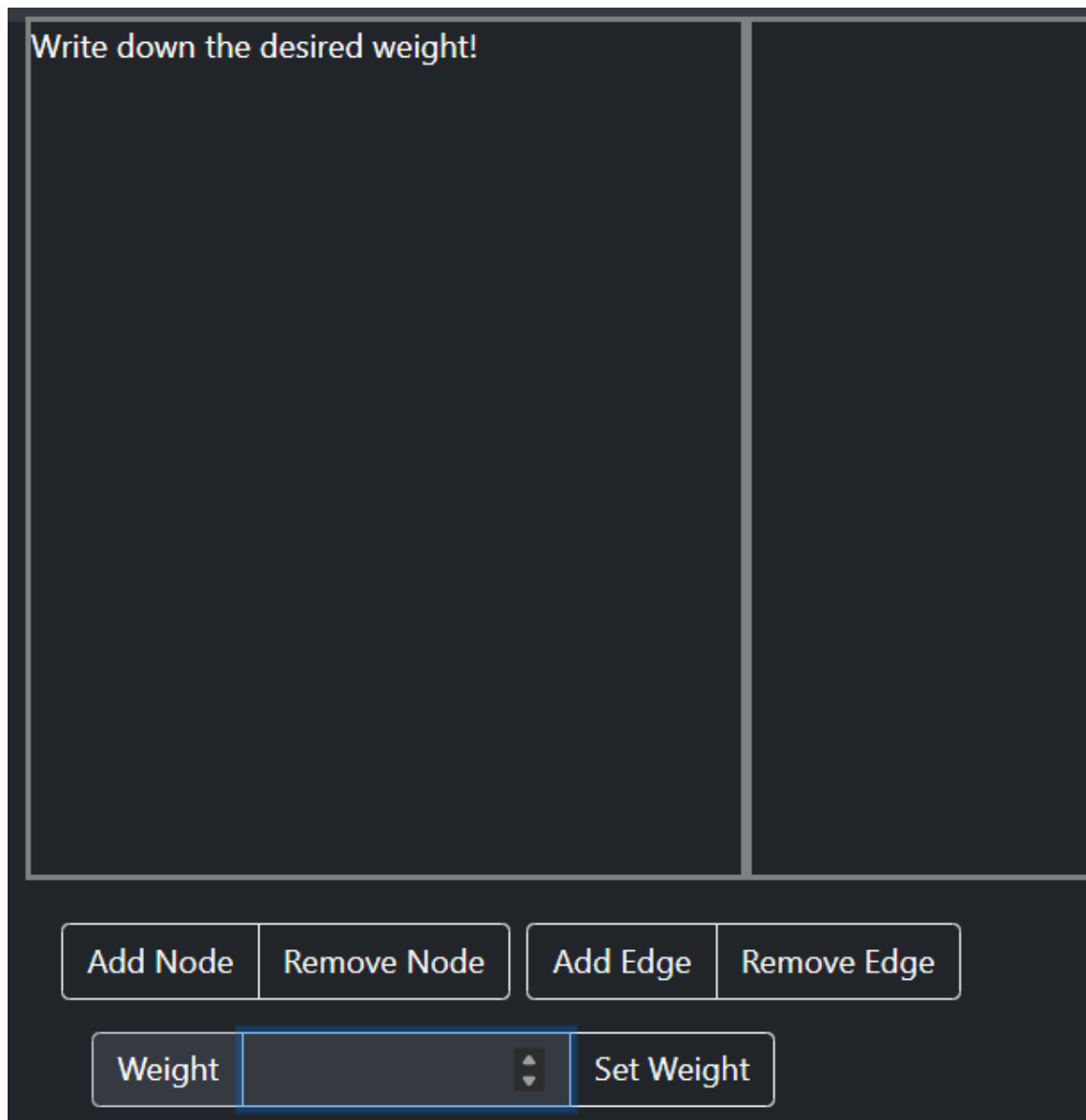


Figure 2.12: Removed edge from graph

2.4.6 Setting the weight for the edge

The weighted graphs are needed, if the learner wants to see the execution of such algorithms as 'Dijkstra' and 'The Bellman-Ford' in environment they are the best fit for. So, this option is available, which user might notice on the graph control panel. First of all, you need to specify the weight that you want your edge to have. This is achieved by clicking on the weight numerical input bar. One might input their desired weight by using the arrows which increase or decrease the number or write it down using keyboard.



Write down the desired weight!

Add Node Remove Node Add Edge Remove Edge


Weight  Set Weight

Figure 2.13: Writing down the weight number

Secondly, user needs to click on the "Set Weight" button. After that, the message will inform that you are free to choose from the edges on a graph to set the weight for it.

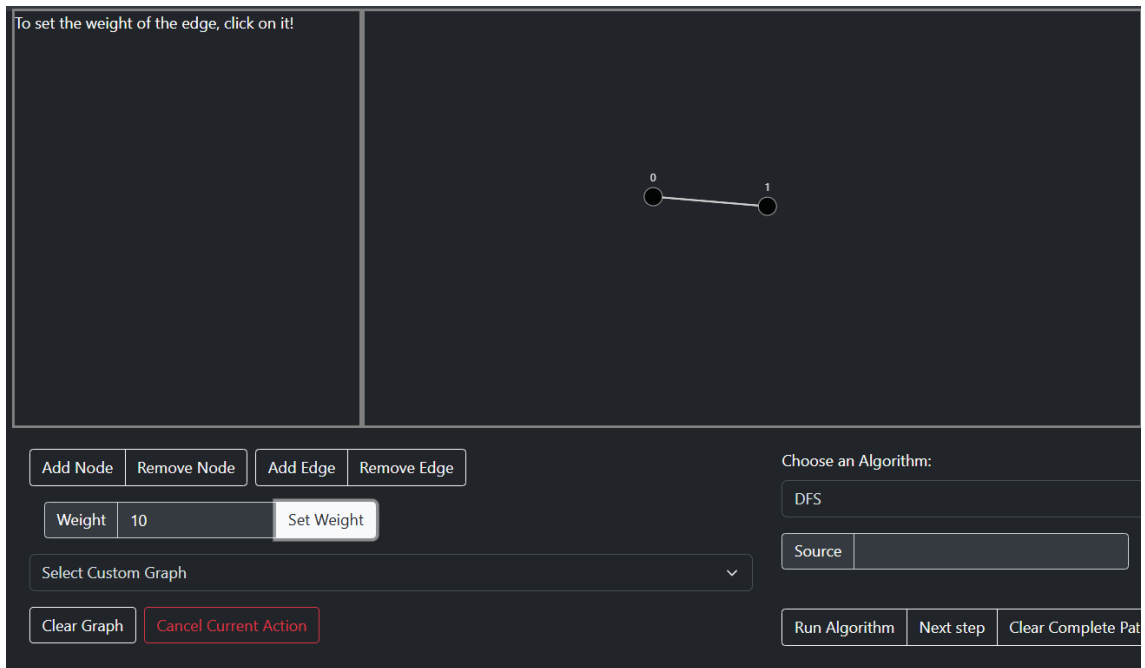


Figure 2.14: Setting the weight for an edge

After completion of the previous instruction, the weight appears on the chosen edge. For the sake of clarity, communication panel shows the node indices between which the edge has been chosen to gain a weight.

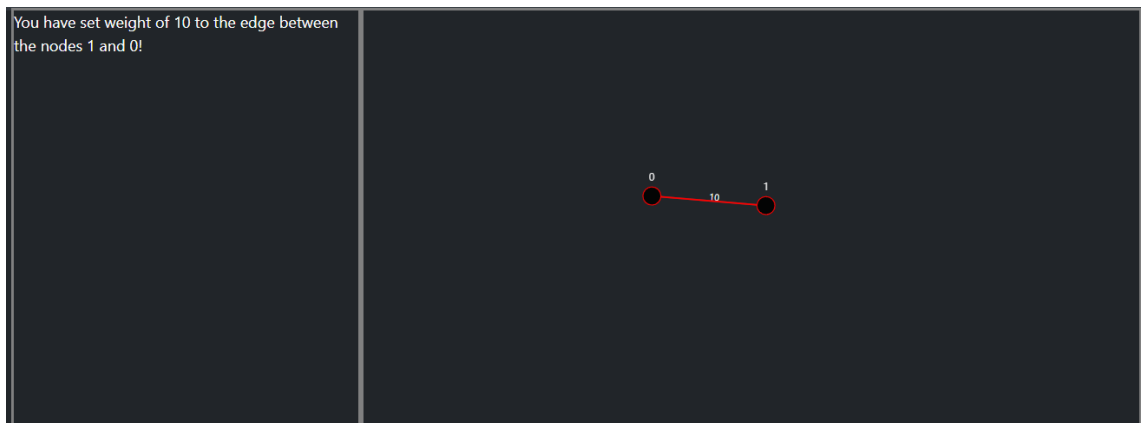


Figure 2.15: Set the weight

2.4.7 Choosing from prepared graphs

The tool does not support importing of the custom graphs yet. However, user might chose from the ones that are already built-in. All that needs to be done is the click on "Select Custom Graph" selection box and the further choice of graph which needs to be imported for the client.

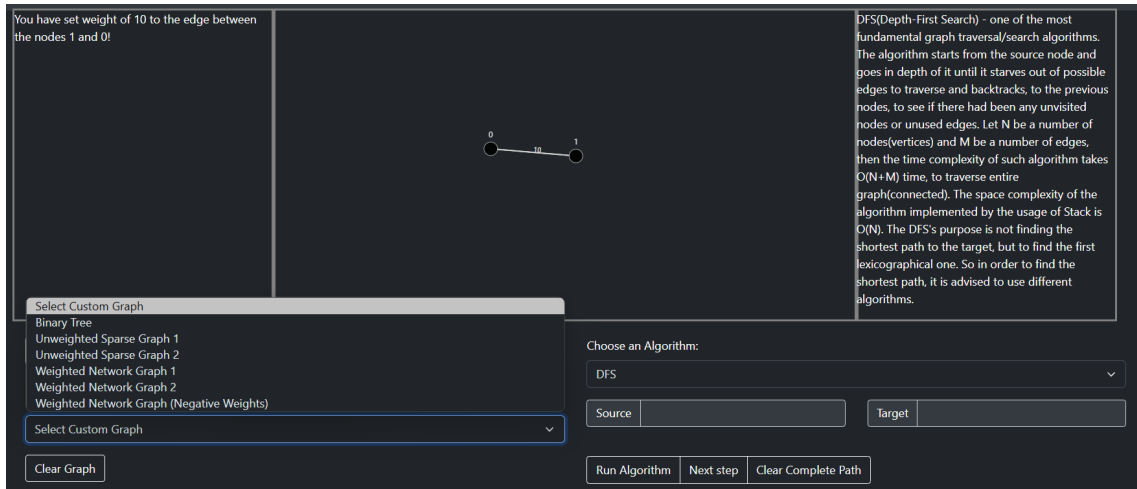


Figure 2.16: Selecting from prepared graphs

2.4.8 Canceling action

After certain actions, such as removing nodes and edges, addition of the edge and setting of weight one has an option to revoke the chosen operation by the "Cancel Current Action" button. It appears every time one of the actions listed in the previous sentence is being executed.

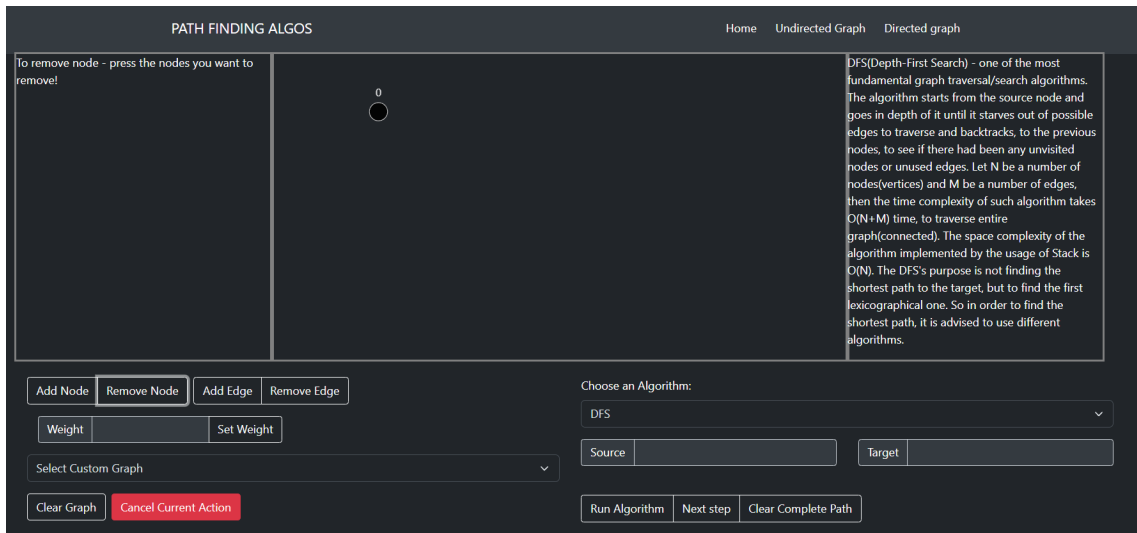


Figure 2.17: Cancel ongoing action

2.4.9 Choosing the algorithm

Now, switching focus to the algorithm control panel, there is a "Choose an Algorithm" selection box. By using it, learner can change the current algorithm

that they would like to be executed on the graph to be chosen. There are four choices such as DFS, BFS, Djikstra and the Bellman-Ford.

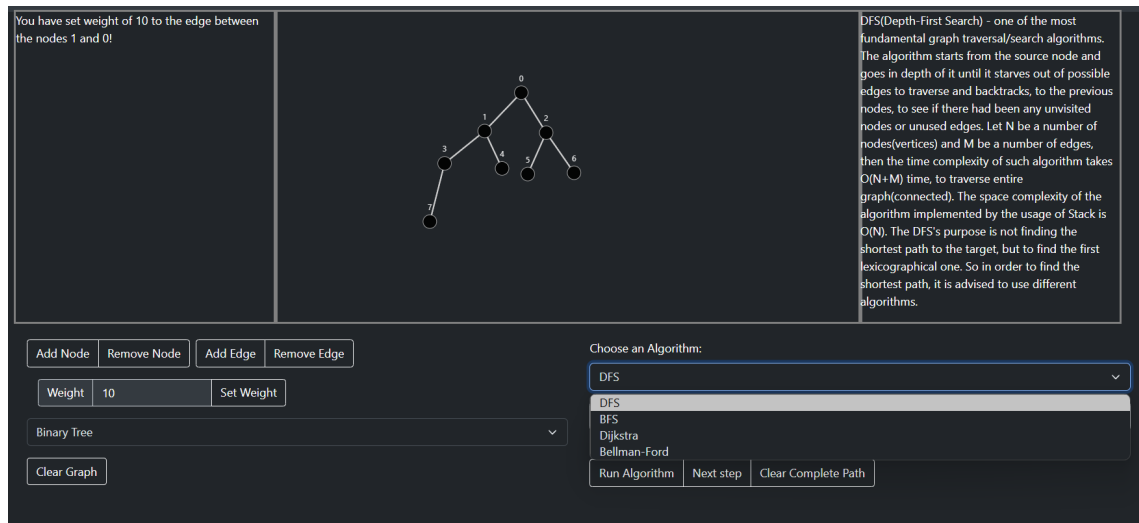


Figure 2.18: Choose the algorithm

2.4.10 Running the algorithm

When one has the complete graph, the algorithm can be executed. Though, there are couple of preparation steps they would need to follow. Initially, choose the source and target nodes from the graph. User can choose these by clicking on the relevant source and target read-only input boxes and then clicking on the nodes they want to select as a source or target. The order is clear, if one follows instructions of the communication panel.

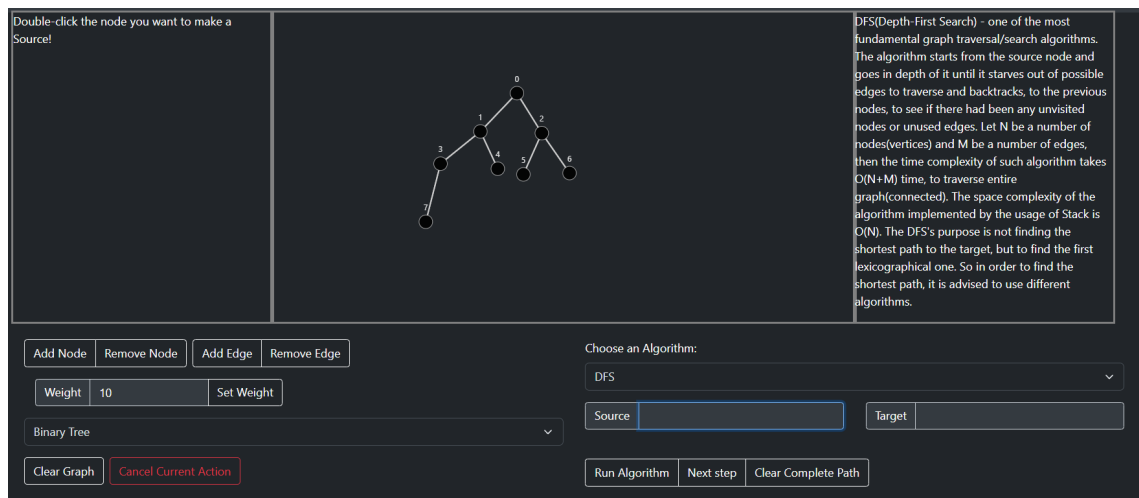


Figure 2.19: Choose the source node

As we have the source and target nodes, we might continue with the execution. In order to proceed, there is a "Run Algorithm" button, which one shall click.

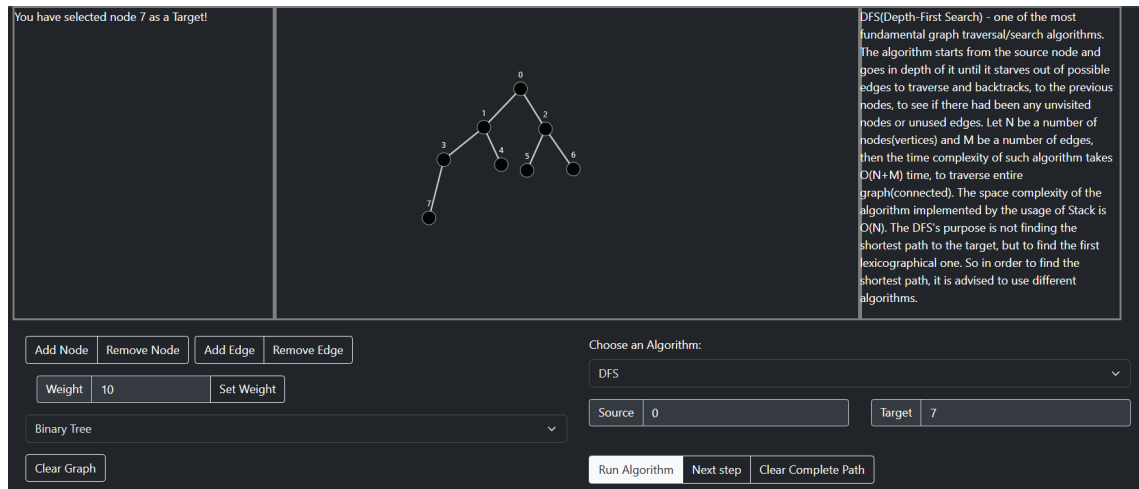


Figure 2.20: Running the algorithm

Figure 2.21 shows the result of pressing on the "Run Algorithm" button. The source node is colored yellow which indicates the current node being processed by the algorithm. Execution is continued by the click of a "Next Step" button.

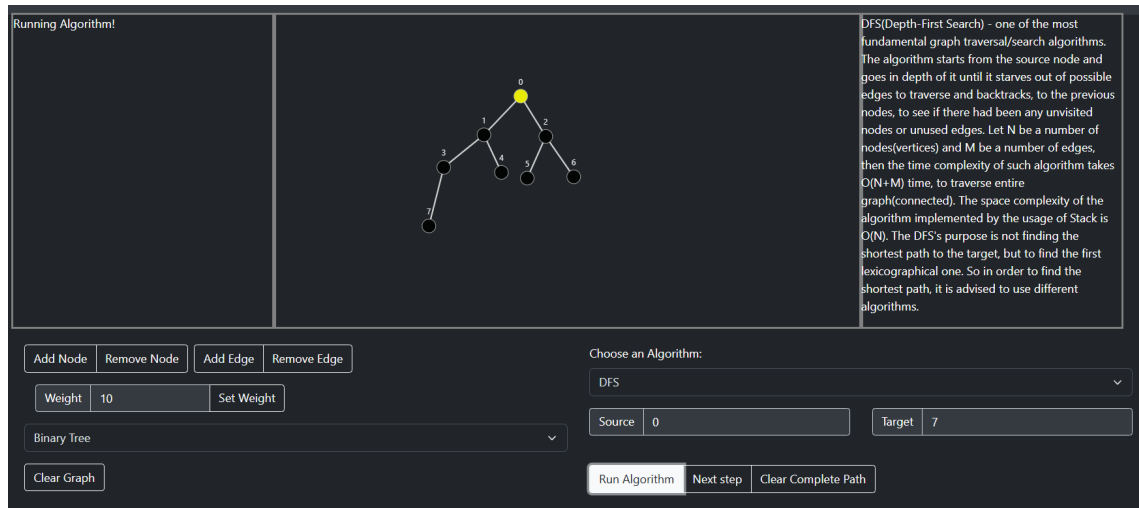


Figure 2.21: After pressing run algorithm

After clicking on the button, algorithm will process one step. On figure 2.22, it is clear that the current processed node has changed, and the previously visited node is colored gray. One might find all of the nodes that have been visited during the time of execution being colored gray, as it is clearly the case. Gray nodes are already visited nodes.

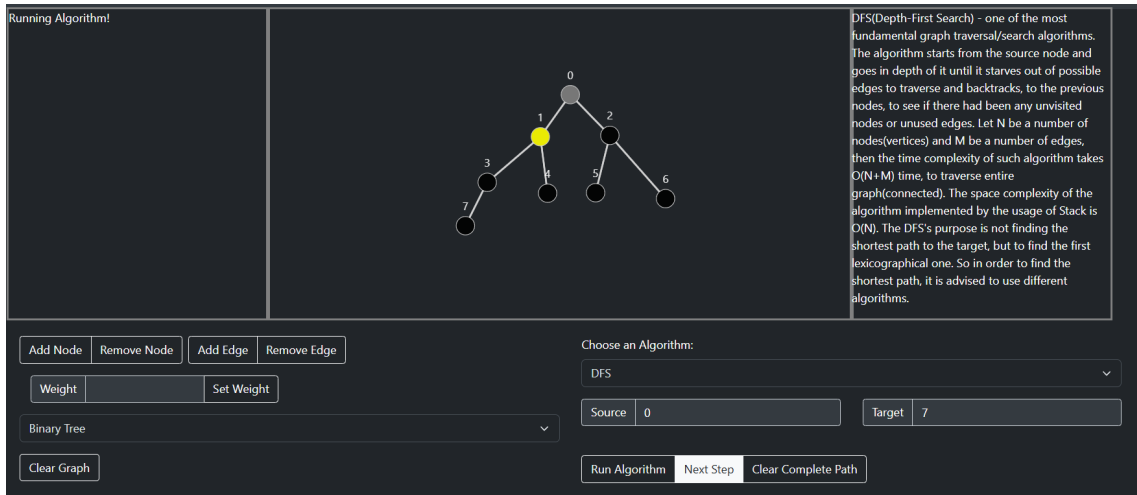


Figure 2.22: After pressing next step

Clicking on the "Next Step" button enough times concludes the final result of the chosen algorithm and results in the path found in the graph from source A to target B nodes. All of the nodes in the path are colored whether cyan or green, where cyan is used for the source and target nodes whereas green for all intermediate nodes of the path. Also, the path is printed out to the communication panel and its cost, in case we are dealing with the algorithm that finds an optimal path based on its weight accumulation.

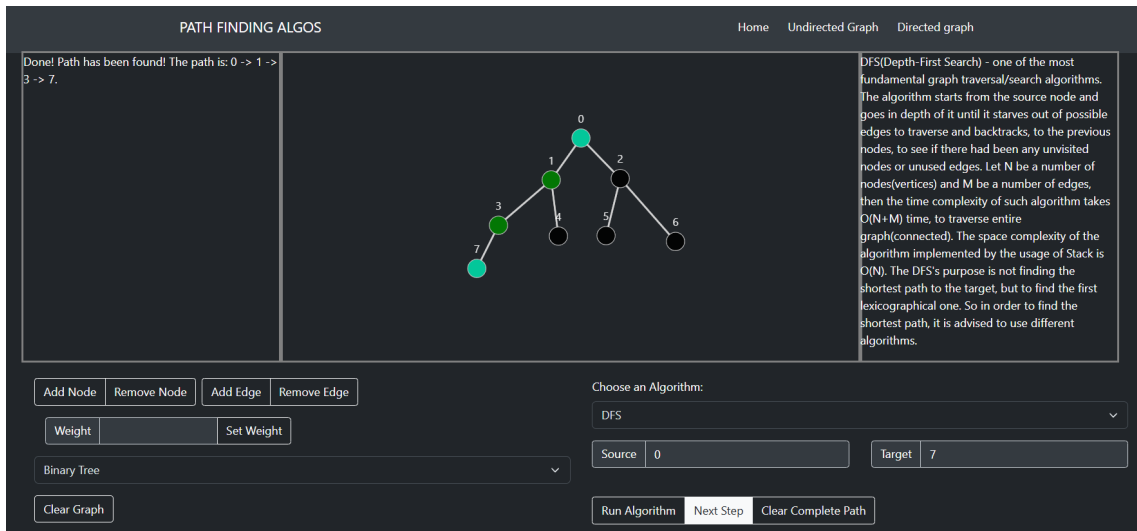


Figure 2.23: Completing algorithm

In order to get rid of the completed paths' coloring, learner can click on "Clear Complete Path" button. After that, all of the nodes regain their original black color.

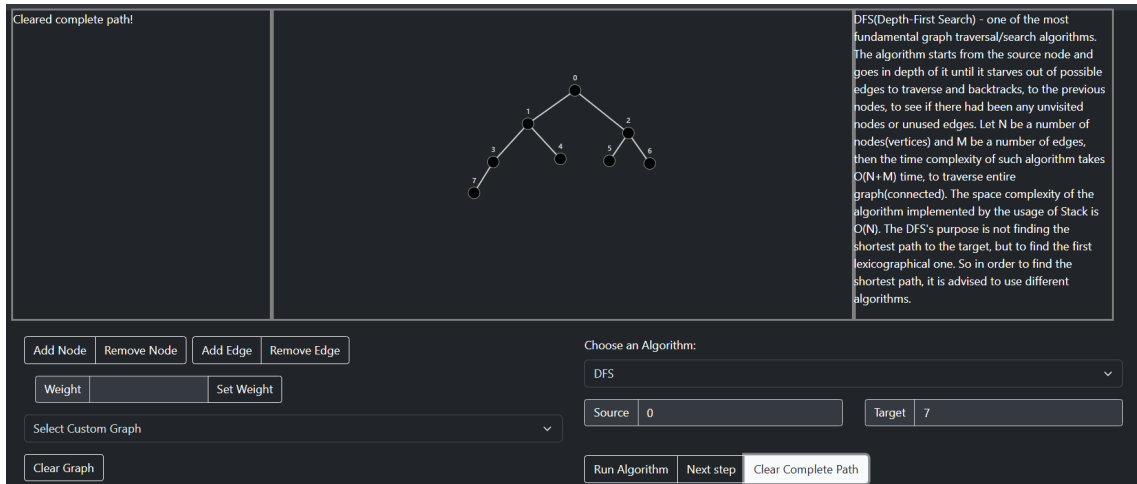


Figure 2.24: Clearing the path

2.4.11 Directed graph page

As every example above has been showed on undirected graph, there is a directed graph page on figure 2.25, so that one sees what the different graph looks like.

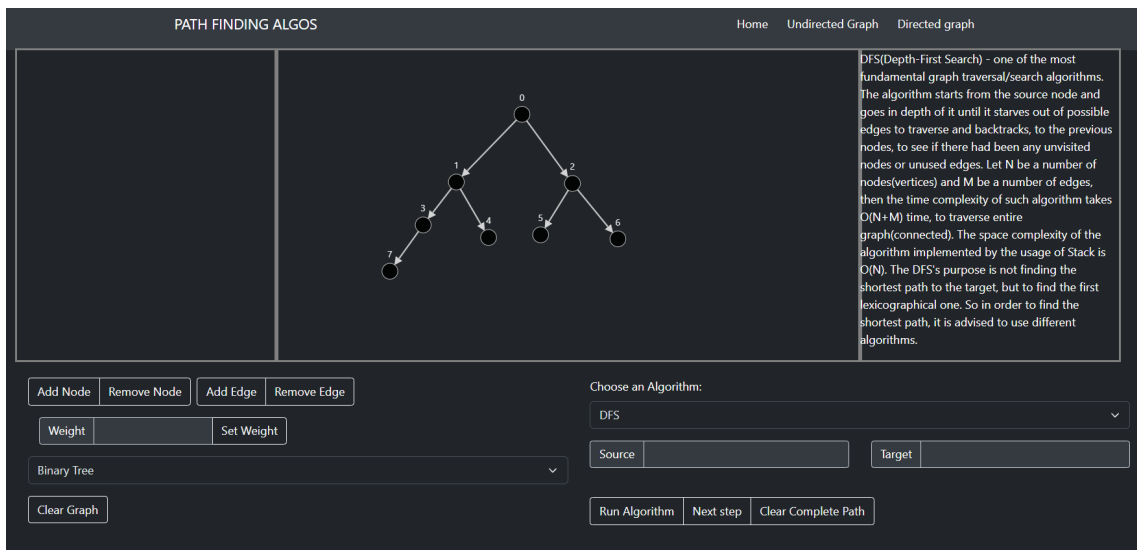


Figure 2.25: Directed graph

So far, it is all a learner needs to know about to proceed with the usage of tool. If one does not plan to change the behavior of the visualizer, then the next chapter is irrelevant to them. Various topics have been covered such as the graph, algorithm control and how one should make use of the communications inside the tool.

Chapter 3

Developer Documentation

In this chapter, one might find relevant information and details regarding the architecture, implementations and testing of the tool. All of it helps to better understand how to change anything or what to do in order to apply it to your own application or a tool if one developer wants to borrow some use cases implementation in the visualizer. Main goal of the chapter is to clearly deliver the structure and preparation that has been made in the background of creation of a tool.

3.1 Architecture

One of the most crucial parts in the development is to have your architecture planned neatly and thoroughly in order to meet the resources one has in their hands.

“At the heart of software architecture is the principle of abstraction: hiding some of the details of a system through encapsulation in order to better identify and sustain its properties.“ [7]

By Shaw, software architecture is the abstraction for the end user, so that the certain details are hidden from them in order to keep everything clear and show the bigger picture of the things going under the hood of a software.

“The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML gives you a standard way to write a system’s blueprints, covering conceptual things such as business processes and system functions, as well as concrete things such as classes written in a specific programming language, database

schemas, and reusable software components.“ [8] UML is used to better show the architectural decisions made as it is both popular and easy to read.

3.1.1 Use cases

On the figure 3.1 you might see the use case for the visualization tool. “A use case is a prose description of a system’s behavior when interacting with the outside world.“ [9] By following the definition of the Alistair, that he has introduced in his article, there are couple of moments on the such prose descriptions one might find there.

So let’s start with the so called "Actor", the entity which will represent the user of a tool and the set of actions that such user might use. In our case "Learner" is our "Actor" and the following actions are introduced: create graph, load prepared graph, control nodes, control edges, choose algorithm, control algorithm.

As one could deduce from the figure, there is a relation between the two actions - create graph and load prepared graph. Relation carries a name "Extend" as it represents the coupling of these actions. Create graph action logically contains the functionality which could be extended in order to implement load prepared graph action.

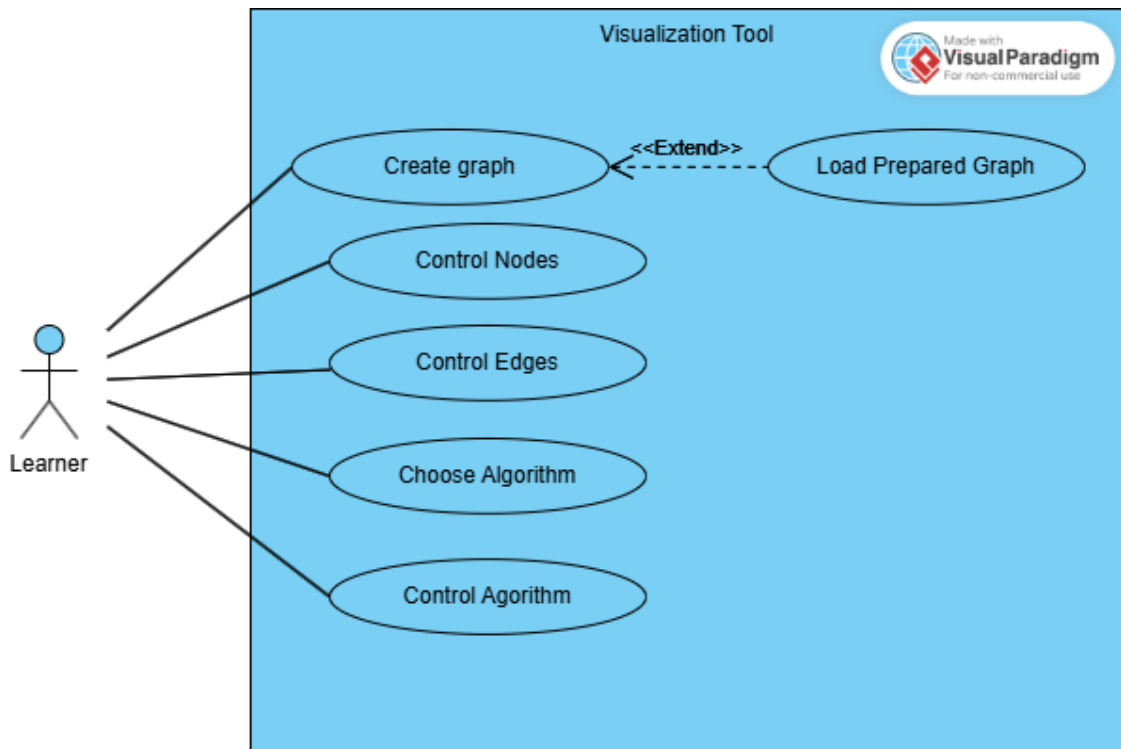


Figure 3.1: Use case diagram

3.1.2 Component diagram

Getting more concrete, developer might find a component diagram on figure 3.2. It explains the high-level design for the components of the tool that interacts with each other in order to achieve the desired result.

Components are the abstractions for the group of methods and classes working together to produce certain output. Though, they are related to each other not by their correspondence but by the end-goal they are created to achieve.

In case of the visualization tool, we have three main components, where one of them is coupled into the other. These are front end, back end and database. Due to the time limitations and initial decisions of the author of such tool, database is coupled into the back end and the mocked database basically inherits its interface. Meaning that the database is implemented in the same place as back end, with all of its methods. Though it may seem that it is a part of back end component, it is also true, that the database serves different purpose and applies different logic in itself. For this cause, there are not two, but three components which one could notice on the diagram.

Front end component serves as a "View" model and only accounts for the representation of data on the screen of the user. Main business logic behind all of data processing is made in the back end. Every time the client - front end - sends certain request to the back end, the latter processes it and send the ready response back to the client with all of the relevant data it may need.

On this stage of the tool, it uses session-based storage. Session proceeds as long as the back end component is up, meaning that if the back end is down, all of the data is cleared up and it is not possible to restore the data. All due to the limitation of the mocked database, which simply stores the data inside of the data structure in the program such as maps, which causes it to store the data only during lifetime of the process.

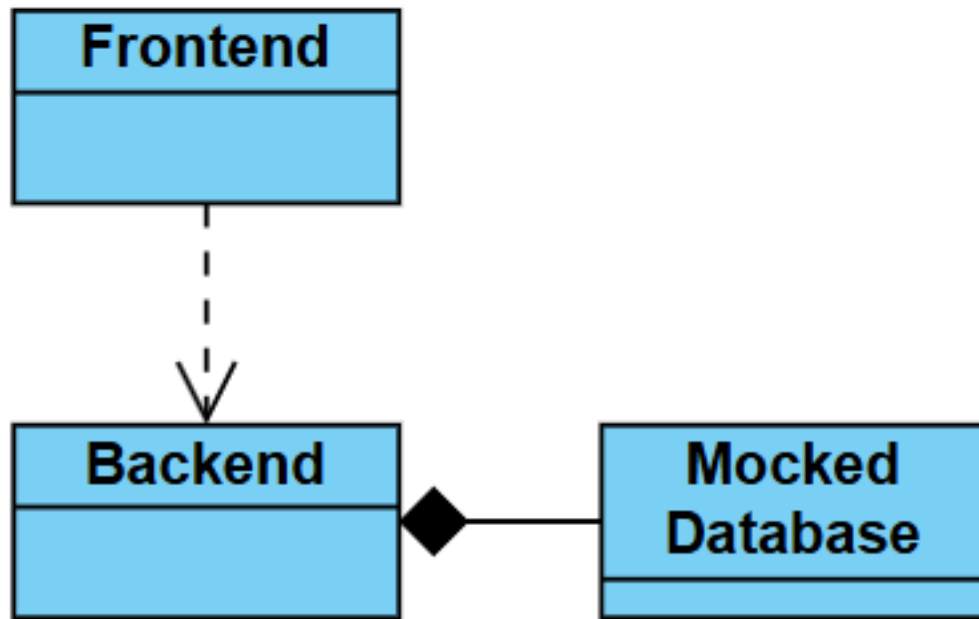


Figure 3.2: Component diagram

3.1.3 Class diagram

Figure 3.3 represents the class diagram of the application. It is not complete in a sense that it contains all of the classes that has been created during the development of the tool but it does contain the main classes that are worth mentioning so that the developer fully grasps the dependencies and coupling which occurs in the code.

View component abstracts all of the code in the front end as it does not contain any class hierarchy that would be necessary to show on the diagram due to its sole goal of representing data. Thus, the view components classes had been reduced to a single component element on the diagram.

“Web Server Gateway Interface, known as WSGI, is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.”[10] In case of the visualization tool, it contains an implementation of WSGI which makes it possible for the client to access and influence the data through requests.

WSGI depends on "Response Handler" which does all of the work related to coupling all of the main logic of the application together. It aggregates the algorithm

and graph classes, which it uses in order to patch responses. Though, it is important to say that it actually decouples the classes as it inverts the dependency to itself instead of having algorithm class anyhow coupled to the graph class. One of the main benefits of such decision is that the graph "component" now can be used freely and "shipped" with other components that might not need the algorithm "component" in them.

Graph and Algorithm interfaces help to abstract the types of theirs. There are six classes that implement them where undirected and directed graphs implement the graph interface whereas Bellman-Ford, Dijkstra, BFS, DFS classes implement the algorithm interface. By such decision, we introduce a new concept called inheritance. “If an object is similar to another, (it has the same attributes and methods), then its class may be inherited from the classes of the similar objects: inherits their properties, and it might modify and augment them.” [11] The purpose of the inheritance is clarified now, as the classes have same properties.

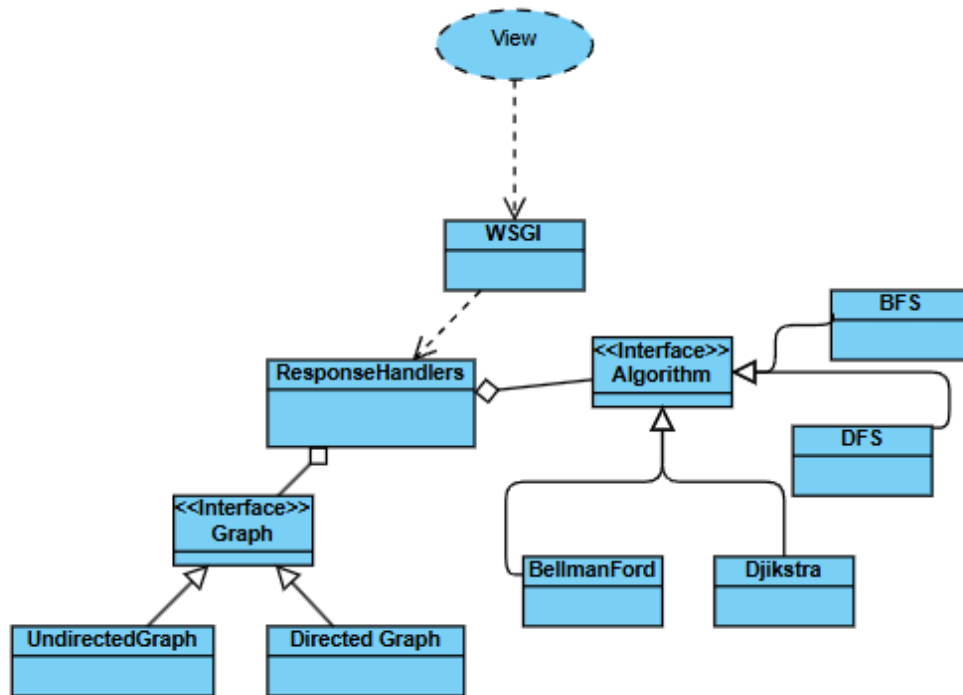


Figure 3.3: Class diagram

3.1.4 Graph representation

Graph representation is an important choice during the development of graphs. The decision on it may affect the future development of the algorithms, sometimes

causing it to be less efficient. Two of the most popular choices are "Adjacency-list" and "Adjacency matrix". In case of the visualization tool, choice has been made in favor of an adjacency list representation.

Here are the definitions of these types by Cormen, "The adjacency-list representation of a graph $G=(V,E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V . For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u,v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G . (Alternatively, it can contain pointers to these vertices.) The adjacency-matrix representation of a graph $G=(V,E)$ assumes that the vertices are numbered $1, 2, \dots, |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that" [3]

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

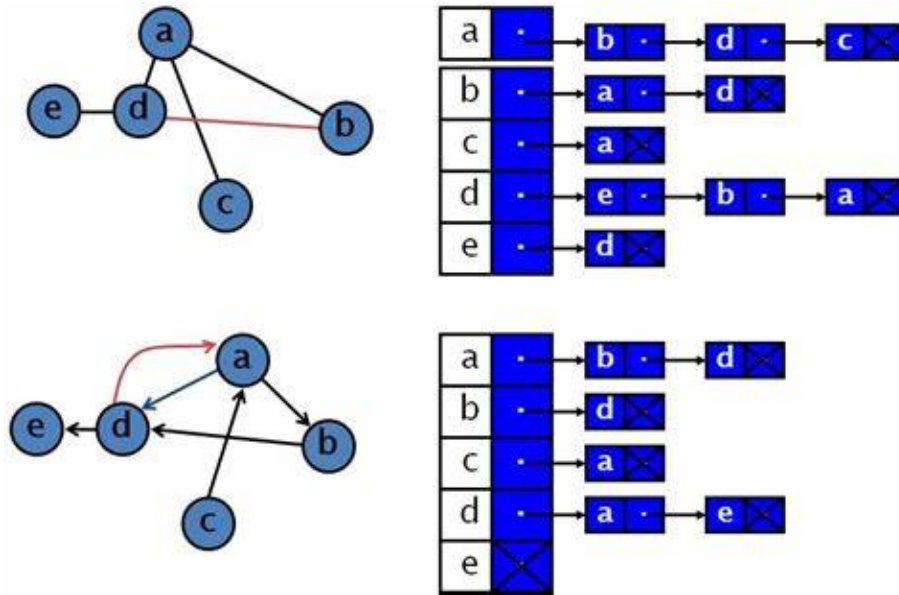


Figure 3.4: Adjacency list representation

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Figure 3.5: Adjacency matrix representation

3.2 Technical details

So that one could clearly comprehend what is happening in the code and why such decisions had been made, they must know the technical details behind the code.

3.2.1 Libraries

Decision on the libraries made a big role in the complexity of developing the tool. Some of them made things harder to implement, some of them reduced a lot of work that could have been done if they were not used. Though it is fair to mention that nothing is perfect and when it comes to the choice of libraries used in development, it is no different. You have to deal with the limitations of certain libraries and at times go through the places you have never expected yourself to be in.

Here are the main libraries that have been used in development of the tool, both in front and back ends.

React JS

As one of the main library for the development of the user end, the React has been chosen. Some definitions from its github page:

“React is a JavaScript library for building user interfaces.” [12]

“Declarative: React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render

just the right components when your data changes. Declarative views make your code more predictable, simpler to understand, and easier to debug.“ [12]

“Component-Based: Build encapsulated components that manage their own state, then compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep the state out of the DOM.“ [12]

The library is popular among the developers, especially the ones, who specifically create client-side applications. Choice came to it, as there were loads of tutorials and templates to learn from, though it did not make it easy to learn. It was one of the choices, which consumed a lot of time just on the phase of learning it. Are there any regrets? No, there was a lot to learn and grasp, one could only be glad to know so much things, and certainly, after learning it, you most probably will know it.

React-bootstrap

Though React is a good choice for the front, one could not achieve an easy UI development without such library as react-bootstrap.

“React-Bootstrap is a complete re-implementation of the Bootstrap components using React. It has no dependency on either bootstrap.js or jQuery.“ [13]

It helped a lot during the development of the tool, as it self-contains a great amount of ready templates and it significantly decreases the time one needs to spend on developing the UI.

React-d3-graph

React-d3-graph is the library used for drawing a graph on ones page. It is build on top of another JavaScript library, which is d3.js. “D3 (or D3.js) is a free, open-source JavaScript library for visualizing data. Its low-level approach built on web standards offers unparalleled flexibility in authoring dynamic, data-driven graphics. For more than a decade D3 has powered groundbreaking and award-winning visualizations, become a foundational building block of higher-level chart libraries, and fostered a vibrant community of data practitioners around the world.“ [14]

D3 solves wide range of problems, while react-d3-graph gathers everything you need to deal with graphs. For this very reason, it is decided to be an optimal choice

as there would be no need in having access to all of the features of D3 library, as it would bring nothing but increased complexity.

It is trivial to choose abstractions when you have a choice.

React-query

One last library that has been used alongside react is react-query. It helped in reducing the number of hook usage such as useEffect and useState, making it more maintainable and bugs resistant.

“TanStack Query (FKA React Query) is often described as the missing data-fetching library for web applications, but in more technical terms, it makes fetching, caching, synchronizing and updating server state in your web applications a breeze.“ [15]

Basically, in the application, only two hooks from the library have been used which are useMutation and useQuery. However, these two hooks has replaced with itself huge amount of repetitive code that resided in the application with just the usage of useEffect hook. The impact of this library on the development process has been crucial.

HTML

“HTML (HyperText Markup Language) is the most basic building block of the Web. It defines the meaning and structure of web content. "Hypertext" refers to links that connect web pages to one another, either within a single website or between websites. Links are a fundamental aspect of the Web. By uploading content to the Internet and linking it to pages created by other people, you become an active participant in the World Wide Web.“ [16]

Static HTML has not been used widely, rather a single instance is used to run the whole web page on. It is all possible due to the way react renders the application.

Python Flask

“Flask is a web framework that allows developers to build lightweight web applications quickly and easily with Flask Libraries. It was developed by Armin Ronacher, leader of the International Group of Python Enthusiasts(POCCO). It is basically based on the WSGI toolkit and Jinja2 templating engine.“ [17]

Whole back end site of the application is built using python. Logic of the graphs and algorithms, response handlers and database, though all of it would not make sense without a proper way of establishing a communication between two ends. Flask is making it possible by giving the developer a WSGI where one could list his endpoints.

3.3 Implementations

By this part of the documentation, one has to gain enough of information in order to understand the implementation of code. Indeed, a reasonable background is needed as well.

Front end related code is not going to be covered as it is simply of no need, to show how the application "shows" itself, rather the business logic code and endpoints are going to be exposed to the developer for the sake of clarification.

3.3.1 API endpoints

Server envelops the following api endpoints in itself.

- GET - get graph, get database
- POST - import graph from file, add node, add edge, add edge with weight, run algorithm
- DELETE - clear graph, remove node, clear algorithm, remove edge
- PUT - set graph type, set weight, change algorithm, do step

By calling these endpoints, the developer might change the data and get the changed data as a response, in order to use it. This is how the client side of the application works, calling the data, receiving, "showing".

3.3.2 Graph management

Let's see how some of the main endpoints actually get the graph data or manage it in case of a modification request.

Getting graph

First of all, the endpoint method declared in the WSGI calls for a graph response handler's method `get_graph`, which will return the parsed response, it also appends the data of the current algorithm if there is such and sends back the prepared response.

```
1  @app.route("/graph", methods=["GET"])
2  def get_graph():
3      response = GraphResponseHandler.get_graph()
4      response.update(AlgorithmResponseHandler.get_algorithm())
5      return jsonify(response)
```

Code 3.1: Get graph endpoint

Going deeper, `get_graph` method of the graph response handler will call the database through, so-called, session storage that stores current graph, algorithm and database, and get the graph table in it. After that, it just sends back the ready graph data.

```
1  @staticmethod
2  def get_graph():
3      graph_data = {'graph': Storage.database.get_table('graph')}
4      if graph_data['graph'] == {}:
5          graph_data.update({'nodes': [], 'edges': {}})
6      return graph_data
```

Code 3.2: Get graph method in response handler

Changing graph

As we have two main types of graph, we have an endpoint for this use case as well. All it takes is a name of the graph and passes down the responsibility to the response handler.

The method of the response handler on code field 3.3, checks whether the provided type name is not the same as the one that is already in use. After the check, it calls session storage's method `change_graph`.

```
1  @staticmethod
2  def set_graph_type(graph_type_name):
3      if isinstance(Storage.graph, graph_types.get(graph_type_name)().
4          .__class__):
```

```
4         return {'msg': f'Graph is already of type - {graph_type_name}'  
5             },  
6         Storage.change_graph(graph_types.get(graph_type_name)())  
7         return GraphResponseHandler._update_database_data()
```

Code 3.3: Change graph type

In the session storage, one might see that it does nothing other than initializing graph in use to the new graph instance. Which means, that the previous graph will be garbage collected with its fields.

One way to get the instance of certain class by its name is application of the factory-like design pattern. Graph factory class employs this design pattern yet, with some modifications included in it. Graph type dictionary actually makes it possible to create certain instances by their names.

```
1     @staticmethod  
2     def change_graph(new_graph):  
3         Storage.graph = new_graph
```

Code 3.4: Change graph in storage

```
1     class GraphFactory:  
2         @staticmethod  
3         def create_UndirectedGraph():  
4             return UndirectedGraph()  
5  
6         @staticmethod  
7         def create_DirectedGraph():  
8             return DirectedGraph()  
9  
10  
11     graph_types = {  
12         'undirected': GraphFactory.create_UndirectedGraph,  
13         'directed': GraphFactory.create_DirectedGraph,  
14     }
```

Code 3.5: Graph Factory

Importing graph from file

Rest of the endpoints do pretty much the same thing as the previous endpoint does. Import graph from file endpoint takes graph file index and calls the response

handler to handle it.

As shown on the figure 3.6 under the persistence folder in the src, we have the graph templates. There the numbers appended at the end of each one corresponding to their index.

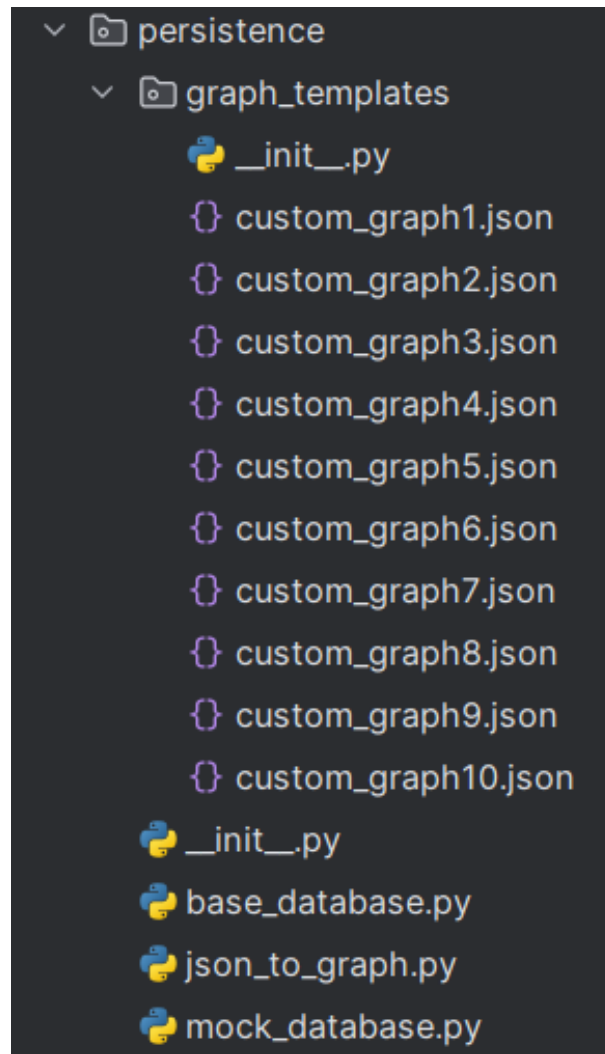


Figure 3.6: Custom graphs

```
1  {
2    "graph": {
3      "nodes" : [0, 1, 2, 3, 4],
4      "edges" : [
5        [[1, 1]],
6        [[2, -10], [3, 1]],
7        [[4, 1]],
8        [],
9        [[3, 9]]
10   ]
}
```

```
11     }  
12 }
```

Code 3.6: Custom graph template

Response handler, in its method, looks for the file with the corresponding index, after which it sends the file it found down to the JsonToGraph class and its method json to graph. At the end, after initializing parsed graph to the one in the storage, method sends the updated data back.

```
1  @staticmethod  
2  def import_ready_graph(index_of_graph):  
3      dirs_to_find = ['app', 'src', 'persistence', 'graph_templates']  
4      path = os.getcwd()  
5      graph_file_name = 'custom_graph' + str(index_of_graph) + '.json'  
6      while len(directory := os.listdir(path)) > 0:  
7          if graph_file_name in directory:  
8              path += os.sep + graph_file_name  
9              break  
10  
11      any_path_found = False  
12      for _dir in directory:  
13          if _dir in dirs_to_find:  
14              path += os.sep + _dir  
15              any_path_found = True  
16              break  
17  
18      if not any_path_found:  
19          break  
20  
21      path_to_graph_file = path  
22      Storage.graph = JsonToGraph.json_to_graph(path_to_graph_file,  
23          Storage.graph.__class__)  
23      return GraphResponseHandler._update_database_data()
```

Code 3.7: Import ready graph response handler

Json to graph method in its turn, read the data from a file and creates a new graph of the type that the previous graph had.

Here one could see the beneficial parts of the interfaces, where the code uses the graphs methods add edge and add node without caring whether it will work the way

the method needs it to work. Graph interface brings with itself the simplification to the code.

```
1  @staticmethod
2  def json_to_graph(json_path, class_of_graph):
3      with open(json_path, 'r') as json_file:
4          json_dict = json.load(json_file)
5          if class_of_graph is UndirectedGraph:
6              new_graph = UndirectedGraph()
7          else:
8              new_graph = DirectedGraph()
9
10         json_dict = json_dict['graph']
11         for _ in json_dict['nodes']:
12             new_graph.add_node()
13
14         index = 0
15         for edges_of_node in json_dict['edges']:
16             for edge in edges_of_node:
17                 new_graph.add_edge(new_graph.nodes[index], new_graph.nodes[
18                     edge[0]], edge[1])
19             index += 1
20
21         return new_graph
```

Code 3.8: Json to graph method

Addition of node

The method of adding a node is the same for both undirected and directed graphs. Also, there is nothing complicated going on, just the creation of a node and appending of it to the list of nodes and edges, after all of which, we return the node.

```
1  def add_node(self) -> Node:
2      new_node = Node()
3      self.nodes.append(new_node)
4      self.edges.append([])
5      return new_node
```

Code 3.9: Adding node

Node class is more interesting as it consists of two useful features, that one could call a "constructor" and a "destructor". Each node has a unique id that identifies it on the graph, as you could see already in the user documentation. All of it comes from here, where during the initialization of the node, it calls the node id generators method, which generates the id for the node, and saves it in itself.

On code 3.10, the developer might see that it calls for its method of id generation, and saves the hash of the node object, lastly, increasing the count of nodes. It is saving the hash of the object due to the requirements of the garbage collector which checks whether the object is being referenced anywhere in the code and calls for its destructor only when it does not.

```
1  class Node:
2      def __init__(self):
3          NodeIDGenerator.generate_id_node_pair(self)
4
5      def __del__(self):
6          NodeIDGenerator.remove_id_of_node(self)
```

Code 3.10: Node class

```
1      @staticmethod
2      def generate_id_node_pair(node):
3          new_id = NodeIDGenerator.generate_id()
4          NodeIDGenerator.ids_and_nodes[new_id] = node.__hash__()
5          NodeIDGenerator.node_count += 1
```

Code 3.11: Generating ids for nodes

Removal of node

Removal of node is more complicated compared to the addition, due to its need to check the edges outgoing from it and incoming into it. Developer can see it on code 3.12 where it does the exact thing mentioned in the previous sentence, though it is not all of the work that is done, as we will have the garbage collector call the destructor on the node as it is no longer in use.

```
1  def remove_node(self, index):
2      self.nodes.pop(index)
3      self.edges.pop(index)
4
```

```
5     for edge in self.edges:
6         index_to_pop = -1
7         for i in range(len(edge)):
8             if edge[i][0] == index:
9                 index_to_pop = i
10            if index_to_pop != -1:
11                edge.pop(index_to_pop)
12            for i in range(len(edge)):
13                if edge[i][0] > index:
14                    edge[i][0] -= 1
```

Code 3.12: Remove node in graph

When it comes to the deconstructor, it calls the id generators method - remove id of node. Getting the node instance, it finds its id and deletes its instance, then shifting all of the hashes one to the left, it deletes its hash from the id generator as well.

```
1     @staticmethod
2     def remove_id_of_node(node):
3         id = NodeIDGenerator.get_id_of_node(node)
4         NodeIDGenerator.remove_id(id)
5         for key in range(id, NodeIDGenerator.node_count-1):
6             NodeIDGenerator.ids_and_nodes[key] = NodeIDGenerator.
7                 ids_and_nodes[key+1]
8             NodeIDGenerator.ids_and_nodes.pop(NodeIDGenerator.node_count -
9                 1)
10        NodeIDGenerator.node_count -= 1
```

Code 3.13: Id removal of node

Addition of edge

Addition of edge is implemented in the simplest way possible. Method checks whether there is an edge between nodes, finding these nodes' indexes and adding them to each others' adjacency list.

```
1     def add_edge(self, first_node, second_node, weight=0):
2         if not (first_node in self.nodes or second_node in self.nodes):
3             raise InvalidNodeException("One of the nodes in the edge is
4                 not in the nodes set.")
5         first_node_index = self.nodes.index(first_node)
```

```
5     second_node_index = self.nodes.index(second_node)
6     self.edges[first_node_index].append([second_node_index, weight
7     ])
8     self.edges[second_node_index].append([first_node_index, weight
9     ])
```

Code 3.14: Adding edge

Removal of edge

To remove the edge, first we check for the edge cases, when one of the nodes does not exist, then we check whether there is an edge between the nodes. After couple of checks, we find the nodes in the list and each others adjacency list, then we remove them from both.

```
1  def remove_edge(self, first_node, second_node):
2      if not (first_node in self.nodes or second_node in self.nodes):
3          raise InvalidNodeException("One of the nodes in the edge is
4          not in the nodes set.")
5      first_node_index = self.nodes.index(first_node)
6      second_node_index = self.nodes.index(second_node)
7      if not (second_node_index in [node[0] for node in self.edges[
8          first_node_index]] or
9      first_node_index in [node[0] for node in self.edges[
10         second_node_index]]):
11          raise InvalidNodeException("There is no edge between the
12          nodes")
13      first_to_second_edge_index = [node[0] for node in self.edges[
14          first_node_index]].index(second_node_index)
15      second_to_first_edge_index = [node[0] for node in self.edges[
16          second_node_index]].index(first_node_index)
17      self.edges[first_node_index].pop(first_to_second_edge_index)
18      self.edges[second_node_index].pop(second_to_first_edge_index)
```

Code 3.15: Removing edge, undirected

In case of a directed graph, we remove only one edge. The one outgoing from the first node to the second one. Indeed, some checks are required to be done beforehand.

```
1  def remove_edge(self, first_node, second_node):
2      if not (first_node in self.nodes or second_node in self.nodes):
```

```
3     raise InvalidNodeException("One of the nodes in the edge is not
4         in the nodes set.")
5     first_node_index = self.nodes.index(first_node)
6     second_node_index = self.nodes.index(second_node)
7     if not (second_node_index in [node[0] for node in self.edges[
8         first_node_index]]):
9         raise InvalidNodeException("There is no edge between the edges"
10            )
11     first_to_second_edge_index = [node[0] for node in self.edges[
12         first_node_index]].index(second_node_index)
13     self.edges[first_node_index].pop(first_to_second_edge_index)
```

Code 3.16: Removing edge, directed

Setting weight

Each edge represents carries certain weight, by default it is 0. In case one want to make the graph truly "weighted", there is a need in setting the weight for edges.

Such an option is available in the tool and the endpoint resembles edge managing endpoints. It gets three inputs: edge incoming node, edge outgoing node and weight, so that it can find the corresponding edge and set the weight of it.

As one might see on figure 3.14, initially it checks whether there is such edge incoming node and edge outgoing node in the graph, then it finds those nodes indexes in the graph and each others adjacency list, in case of undirected graph, as the next step it sets the weight of edges to the one that has been passed as a parameter. If one would have to deal with directed graph, the method would be similar to the one on figure 3.14, the only difference is that we set the weight only for the edge from the edge outgoing node to edge incoming node.

```
1     def set_weight(self, first_node, second_node, weight):
2         if not (first_node in self.nodes or second_node in self.nodes):
3             raise InvalidNodeException("One of the nodes in the edge is
4                 not in the nodes set.")
5         first_node_index = self.nodes.index(first_node)
6         second_node_index = self.nodes.index(second_node)
7         second_node_in_first_nodes_index = [node[0] for node in self.
8             edges[first_node_index]].index(second_node_index)
9         first_node_in_second_nodes_index = [node[0] for node in self.
10             edges[second_node_index]].index(first_node_index)
```

```
8
9     self.edges[first_node_index][second_node_in_first_nodes_index
10         ][1] = weight
    self.edges[second_node_index][first_node_in_second_nodes_index
        ][1] = weight
```

Code 3.17: Setting weight for the edge

3.3.3 Algorithm management

As one could have noticed already, the get graph method appends the algorithm data to the response as well. So in this section, all of the main information regarding the algorithms are going to be covered.

Changing algorithm

There are four types of algorithm which you can change calling the relevant endpoint. It works pretty much the same way as change graph endpoint. Method initializes new algorithm instance which it creates using the algorithm factory. After that, it get the description of such algorithm and sends back the updated data as a response.

The developer can find the algorithm factory on code 3.19 and the corresponding response handler on code 3.18.

```
1     @staticmethod
2     def choose_algorithm(algorithm_name):
3         Storage.change_algorithm(algorithm_names.get(algorithm_name))
4         updated_table = Storage.database.get_tables().get('algorithm',
5             {})
6         updated_table.update({'description': Storage.algorithm.
7             description()})
8         return AlgorithmResponseHandler._update_database_data(
9             updated_table)
```

Code 3.18: Choose algorithm handler

```
1     algorithm_names = {
2         'dfs': AlgorithmFactory.create_DFS(),
3         'bfs': AlgorithmFactory.create_BFS(),
4         'dijkstra': AlgorithmFactory.create_Dijkstra(),
```

```
5     'bellmanford': AlgorithmFactory.create_BellmanFord()  
6 }
```

Code 3.19: Algorithm factory

Starting algorithm

One can start the algorithm by querying run algorithm endpoint of the tool. Then the endpoint call for the handler, which in its turn, runs it on the algorithm which it keeps in the session storage. As we have all of the algorithms classes derive a single algorithm abstract base class, we don't need to deal with handling of the specific algorithms methods being run. After that, we check whether the path has been found, and if not, we update the results dictionary object correspondingly. At the end, we send the final results as we append the description of the algorithm to the results.

```
1  @staticmethod  
2  def run_algorithm(source, target):  
3      algorithm_data = Storage.algorithm.run(Storage.graph.edges,  
4          source, target)  
5      if algorithm_data['path'] == Storage.algorithm.PATH_NOT_FOUND:  
6          algorithm_data.update({'currentStep': -1, 'currentState':  
7              FINISHED_STATE})  
8      else:  
9          algorithm_data.update({'currentStep': 0, 'currentState':  
10              RUNNING_STATE})  
11      algorithm_data.update({'description': Storage.algorithm.  
12          description()})  
13  
14      return AlgorithmResponseHandler._update_database_data(  
15          algorithm_data)
```

Code 3.20: Running algorithm method

For the sake of clarification, one might want to look into code 3.21. It is a run method of a BFS algorithm, which actually traverses the graph and gives back the result. The algorithm itself is not in its original form. Usually, BFS just traverses the whole graph, but in this implementation it stops, when it finds the first path to the target node. Yet it is not wrong, it is still important to highlight. BFS algorithm

in the tool is implemented with the usage of queue abstract data structure, which is also has been implemented by the author.

All of the algorithms implement well-known patterns and there is no difference between them, with the exception of The Bellman-Ford, where it appends the `isBellmanFord` boolean value to the result, identifying that it needs to be handled in a different manner.

```
1  def run(self, graph, source, target):
2  if source is None or target is None:
3      return {'path': Algorithm.PATH_NOT_FOUND, 'steps': []}
4  path = []
5  queue = Queue()
6  queue.push((source, [source]))
7  visited = [False] * (len(graph) + 1)
8
9  while not queue.empty():
10     (current_node, current_path) = queue.pop()
11     if visited[current_node]:
12         continue
13     path.append(current_node)
14     visited[current_node] = True
15     if current_node == target:
16         return {'path': current_path, 'steps': path}
17
18     for node in sorted(graph[current_node]):
19         if not visited[node[0]]:
20             queue.push((node[0], current_path + [node[0]]))
21
22  return {'path': Algorithm.PATH_NOT_FOUND, 'steps': path}
```

Code 3.21: Algorithm's run method

Going through steps of algorithm

Moving on through the methods of algorithm, developer might see the `do_step` method on code 3.22, where the aforementioned `isBellmanFord` value could be noticed.

The method, as a first step, checks whether the algorithm it is running is the Bellman-Ford or not.

When it is not, it checks whether it is a last step or not, if it is, then we finish the algorithm, if not, in this case we increase the step index.

Though, in case it is actually the Bellman-Ford algorithm, we look whether it is a last edge, in other words link, to be checked and apply according changes to the result.

In the end, it sends the updated data back.

```
1  @staticmethod
2  def do_step():
3      updated_table = Storage.database.get_tables().get('algorithm',
4      {})
5      if updated_table.get('isBellmanFord', '') == 'True':
6          if updated_table['currentLink'] >= len(updated_table['links']
7          ]) - 1 or updated_table['currentLink'] == -1:
8              updated_table.update({'currentLink': -1, 'currentState':
9              FINISHED_STATE})
10         else:
11             updated_table.update({'currentLink': (updated_table['
12             currentLink'] + 1)})
13     else:
14         if updated_table['currentStep'] >= len(updated_table['steps']
15         ]) - 1:
16             updated_table.update({'currentStep': -1, 'currentState':
17             FINISHED_STATE})
18         else:
19             updated_table.update({'currentStep': (updated_table['
20             currentStep'] + 1)})
21     return AlgorithmResponseHandler._update_database_data(
22         updated_table)
```

Code 3.22: Algorithm step method

3.4 Testing

Testing is and has played a crucial role in the development. The Test-Driven Development has been used throughout the process and for this cause, there are over a hundred of tests written.

“Test-driven development (TDD) is a method of coding in which you first write a test and it fails, then write the code to pass the test of development, and clean

up the code. This process recycled for one new feature or change. In other methods in which you write either all the code or all the tests first, TDD will combine and write tests and code together into one.“ [18] Though it might seem that it slows the developing process, it actually prevents from making mistakes in the future. Also, with the CI written for the repository, it helped to avoid further problems a lot of times in the process.

```
1  ---
2  name: CI
3
4  push:
5    branches: [ "master" ]
6
7  workflow_dispatch:
8
9  jobs:
10   test:
11     runs-on: ubuntu-latest
12
13     steps:
14       - uses: actions/checkout@v4
15       - uses: actions/setup-python@v5
16       with:
17         python-version: '3.12'
18
19       - name: install dependencies
20         run: pip install -r app/requirements.txt
21       - name: Create env file
22         run: |
23           pwd
24           echo "PYTHONPATH=/home/runner/work/
25             PathFinding/PathFinding" >> $GITHUB_ENV
26           cat $GITHUB_ENV
27       - name: set up environment and run tests
28         run: python3 -v app/main.py &
```

```
28     - name: Run the tests
29     run: python3 -m unittest discover -v app/
        src / tests
30
31
32     build:
33         runs-on: ubuntu-latest
34
35         steps:
36             - uses: actions / checkout@v4
37
38             - name: Run a one-line script
39               run: echo Hello , world !
40
41             - name: Run a multi-line script
42               run: |
43                 echo Add other actions to build ,
44                 echo test , and deploy your project .
45     ---
```

Code 3.23: CI yaml file

3.4.1 Tests structure

On figure 3.7, a folder is shown, which contains all of the test suites. They are separated into two main groups - unit tests and E2E tests.

“Unit testing is a type of software testing that focuses on individual units or components of a software system. The purpose of unit testing is to validate that each unit of the software works as intended and meets the requirements.” [19]

On the other hand, E2E does pretty much the same, except for some detail. E2E tests send requests to the server-side and validates the responses. For this reason, E2Es are mostly slower and take more effort in debugging, as you would need an environment for that. Though, the E2E tests are inevitable when one deals with the endpoints in the system.

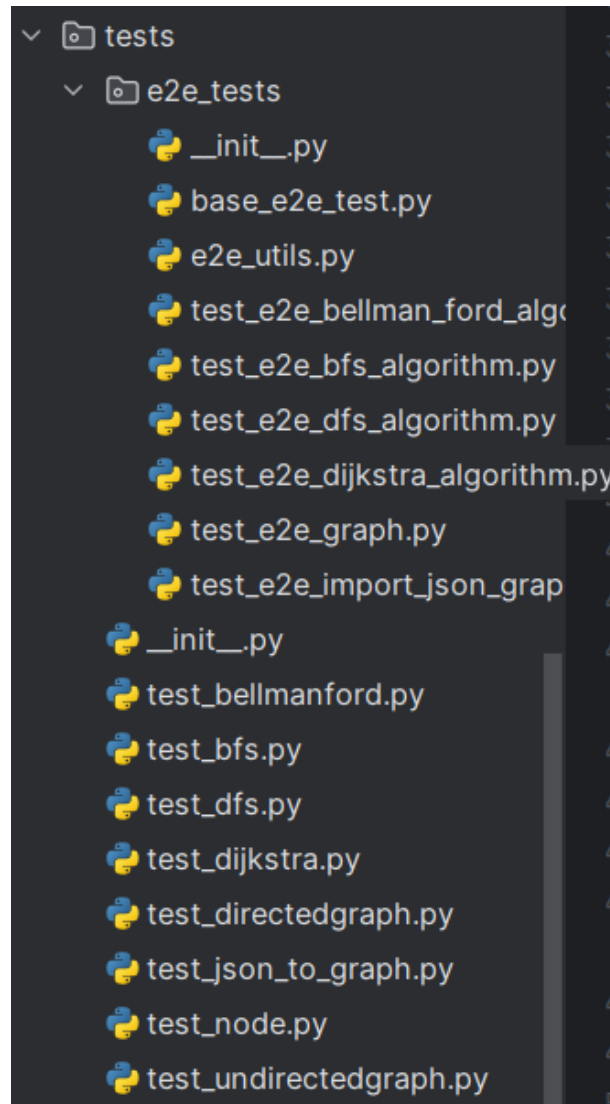


Figure 3.7: Tests folder

3.4.2 Tests implementation

As reviewing hundred tests would probably take around fifty pages to describe, here are some of the crucial and complex test cases. Starting with the unit tests, then E2E tests will help one see the correlation and differences between two ways of testing.

As can be seen on code 3.24, there are three parts of the test: given, when and then. This structure helps us to maintain clear and readable test methods and it also makes it easier to refactor.

In the node test, we see that some nodes are being created and removed after. This test makes sure that after the deconstructor is called on the node instance, it actually removes the necessary data.

```
1 def test_create_and_delete_multiple_times(self):
2     # given-when
3     for i in range(10):
4         self.nodes.append(Node())
5         self.nodes.pop(i)
6         self.nodes.append(Node())
7
8     # then
9     self.assertEqual(len(self.nodes), 10)
10    for i in range(0, len(self.nodes)):
11        self.assertEqual(NodeIDGenerator.ids[i], i)
12        self.assertIn(i, NodeIDGenerator.ids_and_nodes.keys())
13        self.assertEqual(NodeIDGenerator.ids_and_nodes[i], self.nodes
                           [i].__hash__())
```

Code 3.24: Node test

The graph test on code 3.25 is pretty long, yet its purpose is straight. It tries to remove the nodes with the edges between each other, starting from the node, that is in the middle of the other two. Initially, the test creates the nodes and connects them in the given part. As one can see, the middle node is the second node in the test. After that, we can see multiple whens and then's going after each other, where we remove the middle node first, then the first one and at the end, we delete the remaining node, all of that while testing that the behavior of deleting a node is not deviating and not causing an undefined behavior.

```
1 def
    test_remove_three_nodes_with_edges_between_starting_from_middle
      (self):
2     # given
3     node1 = self.graph.add_node()
4     node2 = self.graph.add_node()
5     node3 = self.graph.add_node()
6
7     self.graph.add_edge(node1, node2)
8     self.graph.add_edge(node2, node3)
9
10    # when
11    self.graph.remove_node(NodeIDGenerator.get_id_of_node(node2))
12    del node2
```

```
13
14 # then
15 self.assertEqual([NodeIDGenerator.get_id_of_node(node) for node
16                   in self.graph.nodes], [0, 1])
17 self.assertEqual(len(self.graph.nodes), 2)
18 self.assertEqual(len(self.graph.edges), 2)
19 for edge in self.graph.edges:
20     self.assertEqual(len(edge), 0)
21
22 # when
23 self.graph.remove_node(NodeIDGenerator.get_id_of_node(node1))
24 del node1
25
26 # then
27 self.assertEqual([NodeIDGenerator.get_id_of_node(node) for node
28                   in self.graph.nodes], [0])
29 self.assertEqual(len(self.graph.nodes), 1)
30 self.assertEqual(len(self.graph.edges), 1)
31
32 # when
33 self.graph.remove_node(NodeIDGenerator.get_id_of_node(node3))
34 del node3
35
36 # then
37 self.assertEqual(len(self.graph.nodes), 0)
38 self.assertEqual(len(self.graph.edges), 0)
```

Code 3.25: Undirected graph test

Compared to its E2E alternative, its quite huge. Yet it is obvious, that if we have the correct behavior in the unit tests, it must not be different for the E2E tests. Rather it tests the endpoint itself. The purpose of this E2E test is to make sure, that the endpoints trigger necessary places in the code.

```
1 def test_remove_node(self):
2     # given
3     UndirectedGraphE2ETest._request_add_node()
4     node_id = 0
5
6     # when
7     requests.delete(f'http://127.0.0.1:5000/node/{node_id}').json()
8     graph = self._get_graph()
```

```
9
10 # then
11 self.assertEqual(graph.get('graph'), EXPECTED_GRAPHS[0].get('graph'))
```

Code 3.26: Undirected graph E2E test

At last, the algorithm tests are taking an important part in the application, as they make sure that they are correct. Though, they would not perfectly identify whether the algorithm is valid or not, it would greatly reduce the chances of getting it wrong.

As an example, in the BFS test on code 3.27, it creates a graph and then runs an algorithm on it. In the end, checking its correctness, the test makes sure that it works for such sparse graphs.

```
1 def test_multiple_node_sparse_connected_graph(self):
2     # given
3     for i in range(10):
4         new_node = self.graph.add_node()
5         if i > 0:
6             self.graph.add_edge(new_node, self.graph.nodes[i-1])
7
8     node_id1 = self._get_node_id(0)
9     node_id2 = self._get_node_id(9)
10
11     # when
12     path = self.bfs.run(self.graph.edges, node_id1, node_id2).get('path')
13
14     # then
15     self.assertEqual(path, [self._get_node_id(i) for i in range(len(self.graph.nodes))])
```

Code 3.27: BFS algorithm test

In case of an E2E alternative of the test, it checks whether the next step endpoint is working properly and it returns the values that are expected from it. Significant part of such tests is having an expected output which one could actually deduct as correct solely by looking into it, without usage of external help. It helps to build tests based on those expectations and debug them later, if there is anything going not the way it is supposed to.

```
1  def test_multiple_node_connected_graph(self):
2      # given
3      for i in range(10):
4          TestE2EBFSAlgorithm._request_add_node()
5          if i > 0:
6              requests.post(f'http://127.0.0.1:5000/edges/{i}/{i-1}')
7
8      # when
9      response = requests.post('http://127.0.0.1:5000/algorithm/0/9')
10         .json()
11     gathered_steps = [response['algorithm']['currentStep']]
12     while ((response := requests.put('http://127.0.0.1:5000/
13         algorithm/next_step').json())
14         .get('algorithm').get('currentState') == RUNNING_STATE):
15         gathered_steps.append(response['algorithm']['currentStep'])
16
17     response = response['algorithm']
18
19     # then
20     self.assertEqual(response['path'], [i for i in range(10)])
21     self.assertEqual(gathered_steps, [i for i in range(10)])
22     self.assertEqual(response['currentState'], FINISHED_STATE)
```

Code 3.28: BFS algorithm E2E test

3.4.3 Results

On the figures below, one could see all of the individual tests passing and the time it takes to execute. The amount of tests make it harder for the bugs to slip through the system and makes it easier to indicate them when they appear. It is very dangerous to leave certain parts of the application untested.

: 102 total, 102 passed		11.04 s
		Collapse Expand
test_create_node	passed	0 ms
test_UndirectedGraph		9 ms
UndirectedGraphTest		9 ms
test_add_and_remove_multiple_nodes	passed	2 ms
test_add_edge	passed	0 ms
test_add_multiple_edges	passed	1 ms
test_add_multiple_edges_to_one_edge	passed	1 ms
test_add_multiple_nodes	passed	1 ms
test_add_node	passed	0 ms
test_remove_edge	passed	0 ms
test_remove_first_node_with_an_edge	passed	0 ms
test_remove_multiple_edges	passed	2 ms
test_remove_multiple_nodes	passed	0 ms
test_remove_node	passed	0 ms
test_remove_second_node_with_an_edge	passed	0 ms
test_remove_three_nodes_with_edges_between	passed	0 ms
test_remove_three_nodes_with_edges_between_starting_from_middle	passed	2 ms
test_set_weight_for_the_edge	passed	0 ms

Figure 3.8: Unit tests passing

: 102 total, 102 passed		11.04 s
		Collapse Expand
e2e_tests		10.98 s
test_e2e_bellman_ford_algorithm		1.06 s
TestE2EBellmanFordAlgorithm		1.06 s
test_custom_graph_edge_case_with_negative_number	passed	110 ms
test_custom_graph_with_floating_point_number_weights	passed	99 ms
test_multiple_node_connected_graph	passed	505 ms
test_multiple_node_two_unconnected_graphs	passed	137 ms
test_multiple_node_unconnected_graph	passed	95 ms
test_one_node_graph	passed	31 ms
test_two_node_connected_graph	passed	47 ms
test_two_node_unconnected_graph	passed	34 ms
test_e2e_bfs_algorithm		495 ms
TestE2EBFSAlgorithm		495 ms
test_multiple_node_connected_graph	passed	179 ms
test_multiple_node_two_unconnected_graphs	passed	127 ms
test_multiple_node_unconnected_graph	passed	87 ms
test_one_node_graph	passed	27 ms
test_two_node_connected_graph	passed	43 ms
test_two_node_unconnected_graph	passed	32 ms

Figure 3.9: E2E tests passing

Chapter 4

Conclusion

Summing up, the documentation have covered such topics as graphs, algorithms on them and how one could implement the visualization on them. These graphs are truly a fascinating field to learn, as it brings further understanding in various other areas of life, where one could not have seen any evidence of graph application. The visualization helps people to understand certain topics way easier. At times, it could even wake a curiosity in a man.

Throughout the thesis, there had been many challenges met, which concluded in certain decisions during its development. Some of them were avoided thanks to the proper planning phase, making the components less coupled, others were indicated early in the process and had been solved swiftly as the application of TDD made the development more robust and gradual.

Even though the tool has satisfied its requirements, it is not yet perfect. As one of the big improvements in the future, better control over the algorithm and description of the every step could be implemented. The users might encounter certain difficulties when the visualization tool is used alone, rather it is more effective with the application of theoretical knowledge that one could get by reading relevant books. Therefore, adding more of a description behind the algorithms decisions and steps would ease the process of learning for the user.

During the process, I have learned about such libraries as Flask and React, latter of which had concepts truly hard to grasp. Also, application of my knowledge on graphs helped to see them from the different perspective as well as to achieving comprehension of the algorithms that have been implemented.

Bibliography

- [1] Steven S. Skiena. *The Algorithm Design Manual Second Edition*. Springer-Verlag London Limited, 2008.
- [2] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education, Inc., 2008.
- [3] Thomas H. Cormen et al. *Introduction to Algorithms Fourth Edition*. The MIT Press, 2022.
- [4] *Python Download Page*. May 24, 2024. URL: <https://www.python.org/downloads/>.
- [5] *PyCharm Download page*. May 24, 2024. URL: <https://www.jetbrains.com/pycharm/download>.
- [6] *Node.js*. May 24, 2024. URL: <https://nodejs.org/en>.
- [7] M. Shaw. “Toward higher-level abstractions for software systems”. In: *Data and Knowledge Engineering* (1990).
- [8] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide The 2nd Edition*. Addison-Wesley Professional, 2005.
- [9] Alistair A.R. Cockburn. “Use cases, ten years later”. In: *Humans and Technology* (2002).
- [10] *WSGI page*. May 25, 2024. URL: <https://wsgi.readthedocs.io/en/latest/what.html>.
- [11] *OOP lecture by Kissné Várkonyi Teréz Anna 6.5 Inheritance*. Mar. 19, 2022.
- [12] *React github page*. May 26, 2024. URL: <https://github.com/facebook/react>.
- [13] *React bootstrap page*. May 26, 2024. URL: <https://react-bootstrap.github.io/docs/getting-started/why-react-bootstrap>.

- [14] *D3.js page*. May 26, 2024. URL: <https://d3js.org/what-is-d3>.
- [15] *React query page*. May 26, 2024. URL: <https://tanstack.com/query/latest/docs/framework/react/overview>.
- [16] *MDN web docs HTML*. May 26, 2024. URL: <https://developer.mozilla.org/en-US/docs/Web/HTML>.
- [17] *Flask GeeksForGeeks page*. May 26, 2024. URL: <https://www.geeksforgeeks.org/flask-tutorial/>.
- [18] *TDD GeeksForGeeks page*. May 27, 2024. URL: <https://www.geeksforgeeks.org/test-driven-development-tdd/>.
- [19] *Unit testing GeeksForGeeks page*. May 27, 2024. URL: <https://www.geeksforgeeks.org/unit-testing-software-testing/>.

List of Figures

2.1	Undirected Graph	6
2.2	Directed Graph	6
2.3	Welcome Page	11
2.4	Empty Graph Page	12
2.5	Adding node to graph	13
2.6	Removing node from graph	13
2.7	Removed node from graph	14
2.8	Adding edge to graph	14
2.9	Selecting first node	15
2.10	Selecting second node	15
2.11	Removing edge from graph	16
2.12	Removed edge from graph	16
2.13	Writing down the weight number	17
2.14	Setting the weight for an edge	18
2.15	Set the weight	18
2.16	Selecting from prepared graphs	19
2.17	Cancel ongoing action	19
2.18	Choose the algorithm	20
2.19	Choose the source node	20
2.20	Running the algorithm	21
2.21	After pressing run algorithm	21
2.22	After pressing next step	22
2.23	Completing algorithm	22
2.24	Clearing the path	23
2.25	Directed graph	23
3.1	Use case diagram	25
3.2	Component diagram	27

3.3	Class diagram	28
3.4	Adjacency list representation	29
3.5	Adjacency matrix representation	30
3.6	Custom graphs	36
3.7	Tests folder	49
3.8	Unit tests passing	54
3.9	E2E tests passing	54

List of Codes

3.1	Get graph endpoint	34
3.2	Get graph method in response handler	34
3.3	Change graph type	34
3.4	Change graph in storage	35
3.5	Graph Factory	35
3.6	Custom graph template	36
3.7	Import ready graph response handler	37
3.8	Json to graph method	38
3.9	Adding node	38
3.10	Node class	39
3.11	Generating ids for nodes	39
3.12	Remove node in graph	39
3.13	Id removal of node	40
3.14	Adding edge	40
3.15	Removing edge, undirected	41
3.16	Removing edge, directed	41
3.17	Setting weight for the edge	42
3.18	Choose algorithm handler	43
3.19	Algorithm factory	43
3.20	Running algorithm method	44
3.21	Algorithm's run method	45
3.22	Algorithm step method	46
3.23	CI yaml file	47
3.24	Node test	50
3.25	Undirected graph test	50
3.26	Undirected graph E2E test	51

3.27 BFS algorithm test	52
3.28 BFS algorithm E2E test	53