Oliver Smith
[1810]
[18100]

Graph Theory Interactive Sandbox

Analysis: **2**
Introduction: **2**
Interviews: **3**
Research: **5**
What is a graph? 5
What are optimisation problems? 8
The Three Utilities Problem: 8
Travelling salesman Problem: 8
What are optimisation algorithms? 8
Dijkstra's Algorithm: 9
Prim's Algorithm: 9
Kruskal's Algorithm: 10
Critical Path Analysis Algorithm: 10
Depth First Search: 11
Breadth First Search: 12
Data and Algorithms: **12**
V-Populate Algorithm: 12
E-Populate Algorithm: 13
Depth First Search Algorithm: 14
Breadth First Search Algorithm: 15
Dijkstra's Algorithm: 16
Prim's Algorithm: 17
Kruskal's Algorithm: 18
Critical Path Analysis Algorithm: 18
Model of Proposed System: **19**
Main Menu: 19
Open Graph Screen: 20
Settings Window: 21
Graph Editor: 22
Algorithm Executor: 23
Objectives: **24**
Creating and editing graphs 24
Saving and loading graphs 25
Running Algorithms 25
General Objectives 25

Design: **26**
Introduction: **26**
System Overview: **27**
User Interface: 29
Main Menu: 29
Open Graph Window: 30
Settings Window: 31
Graph Editor: 32
Popup Boxes: 33

Algorithm Executor: 34
Data and Algorithms: 35
Class: Sandbox 35
Class: AlgorithmExecutor 38
Class: Main Menu 42
Notable Algorithms: 43
Dijkstra's Algorithm: 43
Prim's Algorithm: 44
Kruskal's Algorithm: 45
Critical Path Analysis: 46
Drawing edge weights: 47
Drawing digraph arrows: 48
Connected Graphs: 49

Testing: 50
Testing Evidence: 54
Main Menu: 54
Sandbox: 59
Algorithm Executor: 69

Evaluation: 77
Objective Summary: 77
User Feedback: 79
Response to user feedback: 80
End note: 80

Analysis:

Introduction:

I am one of many students who are currently preparing for my A-Level exam in further mathematics with the optional discrete module. During my studies, I noticed that there was a distinct lack of online resources to help students understand and practice the graph optimisation algorithms that are taught within the specification.

My project is aimed to solve this issue by compiling many of the algorithms taught in this module into one, easy to use application. This application will be designed as a tool for both students and tutors to use to demonstrate a variety of optimisation algorithms on their own, custom made graphs.

I believe that this is something that is vital in the understanding and teaching of graph theory at any level.

Interviews:

I sent out a questionnaire to a mixture of further maths teachers who teach graph theory and further maths students to act as an interest check for my application. Here are their responses:

T1: Further Maths Teacher 1
T2: Further Maths Teacher 2
S1: Further Maths Student 1
S2: Further Maths Student 2

Q: How useful would a 'Sandbox' tool be useful for teaching graph theory to students?

T1: Very Useful
T2: Very Useful
S1: Quite Useful
S2: Very Useful

Q: What algorithms would be the most useful to demonstrate?

T1: Dijkstra's algorithm, Prim's algorithm, Kruskal's algorithm, Critical Path Analysis
T2: Dijkstra's algorithm, Prim's algorithm, Kruskal's algorithm, Critical Path Analysis
S1: Critical Path Analysis
S2: Dijkstra's algorithm, Prim's algorithm, Kruskal's algorithm, Critical Path Analysis

Q: What do students struggle with most in Graph Theory?

T1: Remembering what the algorithms do and how to execute them
T2: Understanding and remembering how the algorithms work and how to execute them
S1: Understanding and remembering how the algorithms work and what they do
S2: Understanding and remembering how the algorithms work and are executed

Q: How useful would template graphs for demonstrating algorithms be?

T1: Very Useful
T2: Very Useful
S1: Quite Useful
S2: Quite Useful

Q: How useful would the option to save and edit custom graphs be?

T1: Very Useful
T2: Very Useful
S1: Very Useful
S2: Very Useful

Q: Would you prefer a complex interface with lots of options, or a simpler interface that shows available options more clearly?

T1: Less Options (Simpler GUI)
T2: Less Options (Simpler GUI)
S1: Less Options (Simpler GUI)
S2: Less Options (Simpler GUI)

Q: Which are the better names?

T1: Vertex & Edge
T2: Vertex & Edge
S1: Node & Edge
S2: Vertex & Arc

Q: What other features would be useful?

T1: *None Given*
T2: Showing how the resource levelling diagrams work in CPA
S1: These features are the ones I want the most
S2: The option to undo or remove vertices/edges

From these responses I can confirm that my application would be very useful for both teachers and students studying graph theory.

I have found that Dijkstra's, Prim's, Kruskal's and Critical Path Analysis algorithms would be the most suitable as all of the responses mentioned CPA and the majority mentioned the rest. These algorithms would be better if they were clearly labelled and their result was shown in a straightforward and uncomplicated way.

Template graphs were shown to be a nice addition to the application, but a non essential one, so I will aim to feature them but not make them a major part.

The ability to save graphs and share them with others is essential as it will offer the ability to demonstrate the algorithms quickly and in a format that will remain familiar and exactly as the user desires every time. This opinion is one that is shared amongst everyone that was interviewed.

A simpler GUI was a shared preference for everyone who was interviewed, but I received mixed responses about the preferred terms for the vertices and edges in the graph. I will aim to resolve this issue by allowing the names to be changed in the application in order to avoid any confusion.

I feel as if demonstrating resource levelling diagrams in CPA would fall too far beyond the scope of this project as it would be specific to the OCR A-Level exam board (AQA do not cover this content within their specification) and I would like for this program to have a generic and non-specific focus so that it can be used as a tool for any exam board.

I like the idea of being able to remove vertices and edges from the graph once added as otherwise the user would need to restart their graph just to remove or change one item. I will aim to include this feature as I feel like it would be necessary for the application to feel seamless and have a 'smooth' user experience.

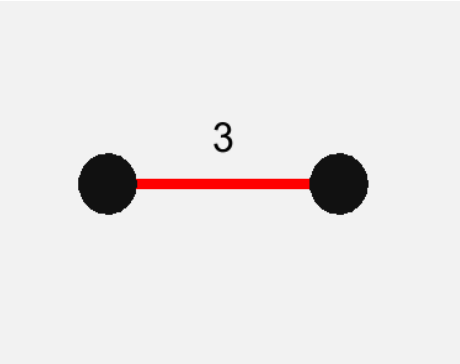
Research:

What is a graph?

Note: *On the basis that I am only going to be implementing support for simple, connected graphs, I will focus my research, and meaning of graph, solely on that.*

A graph is an abstract way of representing a set of data where each element is connected via some relation with at least one other element in the same set.

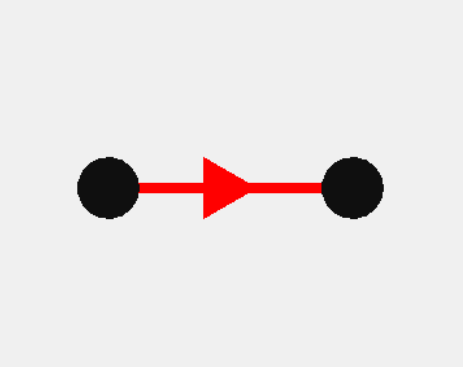
A graph can be represented via an arrangement of vertices (also commonly referred to as nodes or points) and edges (also commonly referred to as arcs or lines) where each vertex represents an element in the set and each edge considers how they are related to one another.



- A graph with two vertices connected via one edge with a weight of 3

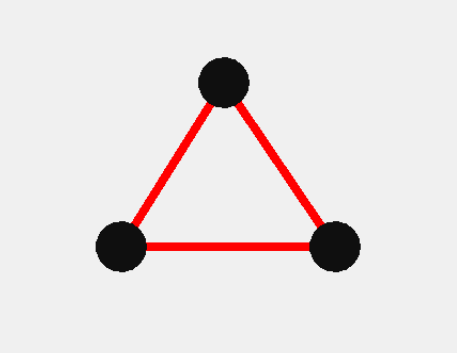
It is possible for an edge to have a ‘weight’ or ‘float’ associated with it. This weight should dictate the ‘cost’ of passing over this edge. If a graph is made up of weighted edges it is known as a ‘network’. Networks are very useful in mathematics and computer science as they can be used to simulate real world problems such as mapping the shortest route between two locations (weight represents distance) or finding the minimum time to complete a set of activities (weight represents time.)

An edge can also be unidirectional where only one direction of travel is permitted. Graphs which contain these types of edges are known as ‘digraphs’. These graphs are particularly useful when it comes to modelling sets of activities as many activities cannot, or would not, be made incomplete once they have been completed.



- A graph with two vertices connected via one unidirectional edge (a digraph)

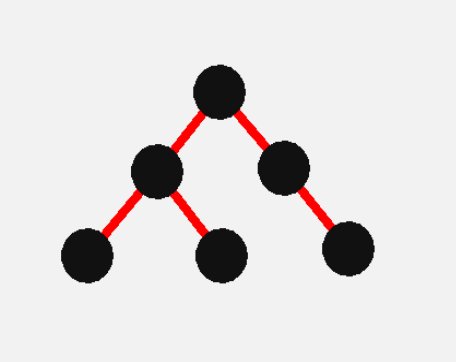
There are many notable types of graphs that can be used for many different purposes. These graphs are mostly used in the form of a subgraph which consists of the original graph with a number of vertices, edges or both removed.



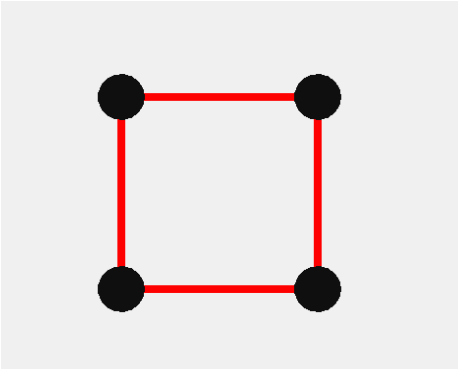
A cycle occurs when it is possible to visit the same vertex twice without passing through any edge more than once. Cycles are quite an important concept as they allow you to get back to where you started without retracing your steps.

- An example of a cycle within a graph

A tree is a type of undirected graph where only one possible path exists between any two vertices. This means that a tree must contain no cycles.
Trees can be used to model heiracal concepts from as simple as family trees to as complex as searching algorithms.



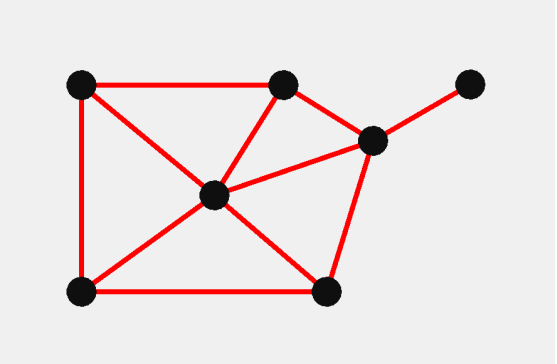
- An example of a tree



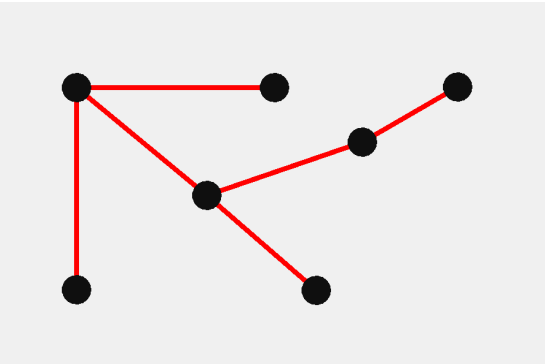
A Hamiltonian graph is a graph that contains a Hamiltonian cycle. A Hamiltonian cycle is a cycle that contains each vertex in the graph exactly once.

This is not to be confused with an Eularian (or traversable) graph where it is possible for a cycle to exist that visits every edge exactly once.

- An example of a Hamiltonian graph



- Graph A



- One possible spanning tree of Graph A

Another common type of subgraph is known as a spanning tree. This subgraph is a tree that visits every possible vertex in the set of vertices that the parent graph consists of.
This idea can be extended into what's called a minimum spanning tree, which is a spanning tree that maintains the smallest possible total weight of the edges (sum of all edge weights combined)

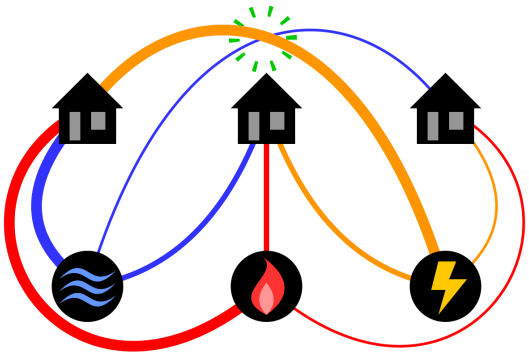
Minimum spanning trees are very useful for a variety of applications such as setting up power lines in a city or, most notably in mathematics, finding the lower bound for the travelling salesperson problem.

What are optimisation problems?

Optimisation problems consist of finding the ideal solution from all possible solutions. In the matter of graph theory, this could include such problems as:

- Finding the shortest path between two vertices
- Finding the shortest path to visit all vertices in one cycle
- Finding the shortest path to visit all edges in once cycle

Some more famous problems that exist:



The Three Utilities Problem:

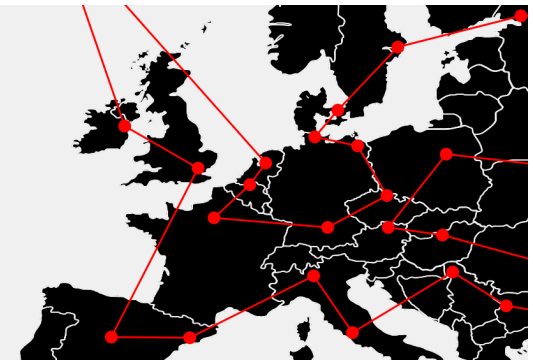
The problem consists of linking the three houses with the three utilities without crossing any paths.

This problem is proven to be impossible by Kuratowski's theorem that a graph is planar if it is not isomorphic to the graph K3,3; which it is.

Travelling salesman Problem:

The travelling salesman problem is the problem of finding the shortest path of a cycle that passes through every vertex at least once and ends at the vertex it started at.

This problem still has no tractable solution.



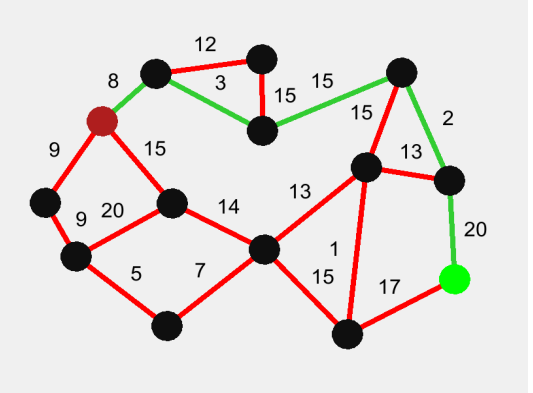
What are optimisation algorithms?

Optimisation algorithms serve to provide a heuristic solution to an optimisation problem. Some examples of optimisation algorithms include:

Dijkstra's Algorithm:

Dijkstra's algorithm is used to find the shortest path between two vertices on a graph.

The algorithm functions by providing a temporary and a permanent label for each vertex on the graph and then executes each step as follows:

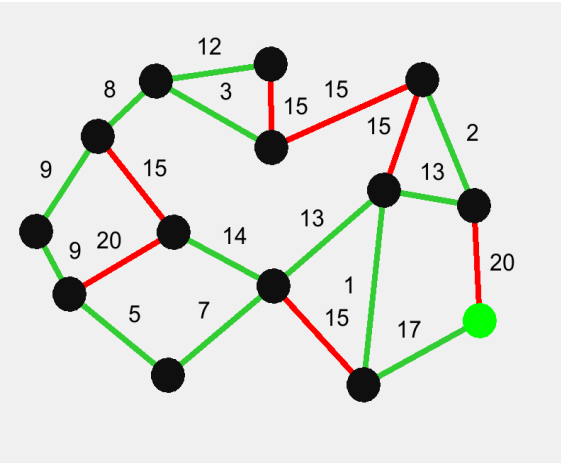


STEP 1 : Make the permanent label of the given start vertex equal to 0.

STEP 2 : For each vertex without a permanent label connected to the current vertex, take the current vertex's permanent label plus the weight of the edge that connects them and if this value is less than the temporary label at the connected vertex, write this value as the new temporary label.

STEP 3 : Choose the connected vertex that is not yet permanent which has the smallest temporary label and make its permanent label equal to its temporary label. This is the vertex you will look at next.

STEP 4 : If every vertex is now permanent, or if the end vertex is permanent the algorithm has finished and the shortest path is the end vertex's permanent label; otherwise return to STEP 2.



Prim's Algorithm:

Prim's algorithm is used to find the minimum spanning tree of a graph.

STEP 1 : Choose a starting vertex.

STEP 2 : Choose the vertex connected to a visited vertex that has the smallest edge weight and isn't already visited. This is the vertex you will look at next.

STEP 3 : If a spanning tree is obtained the algorithm is complete; otherwise return to STEP 2.

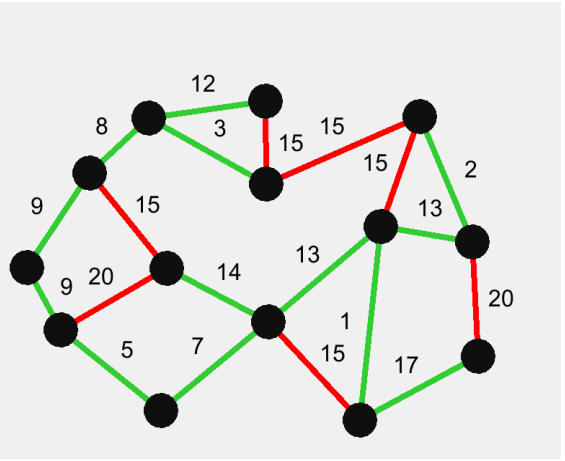
Kruskal's Algorithm:

Kruskal's algorithm is used to find the minimum spanning tree of a graph.

STEP 1 : Start by removing all edges from the graph but leave the vertices.

STEP 2 : Choose the edge with the smallest weight
joining an unvisited vertex and adding it to the graph.

STEP 3 : If a spanning tree is obtained the algorithm is complete; otherwise return to STEP 2.



Both Kruskal's and Prim's algorithms will often produce the same results although it is possible for there to be more than one minimum spanning tree within a graph.

Critical Path Analysis Algorithm:

The critical path analysis is unique on the types of graphs that it can be executed on. These graphs are what are called activity networks and are typically used for scheduling large projects that can be broken down into smaller, more manageable tasks. An activity network is a type of graph where the edges represent tasks to be completed opposed to the vertices, which represent the stage of production. Activity networks are required to be digraphs and contain no cycles because, as mentioned earlier, once a task is completed, it cannot be made incomplete and there should be no way of getting back to a previous stage of production.

The algorithm for critical path analysis (or CPA) is completed in two parts; the forward pass and the backward pass.

The forward pass finds the earliest event time for that vertex / stage of production. The earliest event time is the earliest possible time that the stage of production can be reached.

The backwards pass finds the latest event time for that vertex. The latest event time represents the latest time a stage in production can be reached without affecting the total time that the project takes to be completed.

The algorithm executes as follows:

STEP 1 : Assign an earliest and latest event time event for each vertex in the graph and set both to 0 for the starting vertex.

STEP 2 : For each vertex connected to the current vertex, take the earliest event time of the current vertex and add it to the edge weight. If this value is greater than the earliest event time of the connected vertex, write this value as the new earliest event time.

STEP 3 : Repeat STEP 2 for every connected vertex until there are no more connections left to be explored.

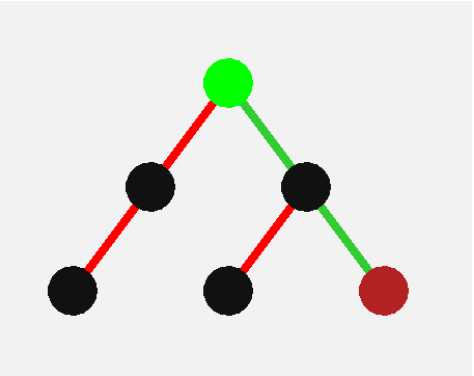
STEP 4 : for the vertex with the latest earliest event time, set the latest event time to the earliest event time.

STEP 5 : Flip all of the directions of the edges

STEP 6 : For each vertex connected to the current vertex, take the latest event time of the current vertex and subtract the edge weight. If this value is less than the latest event time of the connected vertex, write this value as the new earliest event time.

STEP 7 : Repeat STEP 6 for every connected vertex until there are no more connections left to be explored.

Depth First Search:



Depth first search is a tree traversal algorithm that is often used as a base for building other graph traversal algorithms.The algorithm uses a stack to keep track of the last vertex visited, and a list to hold the names of vertices that have already been visited.

Depth first search (or DFS) is used to find all of the connected vertices in a graph from a start vertex.

How to execute the algorithm:

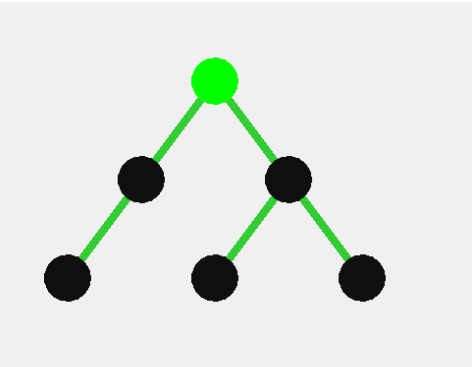
STEP 1 : Add the start vertex to the visited list

STEP 2 : For each connection to the current vertex that is not in the visited list, add it to the visited list and add the current vertex to the stack. The vertex most recently added to the visited list is the new current vertex

STEP 3 : Repeat STEP 2 until there are no more vertices left to be visited then remove the top vertex from the stack. The new vertex on the top of the stack is the new current vertex

STEP 4 : Repeat STEP 3 until the stack is empty and there are no more connections left to be explored.

Breadth First Search:



Breadth first search is another tree traversal algorithm that is often used as a base for building other graph traversal algorithms.The algorithm uses a queue to keep track of the vertices left to visit, and a list to hold the names of vertices that have already been visited.

Breadth first search (or BFS) is used to find all of the connected vertices in a graph from a start vertex.

How to execute the algorithm:

STEP 1 : Add the start vertex to the queue

STEP 2 : For each connection to the current vertex that is not in the visited list, add it to the queue. Once every connected vertex has been added, remove the current vertex from the queue and add it to the visited list.

STEP 3 : Repeat STEP 2 until the queue is empty

Data and Algorithms:

Note: *The edges in the graph should be stored using two arrays, one for the start vertex and one for the end vertex, but it's important to note that undirected edges will have to travel in both directions and so will need to be stored twice; once for each direction.*

V-Populate Algorithm:

The V-Populate algorithm will be called when the user wishes to add a number of vertices to the graph at random on the graph editor screen.

Inputs from user:

- Number of vertices to be added to the screen

Other data to be retrieved:

- The maximum amount of vertices allowed on screen
- The coordinates of other vertices on screen

How the data will be stored:

- The Coordinates of the vertices should be stored in Lists as float data
- The radius of the vertex should be a float variable that is dependant upon screen size and the zoom level in the settings window

The algorithm will work by repeating iteratively for the number of vertices to be added to the screen and picking a random X and Y coordinate between 0 and the maximum width and height of where vertices can be placed. The program will then compare these random coordinates with the coordinates of existing vertices and if the distance between the coordinates is less than twice the radius of the vertex (using pythagoras' theorem) it will repeat until it finds a valid set of coordinates, and then it will permanently add a vertex to the graph with those coordinates.

E-Populate Algorithm:

The E-Populate algorithm will be called when the user wishes to add a number of edges to fully connect the graph.

Inputs from user:

- The maximum value of an edge weight (from 0 - X)
- Whether the edge weight should be an integer value

Other data to be retrieved:

- All existing edges in the graph (start and end vertex)
- The number of vertices in the graph

How the data will be stored:

- The edges should be stored in two lists; one ‘from’ list and one ‘to’ list

The algorithm will work by repeating until the graph is fully connected (which will be found using depth first search). While the graph is not fully connected, the algorithm will repeat for every vertex finding another vertex that is close to it. It will do this by ‘scanning’ within a circular area around the vertex until another vertex that it is not already connected to is found. If no vertex can be found within the given area, the area is extended by a fixed amount before checking again. Once a valid vertex has been found, an edge that starts from the current vertex and ends at the found vertex will be permanently added to the graph.

The weight of the edge will be chosen at random between 0 and the value that the user has entered. If the user has chosen for the edge weight to not be integer values, a random number between 0 and 1 will be subtracted from the integer edge weight (ensuring that the weight does not fall below 0).

Depth First Search Algorithm:

Depth first search is used to explore the current graph from a starting vertex. It will explore ‘depth first’ meaning that it will continue along a path until it reaches a point where it cannot continue. It will then return to an earlier vertex and explore a different path.

Inputs from user:

- The start vertex

Other data to be retrieved:

- All existing edges in the graph (start and end vertex)
- The number of vertices in the graph

How the data will be stored:

- The edges should be stored in two lists; one ‘from’ list and one ‘to’ list
- There should be a list for the visited vertices
- There should be a list that functions as a stack for the connections

The algorithm will run recursively using a stack to keep track of the last vertex visited, and a list to hold the names of vertices that have already been visited. It will start by adding the start vertex to the visited list. For each connection to the current vertex that is not in the visited list, it should add it to the visited list and add the current vertex to the stack. The vertex most recently added to the visited list is the new current vertex. The algorithm should repeat this step until there are no more vertices left to be visited and then it should remove the top vertex from the stack. The new vertex on the top of the stack is the new current vertex. These steps should then be repeated until the stack is empty and there are no more connections left to be explored.

Breadth First Search Algorithm:

Breadth first search (or BFS) is used to find all of the connected vertices in a graph from a start vertex.The algorithm uses a queue to keep track of the vertices left to visit, and a list to hold the names of vertices that have already been visited.

Inputs from user:

- The start vertex

Other data to be retrieved:

- All existing edges in the graph (start and end vertex)
- The number of vertices in the graph

How the data will be stored:

- The edges should be stored in two lists; one ‘from’ list and one ‘to’ list
- There should be a list for the visited vertices
- There should be a list that functions as a queue for the vertices

The algorithm will start by adding the start vertex to the queue. For each connection to the current vertex that is not in the visited list, it will be added to the queue. Once every connected vertex has been added, remove the current vertex from the queue and add it to the visited list. It will then repeat this step until the queue is empty.

Dijkstra's Algorithm:

Dijkstra's algorithm is to find the shortest path between two points. I will model my algorithm off a modified breadth first search algorithm.

Inputs from user:

- The start and end vertices

Other data to be retrieved:

- All existing edges in the graph (start and end vertex)
- All existing edge weight in the graph

How the data will be stored:

- The edges should be stored in two lists; one 'from' list and one 'to' list
- There should be a list for the visited vertices
- There should be a list for the temporary labels
- There should be a list for the permanent labels
- There should be a list that functions as a queue for the vertices
- There should be an array for the shortest path
- There should be a list for the previous vertex

Once the user selects Dijkstra's algorithm to be run on their graph, the program should prompt them to enter their start and end vertices by clicking on them. Once these two vertices have been chosen, the start vertex should be added to the visited list and the permanent and temporary labels should be set to 0.

The connections to the current vertex, that aren't in the visited list, should be stored in a list that will act as a priority queue with the smallest wedge weight first (for optimisation). For each of these vertices in the queue, if the temporary label of the vertex that came before it plus the edge weight is less than the temporary label at that vertex, the algorithm should write this value as the new temporary label. The algorithm should then choose a connected vertex that is not yet in the visited list which has the smallest temporary label and make its permanent label equal to its temporary label. This vertex will be added to the visited list and made the next current vertex.

The algorithm should repeat through these steps until the end vertex is added to the visited list, where it will then read back through the previous vertex list creating a shortest path.

The shortest path and permanent label of the end vertex should be displayed on screen along with the shortest path being outlined on the graph itself.

Prim's Algorithm:

Prim's algorithm is used to find the minimum spanning tree of a graph.

Inputs from user:

- The start vertex

Other data to be retrieved:

- All existing edges in the graph (start and end vertex)
- All existing edge weight in the graph

How the data will be stored:

- The edges should be stored in two lists; one 'from' list and one 'to' list
- There should be a list for the visited vertices
- There should be two lists for the used edges (from and to)

The algorithm should start by adding the starting vertex to the visited list. It should then make a list of all of the vertices that aren't in the visited list that are connected to the visited vertices along with their weights. The algorithm should select the connection with the shortest edge weight and add that vertex to the visited list and add the edge's 'to' and 'from' vertices to the 'to' and 'from' lists respectively. This process should repeat until every vertex has been visited.

Once every vertex has been visited, the algorithm should display the edges of the minimum spanning tree and total weight of the tree along with the tree being outlined on the graph.

Kruskal's Algorithm:

Kruskal's algorithm is used to find the minimum spanning tree of a graph.

Inputs from user:

- N/A

Other data to be retrieved:

- All existing edges in the graph (start and end vertex)
- All existing edge weight in the graph

How the data will be stored:

- The edges should be stored in two lists; one 'from' list and one 'to' list
- There should be a list for the visited vertices
- There should be two lists for the used edges (from and to)

The algorithm should start by sorting all of the existing edge weight from smallest to largest. It should then add the smallest edge that doesn't already exist and doesn't create a cycle. It should repeat these steps until all of the vertices are in the visited list.

Once every vertex has been visited, the algorithm should display the edges of the minimum spanning tree and total weight of the tree along with the tree being outlined on the graph.

Critical Path Analysis Algorithm:

Critical path analysis is used to find the earliest and latest event times for scheduling in an activity network.

Inputs from user:

- The start vertex

Other data to be retrieved:

- All existing edges in the graph (start and end vertex)
- All existing edge weight in the graph

How the data will be stored:

- The edges should be stored in two lists; one ‘from’ list and one ‘to’ list
- There should be a list for the visited vertices
- The earliest and latest event times should be stored in an array

The algorithm should assign an earliest and latest event time event for each vertex in the graph and set both to 0 for the starting vertex. For each vertex connected to the current vertex, it should take the earliest event time of the current vertex and add it to the edge weight. If this value is greater than the earliest event time of the connected vertex, it should write this value as the new earliest event time. The algorithm should repeat this process for every connected vertex until there are no more connections left to be explored.

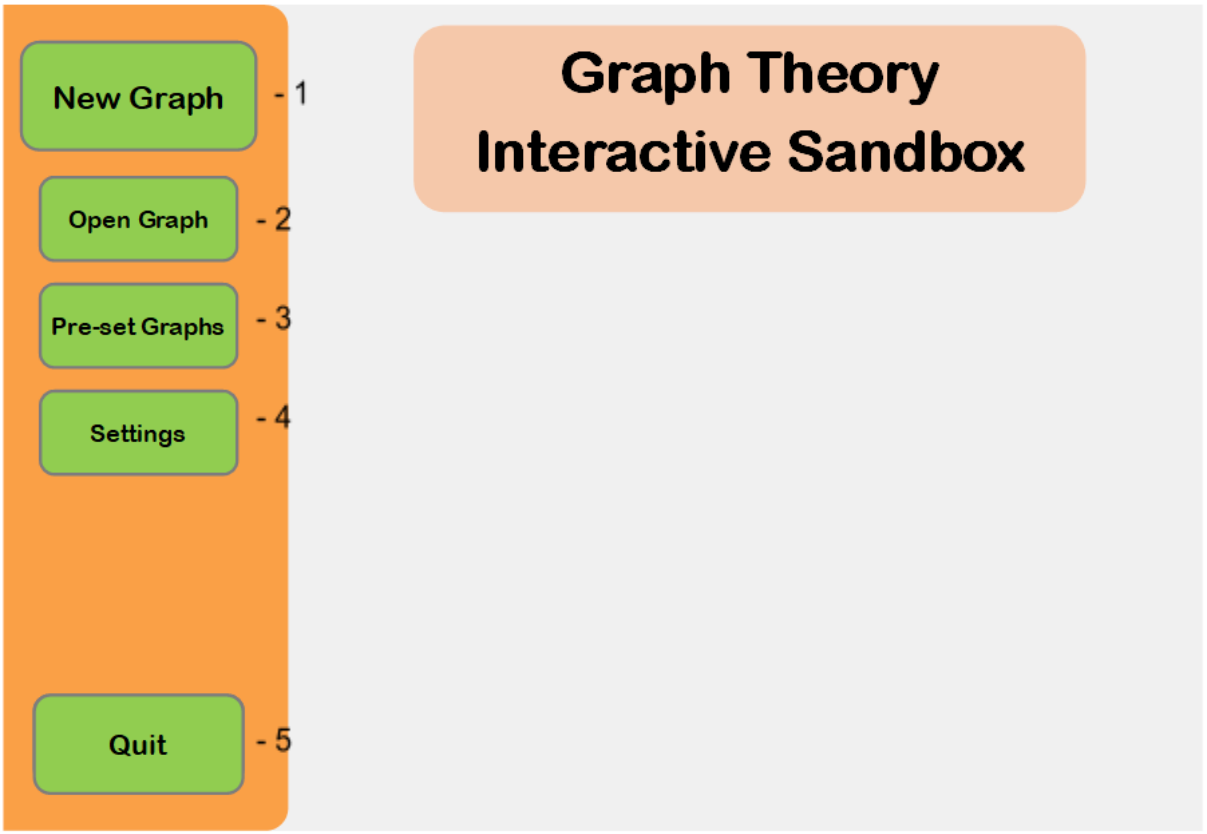
The vertex with the latest earliest event time’s latest event time should then be set to the earliest event time. All of the edge’s direction should then be flipped and for each vertex connected to the current vertex, it should take the latest event time of the current vertex and subtract the edge weight. If this value is less than the latest event time of the connected vertex, it should write this value as the new earliest event time. The algorithm should repeat this process for every connected vertex until there are no more connections left to be explored.

Once the backwards pass has been completed, the algorithm should display the edges and total weight of the critical path along with the critical path, earliest and latest event times being outlined on the graph.

I believe that these algorithms would provide the most functionality as they are some of the most widely taught algorithms in the study of graph theory and act as a basis for many other optimisation algorithms.

Model of Proposed System:

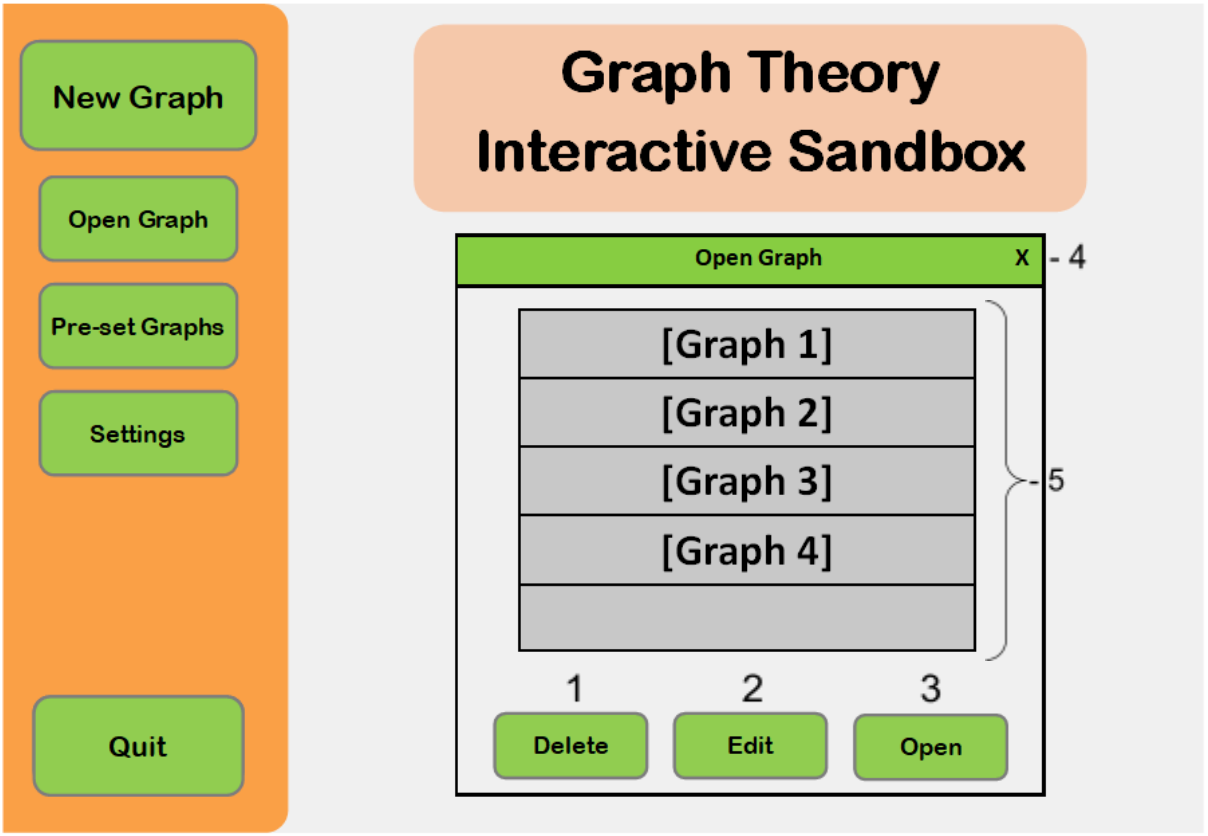
Main Menu:



1 - Takes you to the graph editor screen
2 - Opens the 'Open Graph' window
3 - Opens the 'Settings' window
4 - Takes the user to a list of pre-set graphs that they can open
5 - Exits the application

The main menu should be the first screen that the user sees when opening the application. The user should be able to get access to any of the program's features straight from the main menu; This includes starting a new graph from scratch, opening an existing graph from a file, opening a pre-set graph from a file, changing the settings and exiting the program.

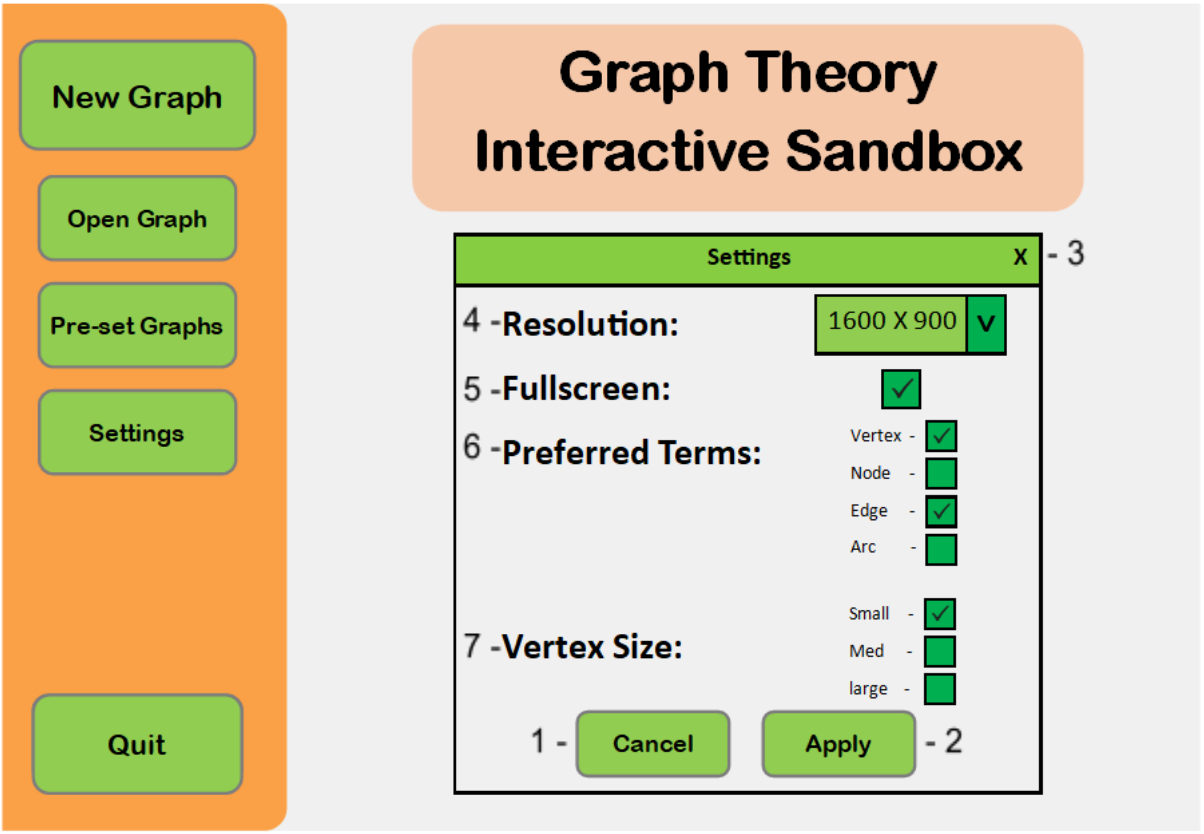
Open Graph Screen:



1 - Deletes the currently selected graph file from the ‘Saved Graphs’ folder
2 - Opens the currently selected graph file in the graph editor
3 - Opens the currently selected graph file in the algorithm executor
4 - Closes the Open Graph window
5 - The list of graphs in the ‘Saved Graphs’ folder that can be selected

The open graph screen should be opened by pressing on the open graph button on the sidebar in the main menu. It should display a list of graph files that are stored within a saved graphs folder. These files can be opened in the edit graph screen, opened in the algorithm executor screen or deleted, all from within this window.

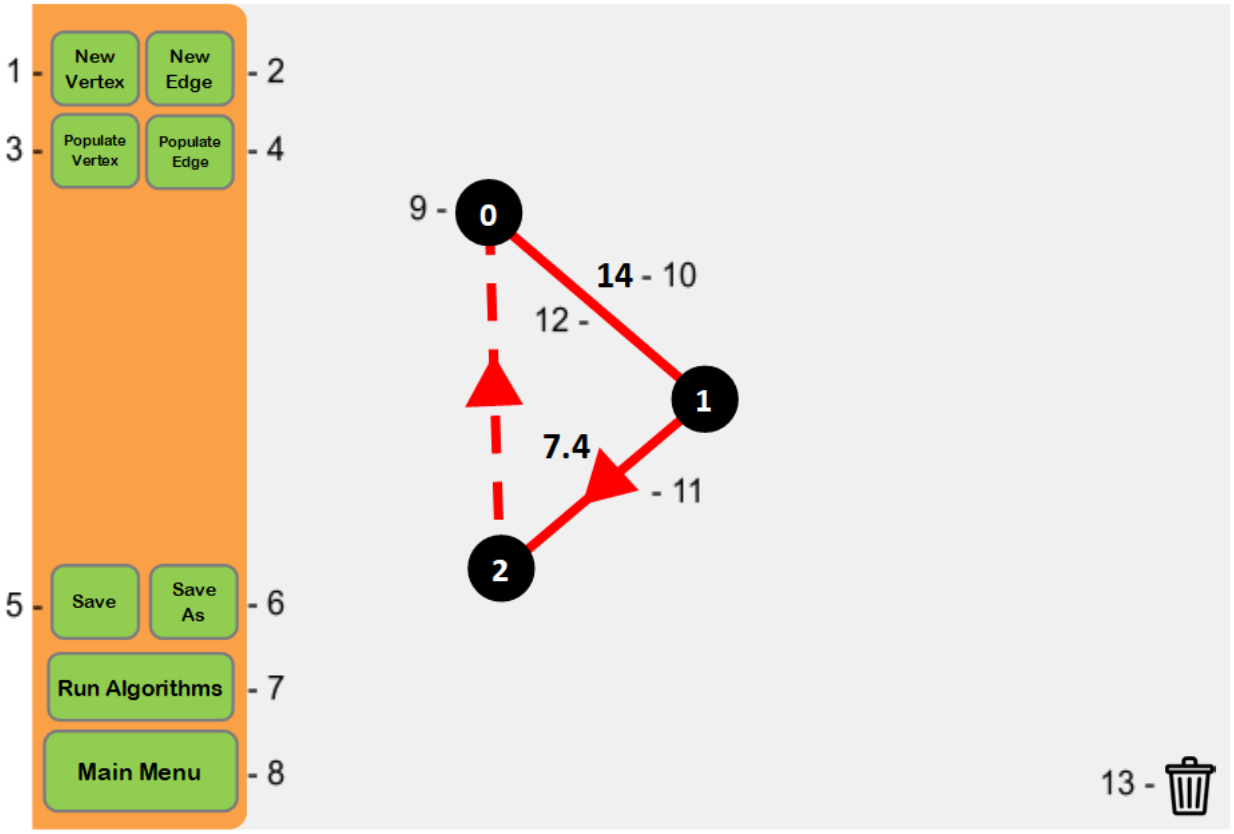
Settings Window:



1 - This button closes the window without saving any changes
2 - This button applies any changes made immediately
3 - Closes the Settings window
4 - A drop down box where you can select the windowed resolution of the application
5 - This toggles fullscreen
6 - Checkboxes where you can change the preferred name of vertices and edges
7 - Checkboxes where you can change the display size of the vertex

The settings screen should be opened by pressing on the settings button on the sidebar in the main menu. It should contain a variety of settings that can be changed such as the resolution, the screen type, the preferred terms for vertices and edges and the vertex size. These settings should be able to be changed via a selection of drop down menus and checkboxes. These settings should be saved to the device so that they can be recalled the next time the application is opened.

Graph Editor:

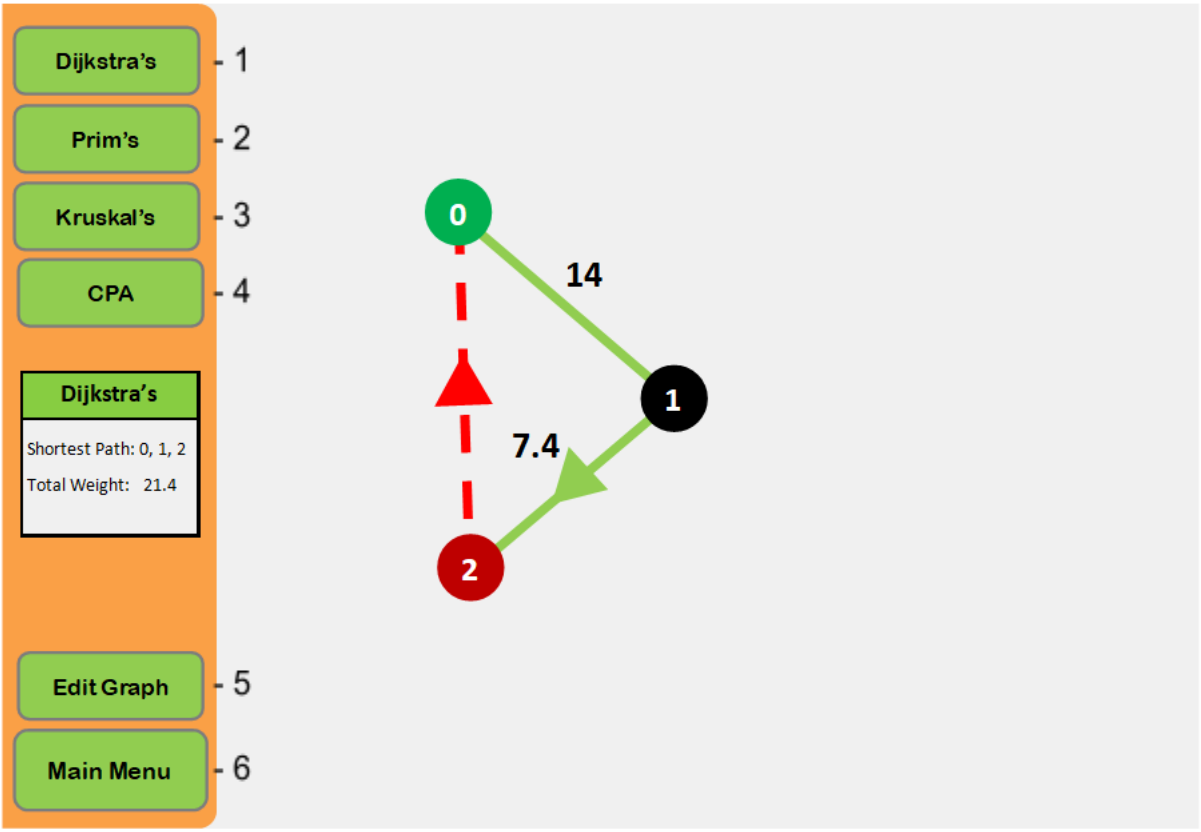


1 - Creates a new vertex that the user can place on the screen
2 - Creates a new edge that the user can place on the screen
3 - Opens the V-Populate box
4 - Opens the E-Populate box
5 - Saves the graph under its current file name
6 - Saves the graph into a new file with a user defined name
7 - Opens the current graph in the algorithm executor
8 - Exits to the main menu
9 - A moveable vertex that can be dragged and dropped by the user
10 - The weight of the edge
11 - A directed edge
12 - An undirected edge
13 - Clears the graph of all vertices and edges

The graph editor is where the user can create or edit graphs from scratch or from an existing graph file. The user can create new vertices and edges that can be added to the graph by clicking where they want to be placed.

The user can also use the V-Populate and E-Populate function to have the application create a fully connected graph for them.

Algorithm Executor:



1 - Runs Dijkstra's algorithm on the current graph
2 - Runs Prim's algorithm on the current graph
3 - Runs Kruskal's algorithm on the current graph
4 - Runs the CPA algorithm on the current graph
5 - Opens the graph in the graph editor
6 - Returns to the main menu

The algorithm executor is where the user will be able to run any of the available algorithms on their graph.

The user will be unable to edit the graph from this screen but they can return to the graph editor by pressing the edit button where the graph will be opened.

Objectives:

My application will be designed to act as an educational tool for use by primarily students and teachers to simulate how graphing algorithms are executed on networks. The program should provide a model solution for a chosen algorithm on any graph that the user has created; as long as the algorithm is relevant to the graph. The main goal of this application should be to aid students' understanding of the algorithms and how they are executed.

Creating this program will require a great deal of knowledge about the algorithms and a very solid understanding of the basic principles on which they function.

1. **Creating and editing graphs**
    - **1.1.** The user should have the choice to create either unidirectional or bidirectional graphs
    - **1.2.** The user should be able to edit these graphs by adding new vertices and edges with custom weights
    - **1.3.** The program should be able to to create its own connected graph with a user chosen number of vertices and random edge weights
        - **1.3.1.** The user should be able to choose a number of vertices between 1 and 400
        - **1.3.2.** The user should be able to choose the range of the weights between 0 and 20 and whether they are integer or float values
    - **1.4.** The user should be able to create as many graphs as they want from scratch or from an existing graph

2. **Saving and loading graphs**
    - **2.1.** The user should be able to save their own graph with custom file names
        - **2.1.1.** The program should store:
            - Whether the graph is unidirectional or bidirectional
            - The coordinates of each vertex
            - Where each edge starts and ends
            - The weight of each edge
    - **2.2.** The user should be able to edit any graph and overwrite that graph's save file
    - **2.3.** The user should be able to share their own graphs with other users by sending them the graph's file

3. **Running Algorithms**
    - **3.1.** The user should be able to run Dijkstra's algorithm on any (valid) graph
        - **3.1.1.** The shortest path should be displayed on the screen in a different colour and with its total weight
    - **3.2.** The user should be able to run Prim's algorithm on any (valid) graph
        - **3.2.1.** The minimum spanning tree should be displayed on the screen in a different colour with its total weight

3.3. The user should be able to run Kruskal's algorithm on any (valid) graph

3.3.1. The minimum spanning tree should be displayed on the screen in a different colour with its total weight

3.4. The user should be able to run the CPA algorithm on any (valid) graph

3.4.1. The critical path should be displayed on screen along with the earliest and latest event time for each vertex and the earliest completion time

4. General Objectives

4.1. The GUI should be simple and user friendly

4.2. The Algorithms should be fast and effective

4.3. The user should be notified if the chosen algorithm cannot be executed on their graph

Design:

Introduction:

The program that I have chosen to create will be a graph theory ‘sandbox’ where the user will get free reign to create any graph that they want and be able to run a selection of algorithms on it.

The algorithms that i would like to demonstrate within my program include:

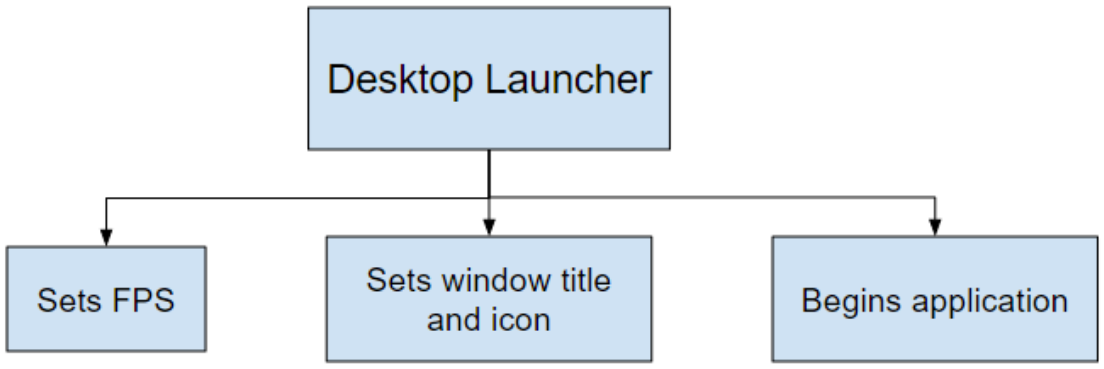
- Dijkstra's Algorithm (shortest path)
- Prim’s algorithm (minimum spanning tree)
- Kruskal’s Algorithm (minimum spanning tree)
- Critical Path Analysis

I would like for the solution to these algorithms to be displayed in a simple and straightforward manner, clearly showing the relevant information graphically and written.

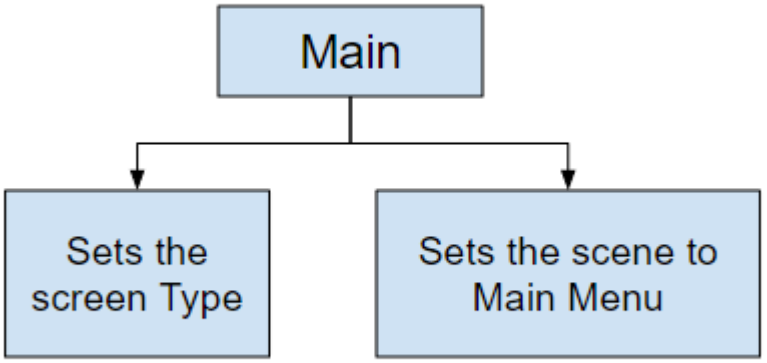
I would like for the user to be able to have the choice of creating either digraphs or undirected graphs, and I would also like the option for the computer to generate a graph from a set of given inputs by the user (i.e number of vertices & bounds for the weight of edges)

I have chosen to develop this program in java using the libGDX development framework with the optional extension of the scene 2D widget/UI Library. This will allow me to draw basic shapes and images to the screen as well as use the UI interfaces included within the provided libraries.

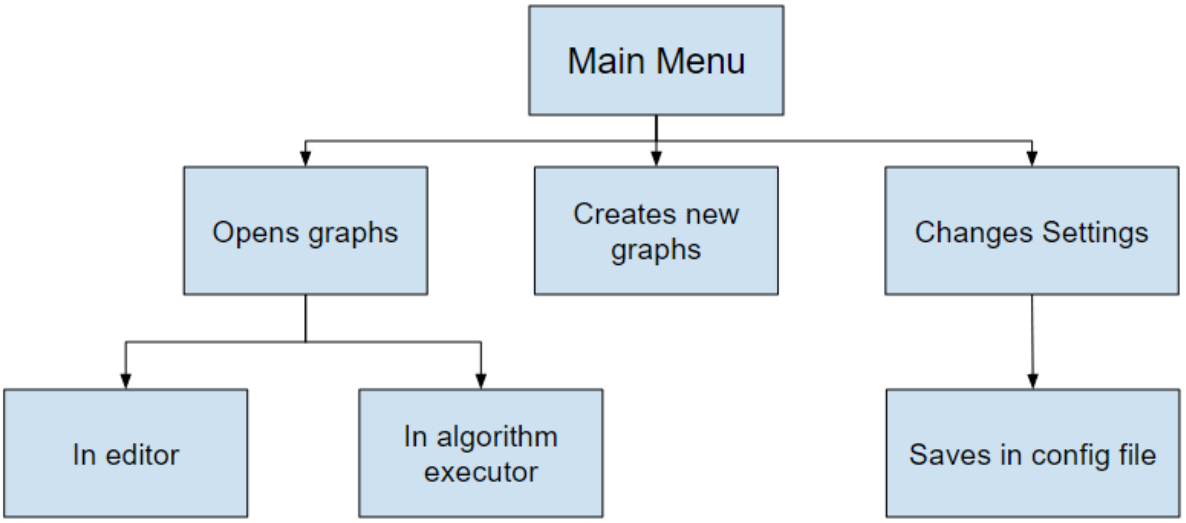
System Overview:



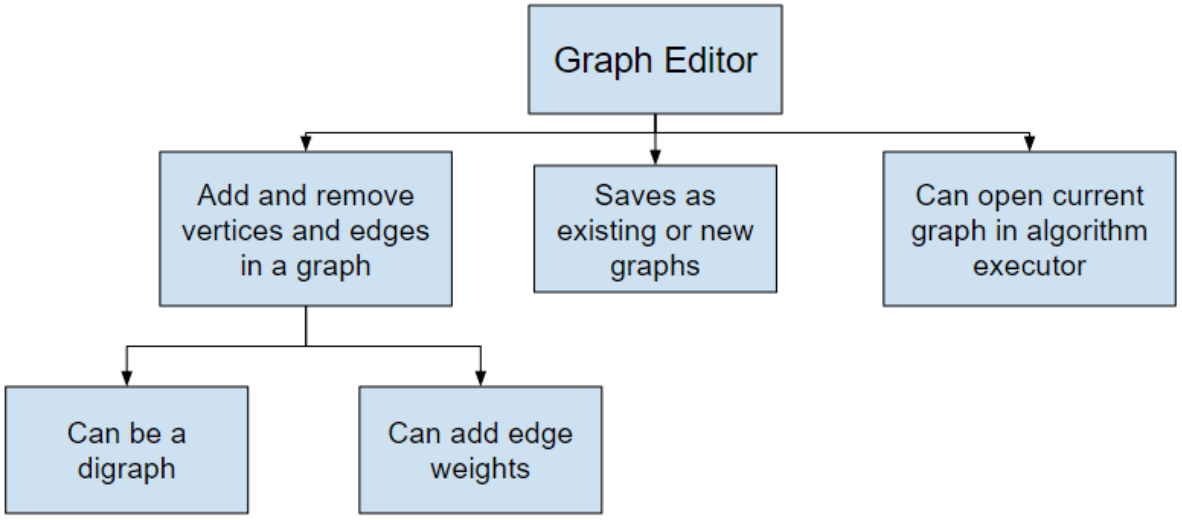
The desktop launcher begins the application.



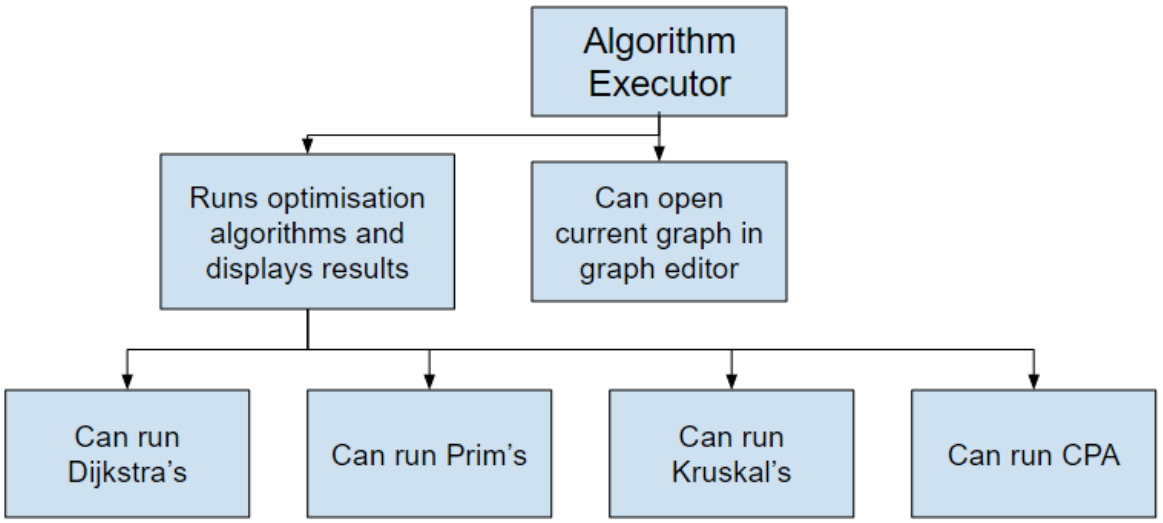
The main class sets up the screen type and sets the scene to the main menu.



The main menu can open graphs in the editor or algorithm executor, create new graphs or change the settings and save the results to a config file.



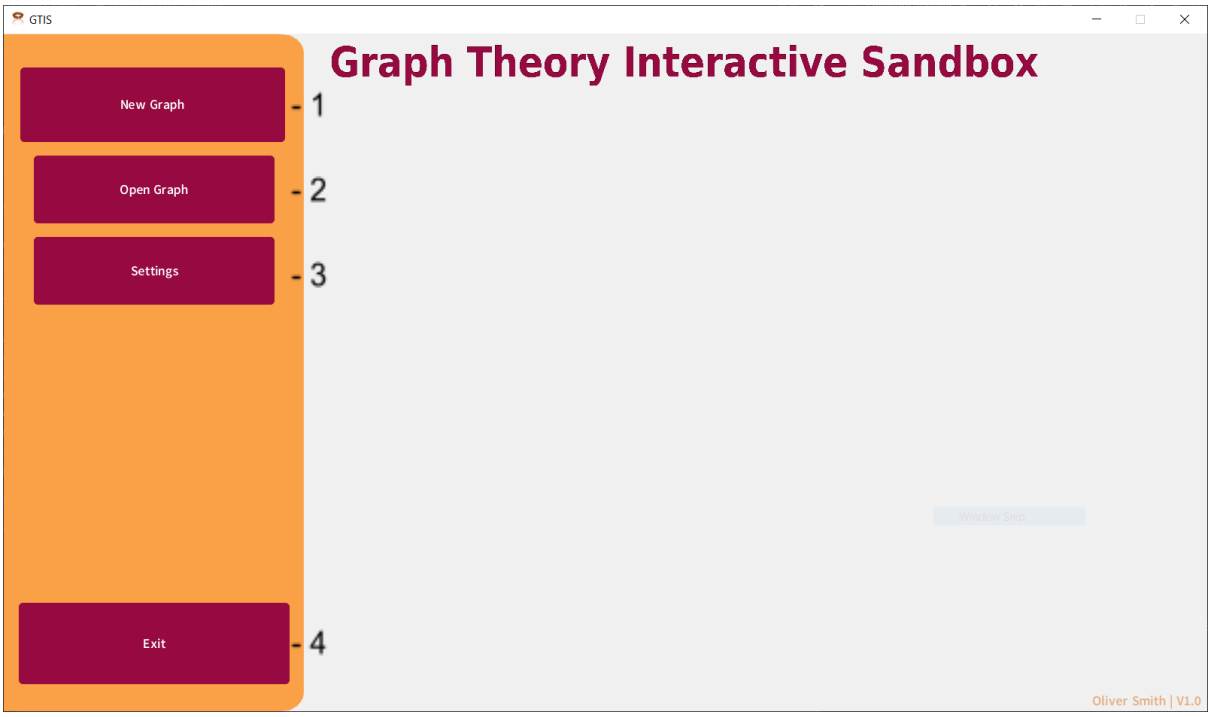
The graph editor, or sandbox, is where the user can add and remove vertices and edges from graphs. These graphs can be saved into a file and can be opened into the algorithm executor.



The algorithm executor is where the user can execute algorithms on the current graph that is loaded. They can run Dijkstra's, prim's, Kruskal's and CPA on their own graph as well as opening the current graph in the graph editor.

User Interface:

Main Menu:

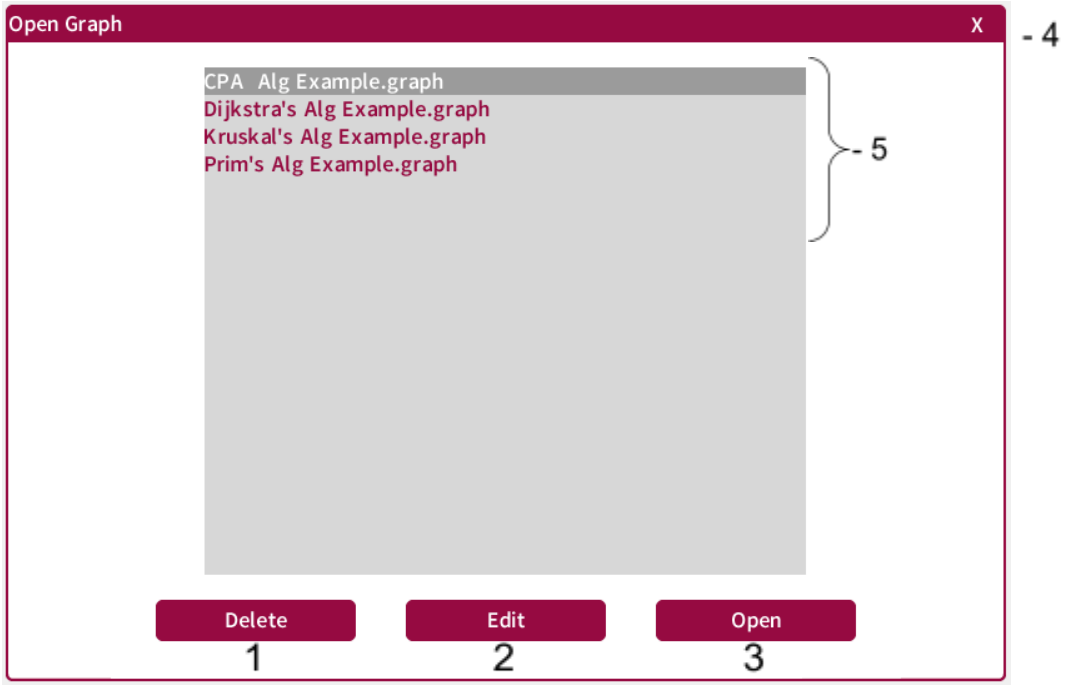


1 - Takes you to the graph editor screen
2 - Opens the ‘Open Graph’ window
3 - Opens the ‘Settings’ window
4 - Exits the application

The main menu is the first screen you see when opening the application. From this screen you can get access to the graph editor and algorithm executor for any graph stored in the Saved Graphs folder or create a new graph from scratch by pressing the new graph button.

You can also get access to the settings window where any changes you make will be stored in the config file.

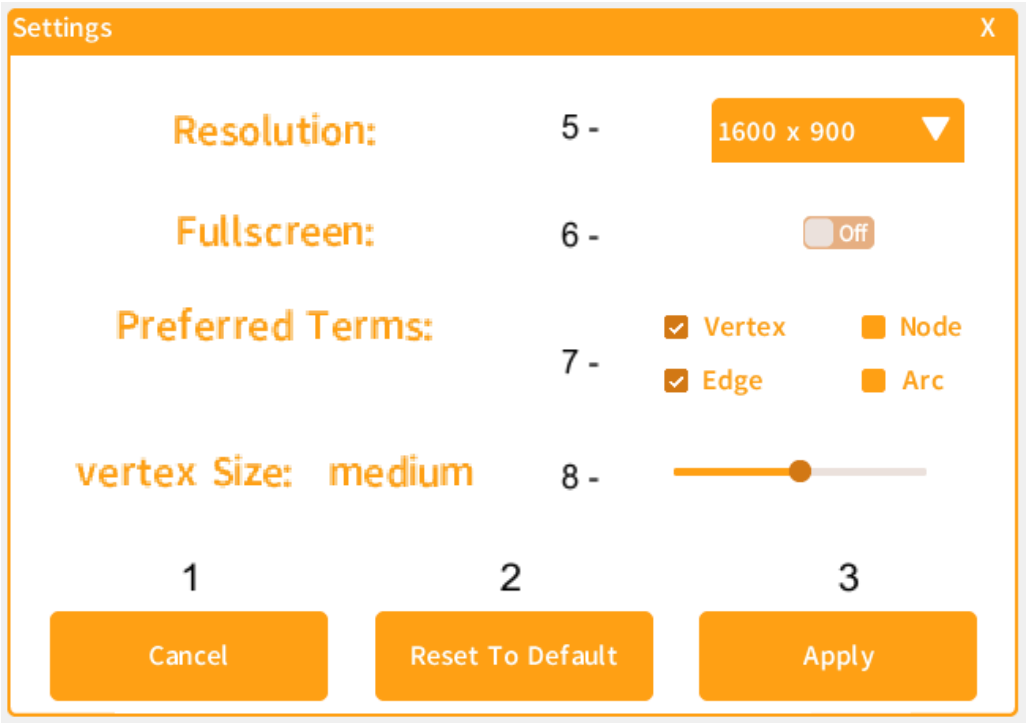
Open Graph Window:



1 - Deletes the currently selected graph file from the ‘Saved Graphs’ folder
2 - Opens the currently selected graph file in the graph editor
3 - Opens the currently selected graph file in the algorithm executor
4 - Closes the Open Graph window
5 - The list of graphs in the ‘Saved Graphs’ folder that can be selected

From the open graph window you can open any graph file stored within the Saved Graphs folder in the graph editor or algorithm executor as well as delete the file by pressing the delete button.

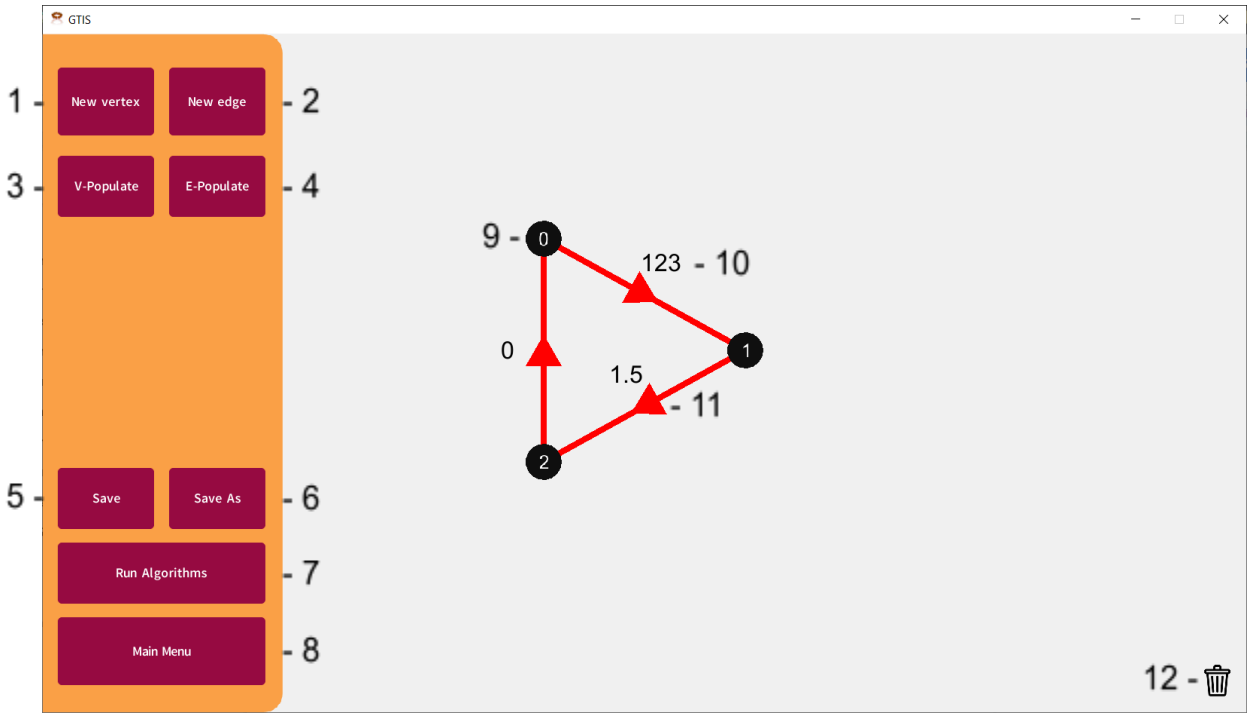
Settings Window:



1 - This button closes the window without saving any changes
2 - This button resets all of the settings to the default options
3 - This button applies any changes made immediately
4 - Closes the Settings window
5 - A drop down box where you can select the windowed resolution of the application
6 - This toggles fullscreen
7 - Checkboxes where you can change the preferred name of vertices and edges
8 - A slider where you can change the display size of the vertex

Any changes that the user makes by pressing the apply button will be saved locally into a config file where the program will be able to restore these changes the next time the application is opened.

Graph Editor:



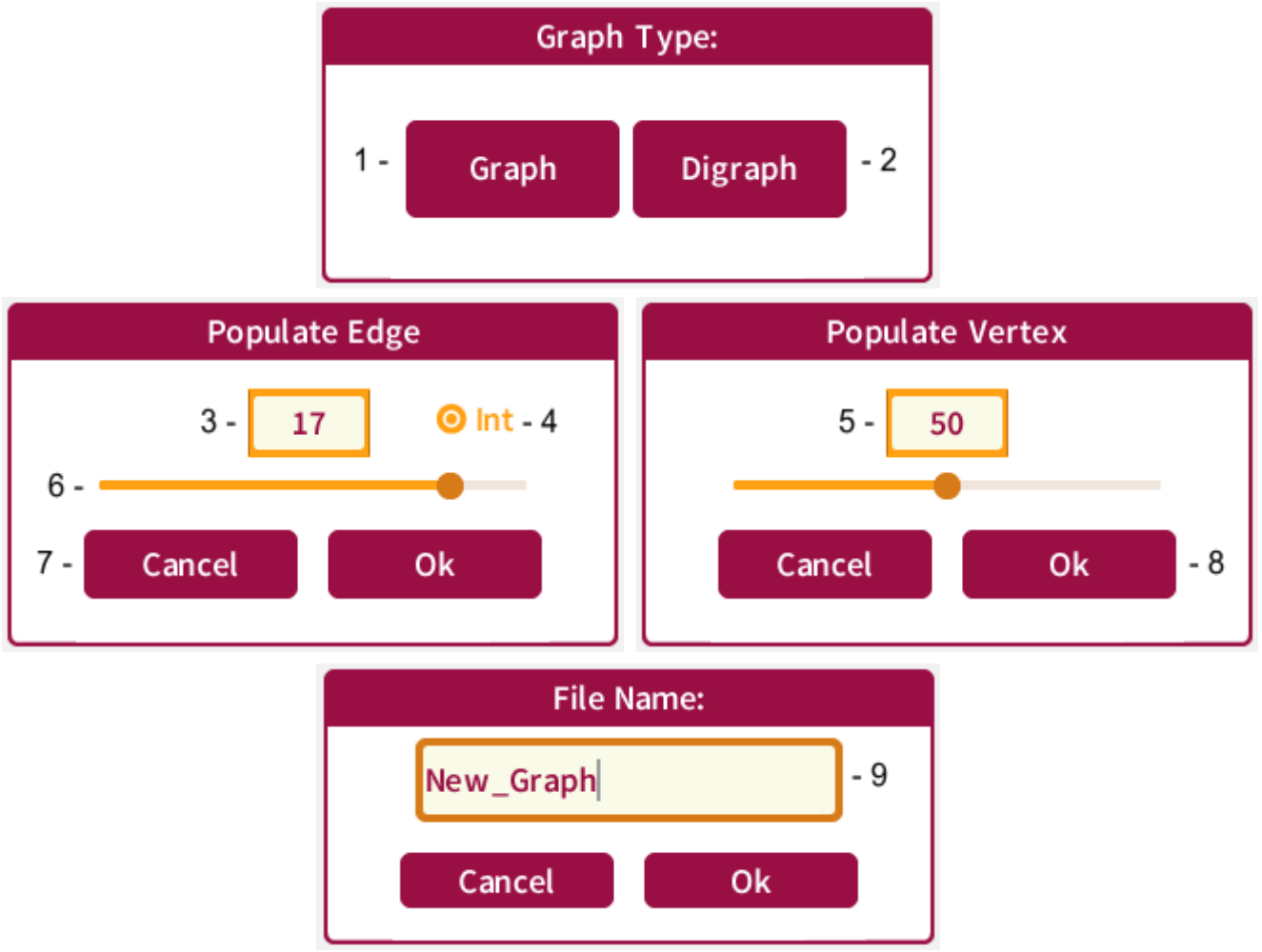
1 - Creates a new vertex that the user can place on the screen
2 - Creates a new edge that the user can place on the screen
3 - Opens the V-Populate box
4 - Opens the E-Populate box
5 - Saves the graph under its current file name
6 - Saves the graph into a new file with a user defined name
7 - Opens the current graph in the algorithm executor
8 - Exits to the main menu
9 - A moveable vertex that can be dragged and dropped by the user
10 - The weight of the edge
11 - A directed edge
12 - Clears the graph of all vertices and edges

The graph editor is where the user can create or edit graphs from scratch or from an existing graph file. The user can create new vertices and edges as well as remove vertices along with the edges they are connected to by right clicking on them. The user can also press the [V] or [E] keys to create new vertices or edges respectively.

The user can also use the V-Populate and E-Populate function to have the application create a fully connected graph for them from a set of values the user inputs (i.e amount of vertices and edge weight).

The graphs can only be saved if there are greater than two vertices and the graph is fully connected otherwise they will inform the user that the graph cannot be saved.

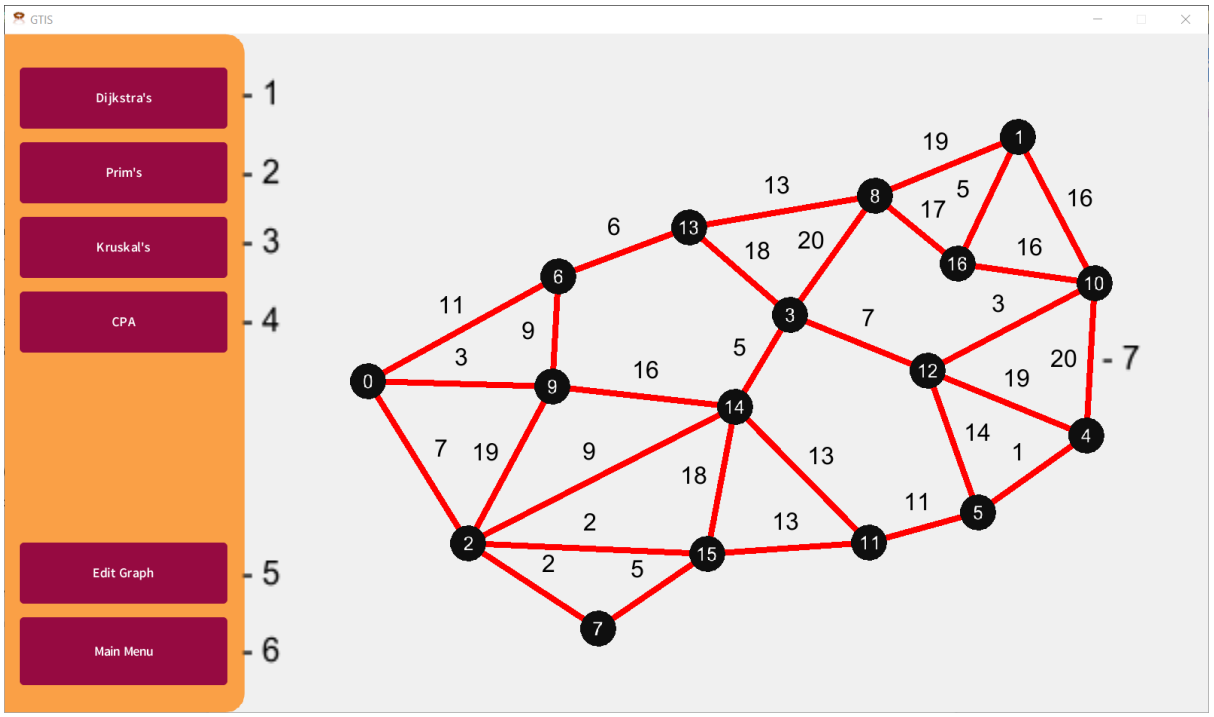
Popup Boxes:



1 - Makes the new graph have undirected edges
2 - Makes the new graph have directed edges
3 - Determines the range of the edge weights when using the E-Populate function
4 - Determines if the edge weight will be integer values when using the E-Populate function
5 - Determines the number of vertice added to the graph when using the V-Populate function
6 - Slider that will change the range of the edge weight / number of vertices
7 - Cancels the action
8 - Confirms the action
9 - A text box where the user can enter a name for their graph to be saved as

Popup boxes are used for many different things in my project as they are a great way of the user inputting values without the screen being too obscure for a smooth and seamless experience.

Algorithm Executor:



1 - Runs Dijkstra's algorithm on the current graph
2 - Runs Prim's algorithm on the current graph
3 - Runs Kruskal's algorithm on the current graph
4 - Runs the CPA algorithm on the current graph
5 - Opens the graph in the graph editor
6 - Returns to the main menu
7 - An undirected edge

The algorithm executor is where the user will be able to run any of the available algorithms on their graph.

The user will be unable to edit the graph from this screen but they can return to the graph editor by pressing the edit button where the graph will be opened.

Data and Algorithms:

Class: Sandbox		
Methods:	**Return Type:**	**Description:**
drawExistingVertex()	void	This draws the existing vertices on the graph using the LibGdx ShapeRenderer
drawExistingEdge()	void	This draws the existing edges on the graph using the LibGdx ShapeRenderer
drawMovingVertex()	void	This draws the new vertex about to be placed on on the graph using the LibGdx ShapeRenderer
drawMovingEdge()	void	This draws the new edge about to be placed on on the graph starting from the first vertex to the mouse position using the LibGdx ShapeRenderer
findClickedVertex()	int	Returns the integer related to the vertex that was just clicked by the mouse
placeNewVertex()	void	Checks if the position of the mouse is a valid position for a vertex to be placed and if so it will place it
placeNewEdge()	void	Checks if the edge about to be placed is valid (not connected to the same vertex / not connected to no vertex / not already an existing edge) and if so it
removeDuplicateEdges()	void	Deletes any invalid edges so that there will be no chance of errors even after editing the .graph file
mouseLookValid()	boolean	Returns false if the mouse is off the borders of the screen
mousePlaceValid()	boolean	Returns false if the vertex would be placed out of bounds if it was placed there
binAnimation()	void	Animates the clear all button
clearAll()	void	Rests the sandbox to
drawDigraphArrows()	void	Will draw arrows on the edge in the direction that you can travel
drawEdgeValues()	void	Draws the edge weight next to the edge
populateVertex()	void	Will randomly add a number of vertices to the sandbox area

populateEdge()	void	Will randomly add a number of edges with a weight chosen randomly within a given range to the sandbox area
getNearbyVertex()	int	Will return a vertex that is close to another given vertex. Used by the populateEdge() method
graphConnected()	boolean	Will return true if the graph is connected
dfs()	ArrayList<Integer>	Returns an Arraylist of the vertices that were visited during a depth first search pass on the graph given a starting vertex
unvisitedVertices()	ArrayList<Integer>	Returns an Arraylist of the vertices that were not visited during a depth first search pass on the graph given a starting vertex
eraseVertex()	void	Removes a given vertex from the graph along with all of its edges
Most attributes have its own get and set method so they are not listed here		
Variable:	**Type:**	**Comment:**
showVertexNumbers	boolean	If true the numbers on the vertices are shown
showEdgeWeights	boolean	If true the weights on the edges are shown
newVertexClicked	boolean	This is true if the new vertex button has been clicked
newEdgeClicked	boolean	This is true if the new edge button has been clicked
lastVertexClicked	int	This holds the value of the last vertex that was clicked
firstVertexClicked	boolean	This is true if the first vertex has been clicked when placing an edge
allowVertexMove	boolean	If true then you can drag and drop the vertices on the screen
saved	boolean	This is true if the graph in its current state is saved and will change to false when the graph is edited
modalBoxVisible	boolean	This is true of a modal popup is on the screen and stops anything but the popup from being clicked
graphIsDigraph	boolean	This is true if the graph is a digraph

firstTimeSave	boolean	This is true if no existing save for this graph is found.
vertexCoordsX	ArrayList<Integer>	This is an array that holds the X-coordinate for each vertex on the graph
vertexCoordsY	ArrayList<Integer>	This is an array that holds the Y-coordinate for each vertex on the graph
edgeListFrom	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph begins from
edgeListTo	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph goes to
edgeWeightList	ArrayList<Float>	This is an array that holds the weight of each edge on the graph
undirectedEdgeListFrom	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph begins from out of each edge that is needed for the algorithm
undirectedEdgeListTo	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph goes to out of each edge that is needed for the algorithm
currentGraphName	String	This is the current name that the graph is saved as
vertexSize	float	This is a standard size that is used as a scale when drawing things to the screen so that they stay proportionate. The size of this value is determined by the ‘vertex size’ slider in the settings menu and the screen resolution
maxVertices	int	This determines the maximum number of vertices available. The size of this value is determined by the ‘vertex size’ slider in the settings menu
maxEdges	int	This determines the maximum weight of an edge when being created by the ‘e-populate’ button. Its value is determined by the ‘ePopSlider’
vertexName	String	This is the name used for all vertices. It can be changed in the settings menu
edgeName	String	This is the name used for all edges. It can be changed in the settings menu
configArray	String[]	This array holds information stored in the

		config file. All of this information can be changed in the settings menu

Class: AlgorithmExecutor		
Methods:	**Return Type:**	**Description:**
drawExistingVertex()	void	This draws the existing vertices on the graph using the LibGdx ShapeRenderer
drawExistingEdge()	void	This draws the existing edges on the graph using the LibGdx ShapeRenderer
drawDigraphArrows()	void	Will draw arrows on the edge in the direction that you can travel
drawFloatValues()	void	Draws the edge weight next to the edge
findClickedVertex()	int	Returns the integer related to the vertex that was just clicked by the mouse
vertexSelectPointer()	void	Writes next to the cursor indicating whether you are currently selecting the start of end vertex
drawFinishedAlgEdges()	void	Draws the subset of edges returned by the algorithm in green. It also turns any edges with a weight of 0 into a dummy node if running CPA
drawFinishedAlgVertices()	void	Draws the start and end vertices on the graph
resetAlg()	void	Resets all values so that another algorithm can be run on it
primsAlg()	void	Sets up the values needed so that Prim's algorithm can be run on it and then runs Prim's algorithm.
findClosestVertexPrims()	ArrayList<Float>	Finds the closest vertex from the list of visited vertices and returns an array containing the vertex the edge is travelling from, where the edge is travelling to and the weight of the edge.
kruskalAlg()	void	Sets up the values needed so that Kruskal's algorithm can be run on it and then runs Kruskal's algorithm.
findNextEdgeKruskal()	ArrayList<Integer>	Finds an unused edge with the smallest weight and returns an array containing the vertex the edge is travelling from, where

		the edge is travelling to and the weight of the edge.
graphHasCycle()	boolean	Sets up the values needed to run dfsCycle(), runs it and then returns true if a cycle is found
dfsCycle()	void	Runs an adapted depth first search algorithm to check if the graph has a cycle
findWeightIndex()	int	Finds the index of an edge within the ArrayList edgeWeightList given the vertex that the edge is coming from and the vertex that the edge is going to
dijkstrasAlg()	void	Sets up the values needed so that bfsDijkstra() can be run and then runs bfsDijkstra() algorithm.
bfsDijkstra()	void	Runs Dijkstra's algorithm using the depth first search algorithm
cpaAlg()	void	Sets up the values needed so that critical path analysis can be run on it and then runs CPA.
bfsForwardPass()	void	Finds the earliest event times for each vertex
bfsBackwardPass()	void	Finds the latest event times for each vertex
drawCpaTimes()	void	Draws the boxes and displays the earliest and latest event times for each vertex
Most attributes have its own get and set method so they are not listed here		
Variable:	**Type:**	**Description:**
showVertexNumbers	boolean	If true the numbers on the vertices are shown
showEdgeWeights	boolean	If true the weights on the edges are shown
modalBoxVisible	boolean	This is true of a modal popup is on the screen and stops anything but the popup from being clicked
graphIsDigraph	boolean	This is true if the graph is a digraph
runningPrims	boolean	This is true if the program is running or

		has ran Prim's algorithm
runningKruskals	boolean	This is true if the program is running or has ran Kruskal'salgorithm
runningDijkstras	boolean	This is true if the program is running or has ran Djikstra's algorithm
runningCpa	boolean	This is true if the program is running or has ran CPA
primsButtonClicked	boolean	This is true if the user has pressed on the 'Prim's' button
kruskalsButtonClicked	boolean	This is true if the user has pressed on the 'Kruskal's' button
dijkstrasButtonClicked	boolean	This is true if the user has pressed on the Djikstra's' button
cpaButtonClicked	boolean	This is true if the user has pressed on the 'CPA' button
firstClickDijkstra	boolean	This is true if 'runningDijkstras' is true and the start vertex has been chosen
vertexCoordsX	ArrayList<Integer>	This is an array that holds the X-coordinate for each vertex on the graph
vertexCoordsY	ArrayList<Integer>	This is an array that holds the Y-coordinate for each vertex on the graph
edgeListFrom	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph begins from
edgeListTo	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph goes to
edgeWeightList	ArrayList<float>	This is an array that holds the weight of each edge on the graph
configArray	Array[String]	This array holds information stored in the config file. All of this information can be changed in the settings menu
vertexSize	float	This is a standard size that is used as a scale when drawing things to the screen so that they stay proportionate. The size of this value is determined by the 'vertex size' slider in the settings menu and the screen resolution

currentGraphName	String	This is the current name that the graph is saved as
undirectedEdgeListFrom	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph begins from out of each edge that is needed for an algorithm
undirectedEdgeListTo	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph goes to out of each edge that is needed for an algorithm
undirectedEdgeWeightList	ArrayList<Float>	This is an array that holds the the weight of each edge that is needed for an algorithm
visitedEdgeListFrom	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph begins from out of each edge that is used to display the used edges after an algorithm is complete
visitedEdgeListTo	ArrayList<Integer>	This is an array that holds the vertex that each edge on the graph goes to out of out of each edge that is used to display the used edges after an algorithm is complete
startVertex	int	This is the starting vertex of an algorithm if the algorithm requires it
endVertex	int	This is the finishing vertex of an algorithm if the algorithm requires it
graphHasCycle	boolean	This is true if the graph has a cycle in it
shortestPathList	ArrayList<Integer>	This array holds the list of vertices in the shortest path for Djikstra's algorithm
sortedConnections	ArrayList<Integer>	This array holds the vertices that can be visited in the current pass of the algorithm
sortedConnectionWeight	ArrayList<Float>	This array holds the total weight of the subgraph if the algorithm was to visit each vertex in 'sortedConnections' in that pass of the algorithm
tempLabels	ArrayList<Float>	This array holds the temporary labels for each vertex when running Djikstra's algorithm
permLabels	ArrayList<Float>	This array holds the permanent labels for each vertex when running Djikstra's algorithm
prevVertexList	ArrayList<Integer	This array holds the vertex visited before

	>	visiting this vertex
criticalVertex	int	This is the vertex that belongs in the critical path
earliestEventTimeList	ArrayList<Float>	This array holds the earliest event time for each vertex when executing the CPA algorithm
latestEventTimeList	ArrayList<Float>	This array holds the latest event time for each vertex when executing the CPA algorithm
sortedPreviousVertex	ArrayList<Integer>	This array holds the vertex visited before visiting this vertex but sorted so that the indexes line up with 'sortedConnection'

Class: Main Menu		
Methods:	**Return Type:**	**Description:**
Most attributes have its own get and set method so they are not listed here		
Variable:	**Type:**	**Comment:**
settingsOpen	boolean	This is true if the settings window is open
openGraphOpen	boolean	This is true if the open graph window is open
settingsChanged	boolean	This is true if the settings have been changed but not applied yet
configArray	String[]	This array holds information stored in the config file. All of this information can be changed in the settings menu
defaultConfigArray	String[]	This array holds the default values for the settings menu

Notable Algorithms:

These are some algorithms that required a significant amount of time and effort to produce the result that I wanted.

Dijkstra's Algorithm:

Dijstra's algorithm is made up of two methods; dijkstraAlg() and bfsDijkstra(). dijkstraAlg() is called after the Dijkstra's button is pressed and the start and end vertices are chosen from the graph.

dijkstraAlg(*int* startVertex, *int* endVertex**):**

dijkstraAlg() is used to set up all of the data needed for the algorithm to run. This includes adding all of the edges in the graph to a list (adding them twice if the graph is undirected), clearing old lists from previous runs of the algorithm (such as the lists that hold the temporary and permanent labels). The method bfsDijkstra() is then called from within dijkstraAlg() being given the parameters of the start and end vertices along with an empty visited list and a last vertex value of -1.

```
bfsDijkstra(new ArrayList<Integer>(),  lastVertex: -1, startVertex, endVertex);
```

After bfsDijkstra() is called, the algorithm will add the shortest path and total weight of the path to the popup box. The algorithm will also add the shortest path to another list so that it can be drawn by another method onto the graph.

bfsDijkstra(*Arraylist<Integer>()* visited*, int* lastVertex, *int* currentVertex, *int* endVertex**):**

bfsDijkstra() is based on a breadth first search algorithm that I made. This is a recursive algorithm that calls itself for every time it adds a new vertex to the visited list. The algorithm starts out by adding the current vertex to the visited list and setting its permanent label to its temporary label. It then finds all of the connections to the current vertex along with their would be temporary label if they were visited from the current vertex and adds them to two lists (connections and connectionWeight). After all of the connections to the current vertex have been found, the connections and their weights are added to a priority queue (sortedConnections and sortedConnectionWeight) via insertion sort. The algorithm then updates the values in the temporary labels list (tempLabels) if necessary before calling a recursive call passing in its visited list, the currentVertex as the lastVertex, the connection with the smallest temporary label as the currentVertex and the end vertex remains the same.

```
bfsDijkstra(visited, currentVertex, sortedConnections.get(0), endVertex);
```

This method will be called recursively until the end vertex is added to the visited list at the start of a call. The permanent label is used as the total weight and the method will record the lastVertex values as it unwinds to produce the shortest path.

```
if (currentVertex == endVertex) {
    shortestPathList.add(currentVertex);
    criticalVertex = prevVertexList.get(currentVertex);
    shortestPathList.add(criticalVertex);
    return;
}
```

Prim's Algorithm:

Prim's algorithm is made up of two methods; primsAlg() and findClosestVertexPrims(). primsAlg() is called after the Prim's button is pressed and the start vertex is chosen from the graph.

primsAlg(*int* startVertex**):**

primsAlg() is used to set up all of the data needed for the algorithm to run. This includes making the graph undirected (as prim's does not work with directed graphs), adding all of the edges in the graph to a list, clearing old lists from previous runs of the algorithm (such as the lists that hold the order of the vertices visited). The method findClosestVertexPrims() is then called from within a for loop in primsAlg() that will repeat the same amount of times as the number of vertices in the graph. findClosestVertexPrims() is passed the list of visited vertices.

```
for (int i = 0; i < vertexCoordsX.size() - 1; i++) {
    ArrayList<Float> primsArray = new ArrayList<>(findClosestVertexPrims(vertexList));
    visitedEdgeListFrom.add((int) (float) primsArray.get(0));
    visitedEdgeListTo.add((int) (float) primsArray.get(1));
    totalWeight += primsArray.get(2);
    vertexList.add((int) (float) primsArray.get(1));
}
```

The last primsArray holds the returned values from findClosestVertexPrims(). The elements of this list are the vertex the chosen edge spans from, the vertex the chosen edge spans to and the weight of the chosen edge. The edge is recorded using two arrays (visitedEdgeListTo and visitedEdgeListFrom) and the weight of the edge is added to the total weight of the minimum spanning tree.

findClosestVertexPrims(*Arraylist<Integer>()* vertexList**):**

This method starts out by finding all of the unvisited vertices that are connected to any of the vertices in the vertexList. These edges are then recorded in a list with their weights where the closest vertex (with the smallest edge weight) is then chosen as the next vertex to visit. This result is then returned in a list containing the vertex that the edge is travelling from, travelling to and the edge weight.

```
ArrayList<Float> returnList = new ArrayList<>();
if (vertexList.contains(edgeListFrom.get(closestVertex))) {
    returnList.add((float) edgeListFrom.get(closestVertex));
    returnList.add((float) edgeListTo.get(closestVertex));
} else {
    returnList.add((float) edgeListTo.get(closestVertex));
    returnList.add((float) edgeListFrom.get(closestVertex));
}
returnList.add(edgeWeightList.get(closestVertex));
return returnList;  // from: 0    To:   1    Weight:   2
```

Kruskal's Algorithm:

Kruskal's algorithm is made up of two methods; kruskalAlg() and findNextEdgeKruskals(). KruskalAlg() is called after the Kruskal's button is pressed.

kruskalAlg():

kruskalAlg() is used to set up all of the data needed for the algorithm to run. This includes making the graph undirected (as kruskal's does not work with directed graphs), adding all of the edges in the graph to a list, clearing old lists from previous runs of the algorithm (such as the lists that hold the order of the vertices visited). The method findNextEdgeKruskals() is then called from within a for loop in kruskalAlg() that will repeat the same amount of times as the number of vertices in the graph.

```
for (int i = 0; i < vertexCoordsX.size() - 1; i++) {
    ArrayList<Float> kruskalArray = new ArrayList<>(findNextEdgeKruskals());

    visitedEdgeListFrom.add((int) (float) kruskalArray.get(0));
    visitedEdgeListTo.add((int) (float) kruskalArray.get(1));
    totalWeight += kruskalArray.get(2);
    for (float vertex : kruskalArray) {
        if (!vertexList.contains((int) vertex) && kruskalArray.get(2) != vertex)
            vertexList.add((int) vertex);
    }
}
```

The last kruskalArray holds the returned values from findNextEdgeKruskals. The elements of this list are the vertex the chosen edge spans from, the vertex the chosen edge spans to and the weight of the chosen edge. The edge is recorded using two arrays (visitedEdgeListTo and visitedEdgeListFrom) and the weight of the edge is added to the total weight of the minimum spanning tree. The algorithm then adds any vertex that hasn't been visited yet to the vertexList so that it can display the path

findNextEdgeKruskals():

This method starts out by finding all of the unvisited vertices and their edges. These edges are checked so that they don't create a cycle by performing a depth-first search at every vertex that has been visited and making sure that no vertex is visited more than once. If these edges do not produce a cycle, they are then recorded in a list with their weights where the closest vertex (with the smallest edge weight) is then chosen as the next vertex to visit. This result is then returned in a list containing the vertex that the edge is travelling from, travelling to and the edge weight.

```
ArrayList<Float> returnList = new ArrayList<>();
returnList.add((float) smallestEdgeFrom);
returnList.add((float) smallestEdgeTo);
returnList.add(smallestWeight);
return returnList;  // from: 0    To:   1    Weight:   2
```

Critical Path Analysis:

Critical path analysis is made up of three methods; cpaAlg(), bfsForwardPass() and bfsBackwardPass()

cpaAlg(*int* startVertex**):**

cpaAlg() is used to set up all of the data needed for the algorithm to run. This includes adding all of the edges in the graph to a list, clearing old lists from previous runs of the algorithm (such as the lists holding the earliest and latest event times). For the CPA algorithm to be run the graph must be a digraph, if not the algorithm will inform the user and not run.The method bfsForwardPass() is run first from within cpaAlg() and is given an empty visited list, a lastVertex value of -1 and the currentVertex as the startVertex.

bfsBackwardPass() is called second as long as the graph was found to have no cycles and that every vertex was visited on the first pass. bfsBackwardPass() is passed an empty visited list, a lastVertex value of -1 and the vertex with the largest earliest event time.

Once bfsBackwardsPass() is complete the algorithm checks again for any cycles or unvisited vertices before displaying the minimum completion time (largest earliest event time), critical path and the earliest and latest event times for each vertex.

bfsForwardPass(*Arraylist<Integer>()* visited, *int* lastVertex, *int* currentVertex**):**

bfsForwardPass() is based on a breadth first search algorithm that I made. This is a recursive algorithm that calls itself for every time it adds a new vertex to the visited list. The algorithm starts out by adding the current vertex to the visited list and setting its earliest and latest event times to 0. It then finds all of the connections to the current vertex along with their would be earliest event time if visited from the current vertex (current vertex's event time plus the edge weight).

After all of the connections to the current vertex have been found, the connections and their weights are added to a priority queue (sortedConnections and sortedConnectionWeight). The algorithm then updates the values in the earliest event time list (earliestEventTimeList) if necessary before calling a recursive call passing in its visited list, the currentVertex as the lastVertex, and the vertex with the largest earliest event time.

```
bfsForwardPass(visited, currentVertex, sortedConnections.get(0));
```

This method will be called recursively until there are no more vertices left to visit in the queue or it finds a cycle in the graph. If there is a cycle, the algorithm will stop running and the user will be notified.

bfsbackwardsPass(*Arraylist<Integer>()* visited, *int* lastVertex, *int* currentVertex**):**
bfsBackwardsPass() is based on a breadth first search algorithm that I made. This is a recursive algorithm that calls itself for every time it adds a new vertex to the visited list. The algorithm starts out by adding the current vertex to the visited list and setting its latest event time to the earliest event time of the current vertex. It then finds all of the connections to the current vertex along with their would be latest event time if visited from the current vertex (current vertex's latest time minus the edge weight).

After all of the connections to the current vertex have been found, the connections and their weights are added to a priority queue (sortedConnections and sortedConnectionWeight). The algorithm then updates the values in the latest event time list (latestEventTimeList) if necessary before calling a recursive call passing in its visited list, the currentVertex as the lastVertex, and the vertex with the smallest latest event time.

```
bfsBackwardPass(visited, currentVertex, sortedConnections.get(0));
```

This method will be called recursively until there are no more vertices left to visit in the queue or it finds a cycle in the graph. If there is a cycle, the algorithm will stop running and the user will be notified.

Drawing edge weights:

When drawing the edge weight onto the graph I had some trouble with making it clear what edge that it corresponded to. To do this is created a method called drawFloatValues().

This method starts by checking whether the graph is unidirectional or bidirectional as I wanted to display the number a little further away from the digraph arrow so that they didn't overlap. Next the method finds the midpoint of the edge by adding the vertices X and Y coordinates and halving them. It then finds the angle that they make with the positive X-axis by using the inverse tan function and the sum of the X and Y coordinates of the vertices.

```
float midpointX = (vertexCoordsX.get(edgeListFrom.get(i)) + vertexCoordsX.get(edgeListTo.get(i))) * 0.5f;
float midpointY = (vertexCoordsY.get(edgeListFrom.get(i)) + vertexCoordsY.get(edgeListTo.get(i))) * 0.5f;
double rotationAngle = (0.5f * Math.PI) + Math.atan((double) (vertexCoordsY.get(edgeListTo.get(i)) -
```

The algorithm then uses basic trigonometry involving the angle and a factor that is dependent on the screen and vertex size to work out the amount that is to be added to the X (rCOS(a)) and Y (rSIN(a)) coordinates.

```
float addY = (float) (2 * vertexSize * Math.sin(rotationAngle));
float addX = (float) (2 * vertexSize * Math.cos(rotationAngle));
```

The algorithm then gets the edge weight value and converts it from a float to a string before using the GlyphLayout().layout() function to get the string width and height. It then uses all of these values to draw the edge weight to the screen using the spriteBatch().

```
batch.begin();
if (graphIsDigraph)
    font.draw(batch, weightText,  x: midpointX - 0.5f * fontWidth + addX,  y: midpointY + 0.5f * fontHeight + addY);
else
    font.draw(batch, weightText,  x: midpointX - 0.5f * fontWidth + 0.75f * addX,  y: midpointY + 0.5f * fontHeight + 0.75f * addY);
batch.end();
```

Drawing digraph arrows:

When drawing a digraph there needs to be an arrow pointing in the direction of travel. To draw this arrow I used a method called drawDigraphArrows(). This method starts by finding the midpoint of the edge by adding their X and Y coordinates and halving them.

```
float midpointX = (vertexCoordsX.get(edgeListFrom.get(i)) + vertexCoordsX.get(edgeListTo.get(i))) * 0.5f;
float midpointY = (vertexCoordsY.get(edgeListFrom.get(i)) + vertexCoordsY.get(edgeListTo.get(i))) * 0.5f;
```

After the midpoint is found, the algorithm finds the coordinates of where the three vertices of the arrow would be if it was pointing straight upwards. To do this I use pythagoras with the size of the edge (vertexSize) to find the Y coordinate and simple maths to find the X coordinate.

```
float x1 = -vertexSize;
float y1 = (float) (0.5f * Math.sqrt((2 * vertexSize * 2 * vertexSize) - (vertexSize * vertexSize)));
float x2 = vertexSize;
float y2 = (float) (0.5f * Math.sqrt((2 * vertexSize * 2 * vertexSize) - (vertexSize * vertexSize)));
float x3 = 0;
float y3 = (float) (0.5f * Math.sqrt((2 * vertexSize * 2 * vertexSize) - (vertexSize * vertexSize)));
```

Next I have to find the angle of rotation of the triangle. I start by getting the angle of rotation from the X-Axis by using the inverse tan function. After the algorithm does this, I need to correct it so that it is pointing in the right direction. To do this I find if the vertex that is being travelled to is to the left or right of where the edge is coming from. If it is to the left, I add 90° and if it is to the right I subtract 90°.

```
double rotationAngle;
if (vertexCoordsX.get(edgeListTo.get(i)) < vertexCoordsX.get(edgeListFrom.get(i)))
    rotationAngle = (0.5f * Math.PI) + Math.atan((double) (vertexCoordsY.get(edgeListTo.get(i))
else
    rotationAngle = -(0.5f * Math.PI) + Math.atan((double) (vertexCoordsY.get(edgeListTo.get(i))
```

Finally i put all of these parts together and use the libGDX shaperenderer to draw the triangle to the correct coordinates on the screen with the correct rotation

Connected Graphs:

To test if a graph is connected or not I use a depth first search algorithm that I made. The method that I made takes the current vertex being explored, a visited vertices list and the last vertex that was explored before the current vertex.

```
private void dfsCycle(int currentVertex, ArrayList<Integer> visited, int lastVertex) {
```

This algorithm starts by finding all of the connected vertices to the current vertex and storing them in a list called connections

```
ArrayList<Integer> connections = new ArrayList<>();
for (int i = 0; i < newVisitedEdgeListFrom.size(); i++) {
    if (newVisitedEdgeListFrom.get(i) == currentVertex && !connections.contains(newVisitedEdgeListTo.get(i)))
        connections.add(newVisitedEdgeListTo.get(i));
}
```

The algorithm then has a for each loop that repeats for every connection in the connections list. If the connection is already in the visited vertices list and it is not the last vertex visited, the variable graphHasCycle is changed to true, the loop breaks and the algorithm starts to unwind. If the vertex is not already in the visited list, the method calls itself with the visited list, the current vertex as the last vertex and the connection as the current vertex.

```
for (Integer connection : connections) {
    if (!visited.contains(connection)) {
        dfsCycle(connection, visited, currentVertex);
    } else if (connection ≠ lastVertex) {
        graphHasCycle = true;
        break;
    }
}
```

The algorithm finished when there are no more vertices left to be visited or when it finds a cycle.

Testing:

Throughout the development of this application, I have had to carry out tests consistently in order to make sure that every aspect of my program functions exactly as I intended for it to.

An example of this is when I was developing how to display a digraph arrow on an edge. I had to find a way of displaying the arrow pointing the correct direction at exactly the centre of the line. To do this I tried many different methods such as using a 3x3 rotation matrix to transform the coordinates and using vectors to find the angle of rotation, but I eventually figured out a way of using trigonometric values and the positions of the vertices to draw them correctly.

M = Main Menu **S** = Sandbox (Editor) **A** = Alg Executor

Obj	Test	Test description	Expected result	Pass
	M1.1	Changing settings in the settings window to make sure they work	All settings should be applied immediately	✔
	M1.2	Making sure the 'config.txt' file gets updated	Changes to the 'config.txt' file should be made as necessary	✔
1.4	**M2.1**	Graphs can be both opened in the editor from the open graph window	The correct graph should be opened in the graph editor	✔
	M2.2	Graphs can be opened in the algorithm executor from the open graph window	The correct graph should be opened in the graph algorithm executor	✔
	M2.3	Graphs can be deleted from the open graph window	The correct .graph file should be deleted from the computer	✔
1.4	**M3**	A new graph can be opened in the graph editor by pressing on the 'New Graph' button	An empty graph should be opened in the graph editor	✔
1.2	**S1.1**	A new vertex should be added by clicking the 'New Vertex' button	A new vertex should follow the cursor until it is placed on the screen where it should stay	✔
	S1.2	A new edge should be added by clicking the 'New Edge' button	A new vertex should be placed on the screen by clicking the vertex it will start from then clicking the vertex it will end at	✔
	S2.1	The graph should be able to display digraphs	The graph's edges should be unidirectional with arrows	✔

			indicating the direction of travel	
1.1	**S2.2**	The graph should be able to display undirected graphs	The graph's edges should be bidirectional with no arrows	✔
1.3.1	**S3.1**	The 'V-Populate' should add a chosen number of vertices to the graph	A chosen number of vertices should be added to the screen	✔
1.3.2	**S3.2**	The 'E-Populate' should fully connect the graph with weights within a range defined by the user	The graph should be fully connected with the weights of the edges being within the user defined range	✔
2.1.1 2.3	**S4.1**	The program should be able to 'save as' a .graph file to the computer with the current graph's data and a custom name	When the 'Save As' button is pressed and the graph is valid (2+ vertices and fully connected,) a new '.graph' file should be created in the 'Saved Graphs' folder with the name that the user input. This file should contain whether the graph is unidirectional or bidirectional, the coordinates of each vertex, where each edge starts and ends and the weight of each edge	✔
2.2	**S4.2**	The program should check that the file does not have the same name as an existing .graph file and if it does it should change it	The program should change the name of a new file if it has the same name as an existing file within the same directory	✔
	S4.3	The program should be able to overwrite a save if the .graph file already exists and is open in the graph editor	When the 'Save' button is pressed, the program should change the relevant '.graph' file's content and update it to the new graph	✔
	S5.1	The program should clear all edges and vertices when the clear all button is pressed	When the bin ('Clear All Button') is pressed, the screen should be cleared of all vertices and edges	✔
	S5.2	The program should delete a vertex and all connected edges when it is right clicked	When a vertex is right clicked it should be removed from the screen along with	✔
	A1.1	When the 'Djikstra's' button is pressed, the user should be prompted to choose the start and end vertex of Dijkstra's algorithm	After the 'Djikstra's' button is clicked, the user should be prompted to pick a start vertex. When a start vertex is chosen, the user should be prompted to choose an end vertex	✔

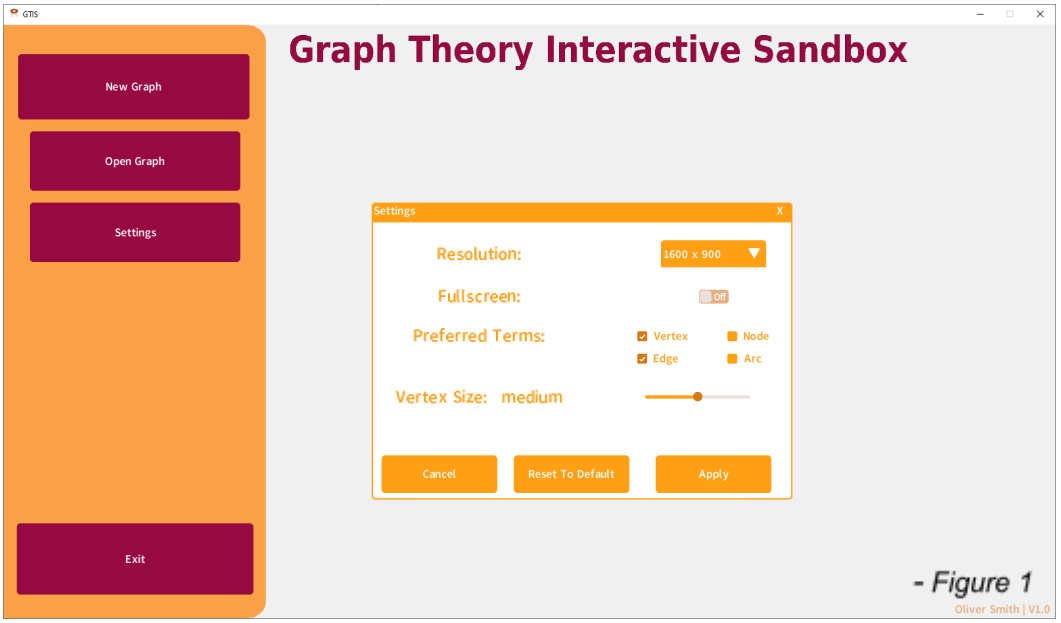
3.1	**A1.2**	The program should then run Djikstra's algorithm between the start and end vertices and display the vertices in the shortest path along with the total weight	The program should write the vertices in the shortest path and the total weight of the shortest path to the screen	✔
3.1.1	**A1.3**	The program should display the shortest path in a different colour on the graph	A green path should overlay the edges indicating the shortest path	✔
	A1.4	If the program could not find a path between the two vertices it should notify the user	The program should write a message to the screen saying 'No Path Found!'	✔
	A2.1	When the 'Prim's' button is pressed, the user should be prompted to choose the start vertex of 'Prim's' algorithm	After the 'Prim's' button is clicked, the user should be prompted to pick a start vertex	✔
3.2	**A2.2**	The program should then run Prim's algorithm spanning from the start vertex and displaying the path and total weight	The program should write the path it took the total weight of the shortest path to the screen	✔
3.2.1	**A2.3**	The program should display the minimum spanning tree in a different colour on the graph	A green path should overlay the edges indicating the minimum spanning tree	✔
3.3	**A3.1**	When the 'Kruskal's' button is pressed, the program should run Kruskal's algorithm and display the path and total weight	The program should write the path it took the total weight of the shortest path to the screen	✔
3.3.1	**A3.2**	The program should display the minimum spanning tree in a different colour on the graph	A green path should overlay the edges indicating the minimum spanning tree	✔
	A4.1	When the 'CPA' button is pressed, the user should be prompted to choose the start vertex of the CPA algorithm	After the 'CPA' button is clicked, the user should be prompted to pick a start vertex	✔
3.4	**A4.2**	The program should then run CPA spanning from the start vertex and displaying critical vertices and the minimum completion time	The program should write the critical vertices and the minimum completion time to the screen	✔
3.4.1	**A4.3**	The program should display the critical path in a different	A green path should overlay the edges indicating critical path	✔

		colour on the graph		
3.4.1	**A4.4**	The program should display the earliest and latest event times for each vertex	A box should be drawn over each vertex containing the earliest and latest event times The program should	✔
4.3	**A4.5**	If the algorithm cannot be run on the graph it should tell the user	A window should pop up saying that the chosen algorithm cannot be run on this graph	✔

Testing Evidence:

Main Menu:

Test M1.1:



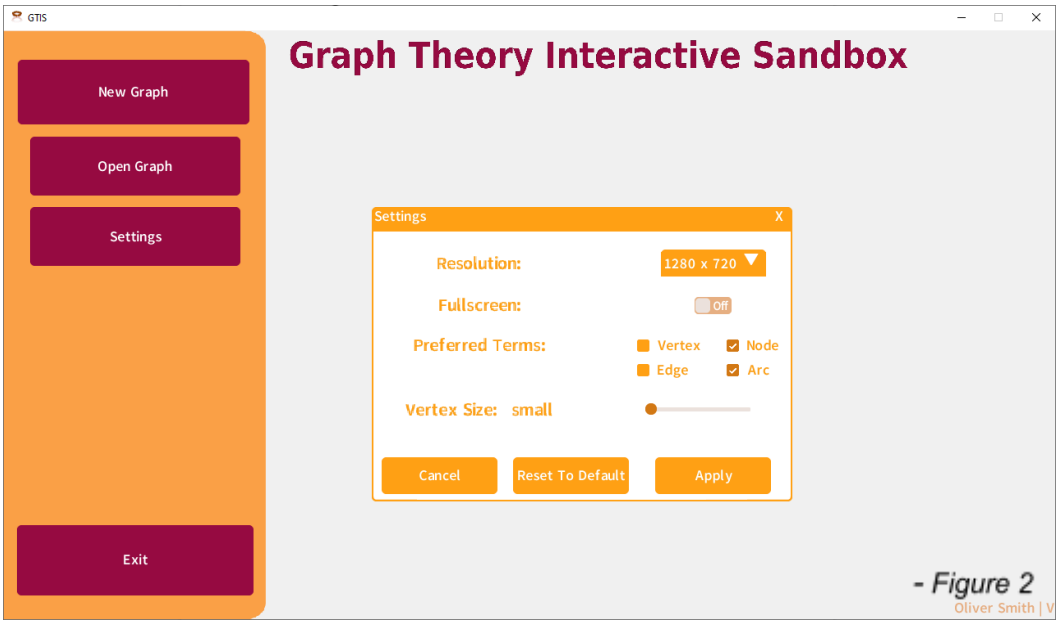
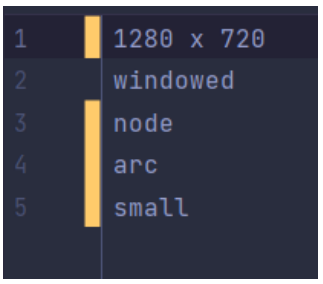


*Figure 1 : Screen **before** applying changes*
*Figure 2 : Screen **after** applying changes*

As we can see from figure 2, the screen's resolution has changed immediately to the chosen resolution

Test M1.2:

- Figure 1

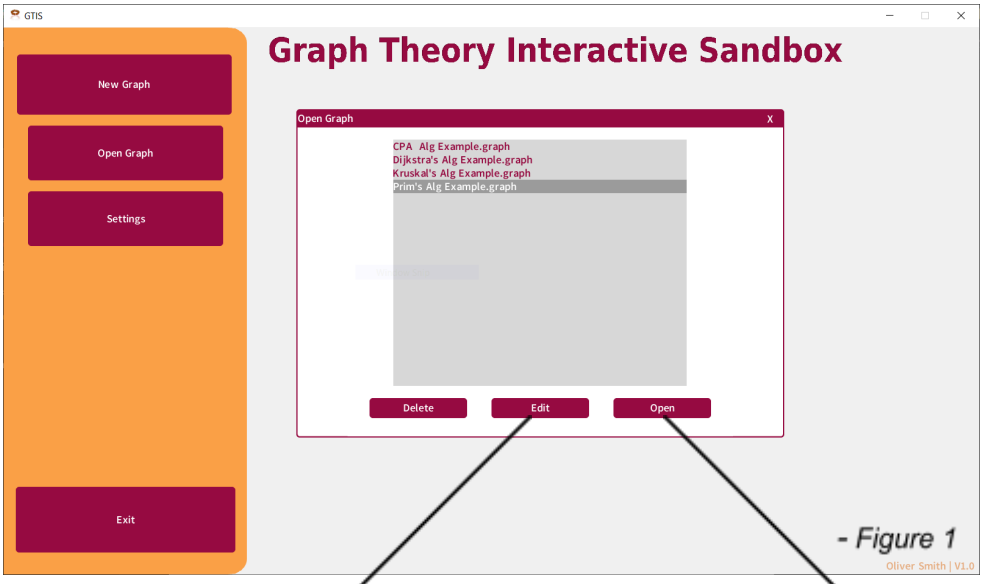


- Figure 2

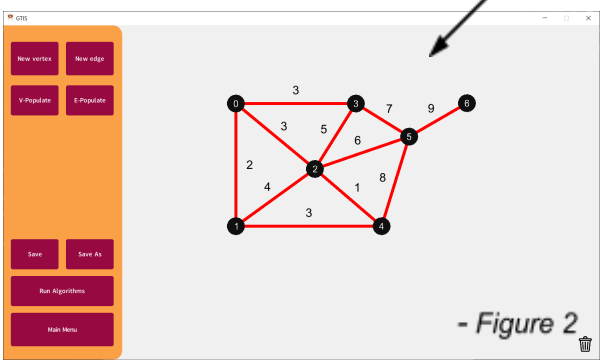
*Figure 1 : config.txt **before** applying changes*
*Figure 2 : config.txt **after** applying changes*

As we can see from figure 2, the program has updated the relevant lines in the config file to the correct data

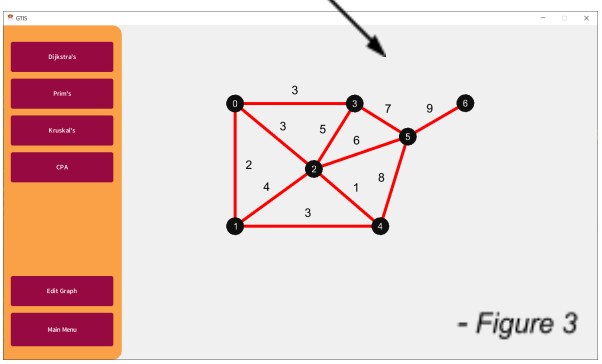
Test M2.1 & M2.2:



- Figure 1



- Figure 2

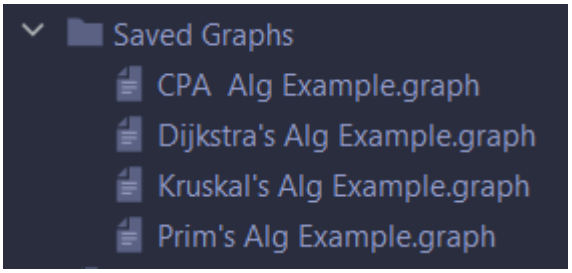


- Figure 3

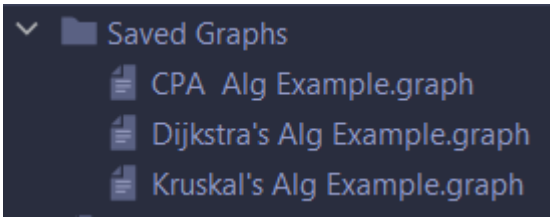
Figure 1 : Open Graph window with the 'Prim's Algorithm Example' graph selected
Figure 2 : 'The screen after the 'Edit' button was pressed
Figure 3 : 'The screen after the 'Open button was pressed

We can see in figure 2 that after pressing the edit button the '*Prim's Algorithm Example'* graph was opened in the graph editor.
We can see in figure 3 that after pressing the open button the '*Prim's Algorithm Example'* graph was opened in the algorithm executor.

Test M2.3:



- *Figure 1*

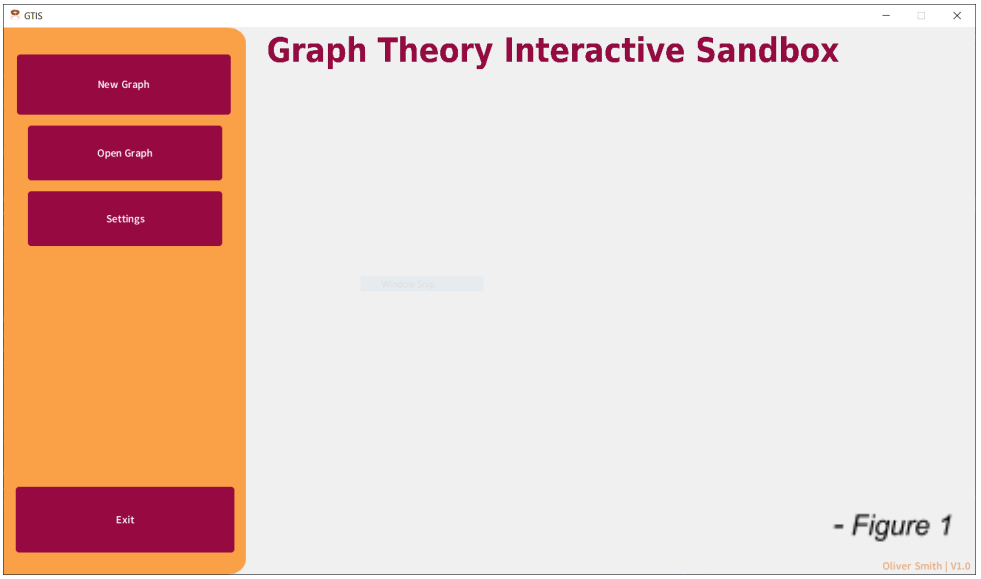


- *Figure 2*

*Figure 1 : Saved Graphs folder **before** the delete button was pressed*
*Figure 2 : Saved Graphs folder **after** the delete button was pressed*

We can see in figure 2 that after pressing the delete button the '*Prim's Algorithm Example'* graph was removed from the Saved Graphs folder

Test M3:



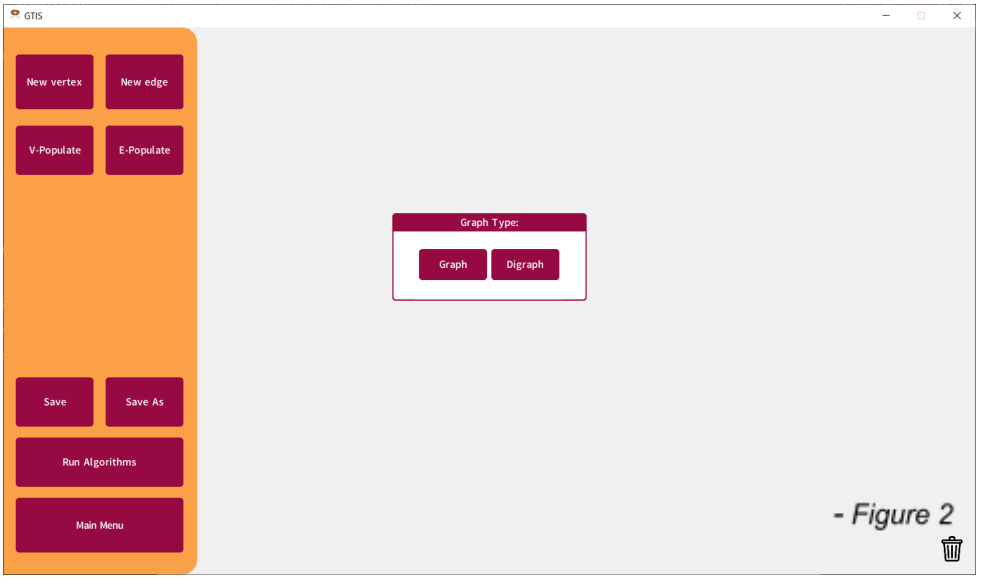
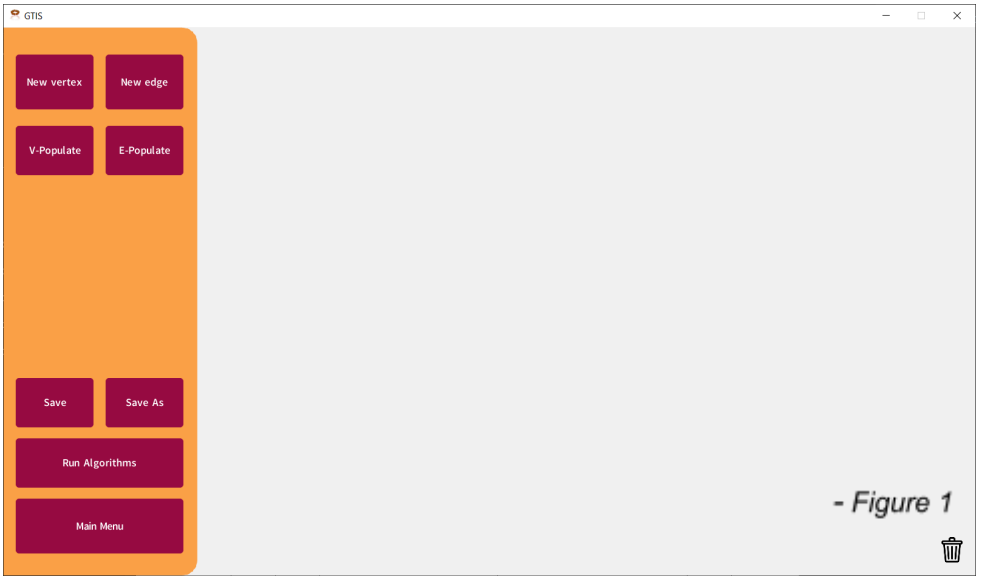


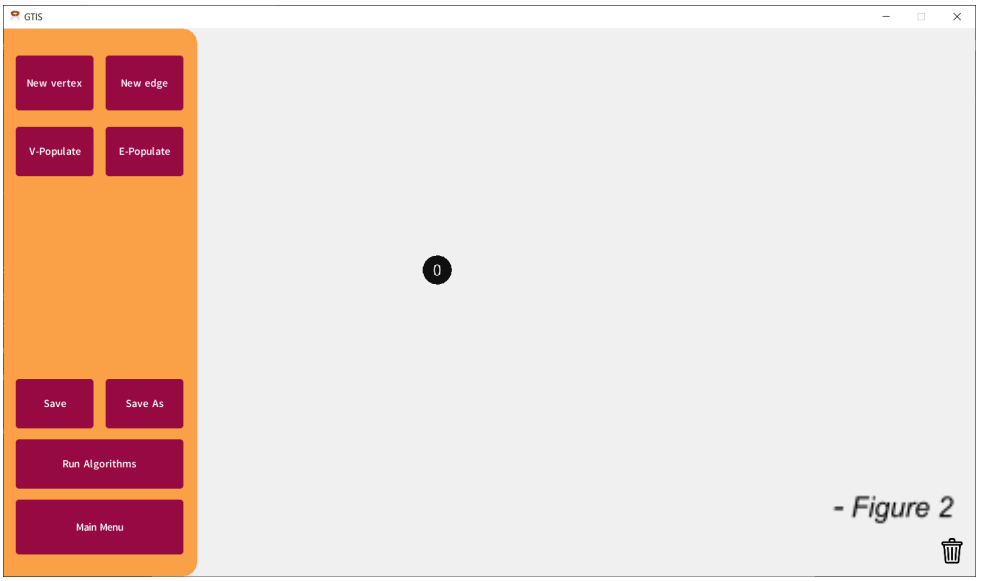
*Figure 1 : The screen **before** pressing the new graph button*
*Figure 2 : The screen **after** pressing the new graph button*

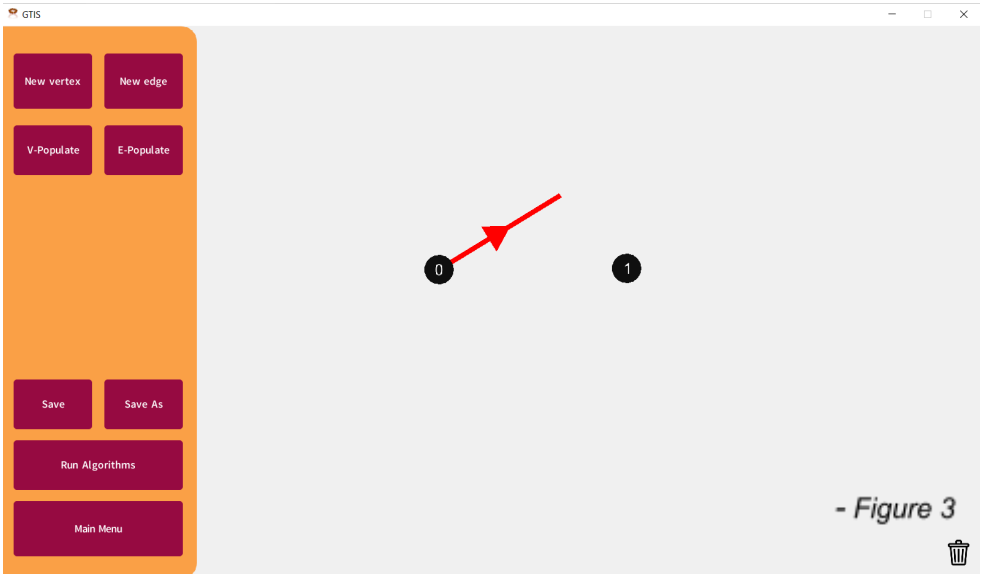
As we can see from figure 2, we can see that an empty graph has been opened in the graph editor.

Sandbox:

Test S1.1 & S.1.2:







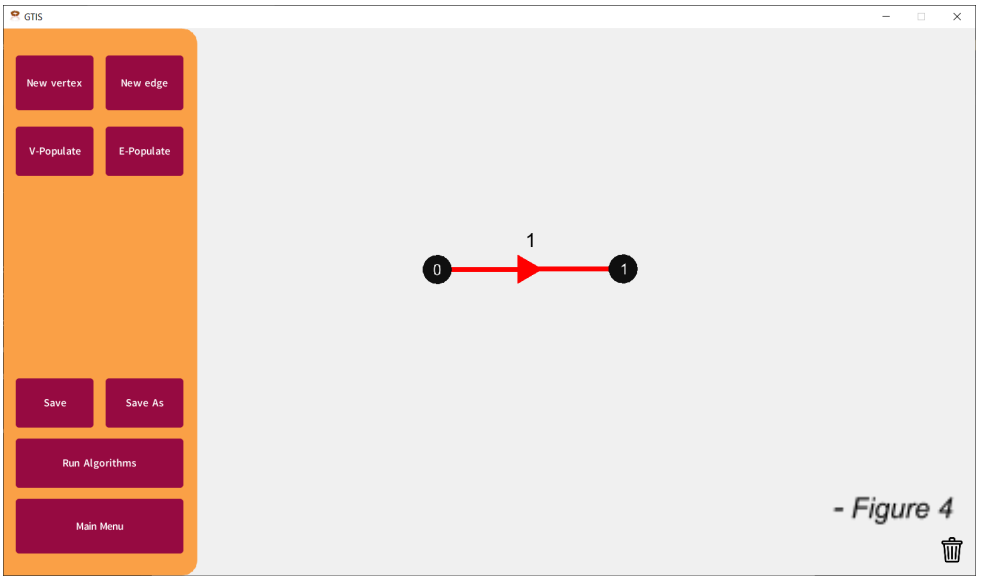
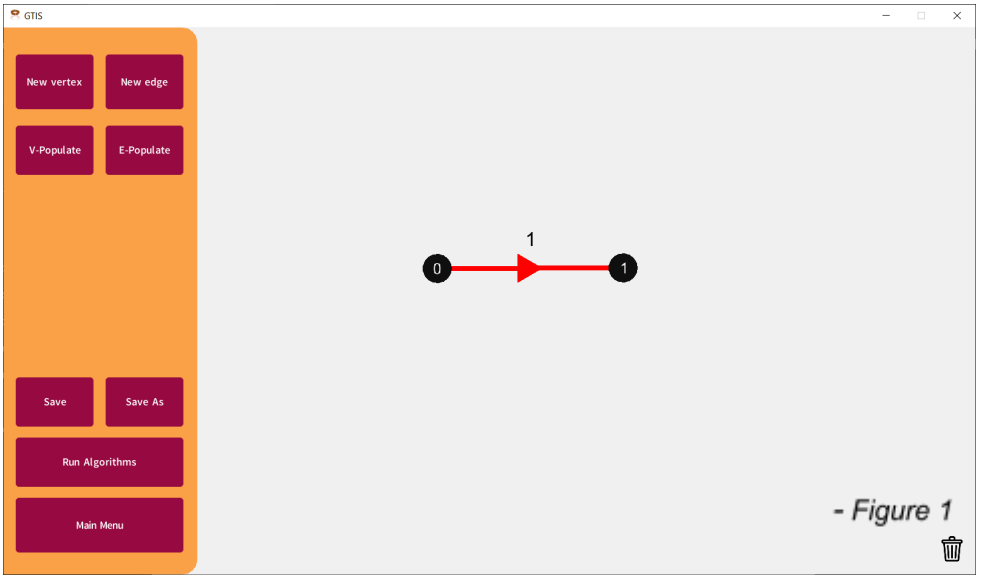


Figure 1 : An empty graph editor with no vertices or edges
Figure 2 : The editor after the New Vertex button was pressed and a vertex was placed
Figure 3 : The editor after the New Edge button was pressed and the first vertex was clicked
Figure 4 : The editor after the edge has been placed

When the New Vertex button is pressed a vertex follows the cursor until it is placed by clicking the mouse button as shown in figure 2.
When the New Edge button is pressed, the user can press the vertex where the edge should start from, as shown in figure 3, and will be connected to the next vertex that the user presses, as shown in figure 4

Test S2.1 & S2.2:



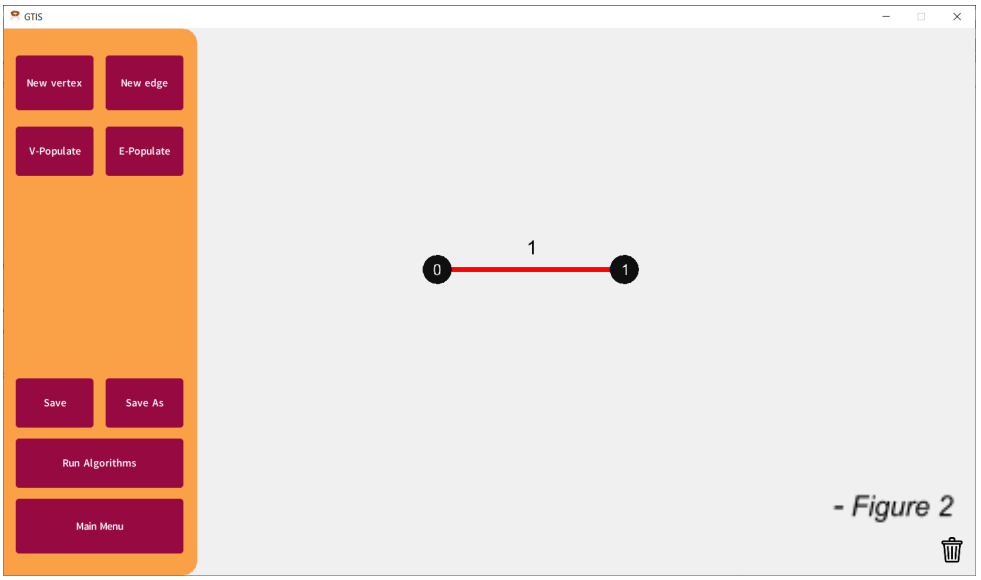
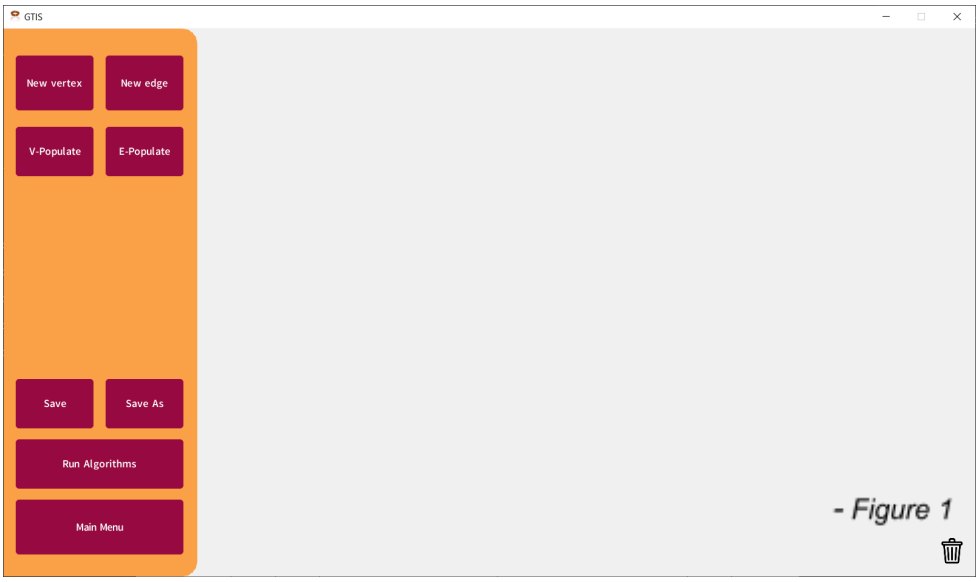
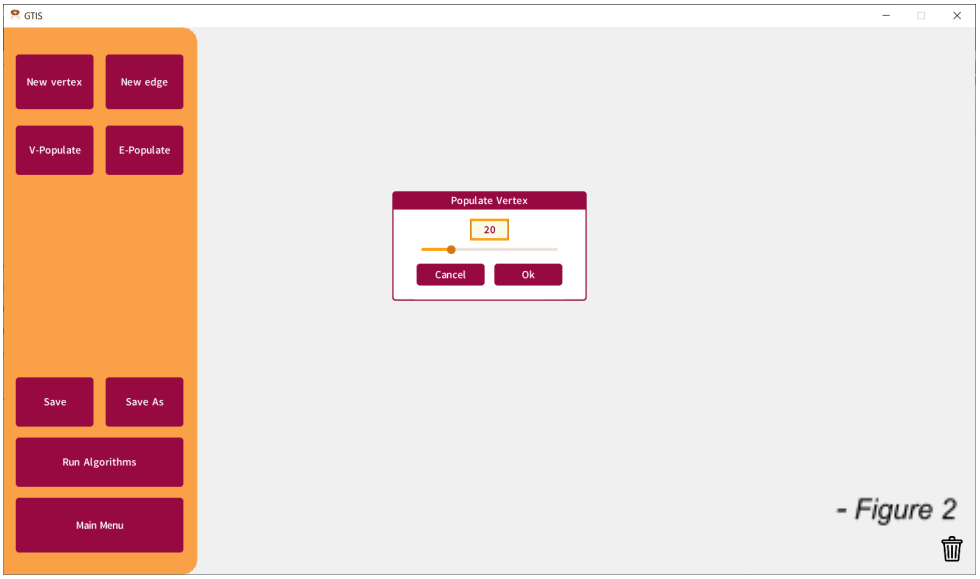


*Figure 1 : Two vertices connected by a **unidirectional** edge displayed in the graph editor*
*Figure 2 : Two vertices connected by a **bidirectional** edge displayed in the graph editor*

As we can see in figure 1, a unidirectional edge is being displayed with arrows indicating the direction of travel.
As we can see in figure 2, a bidirectional edge is being displayed with no direction of travel

Test S3.1:





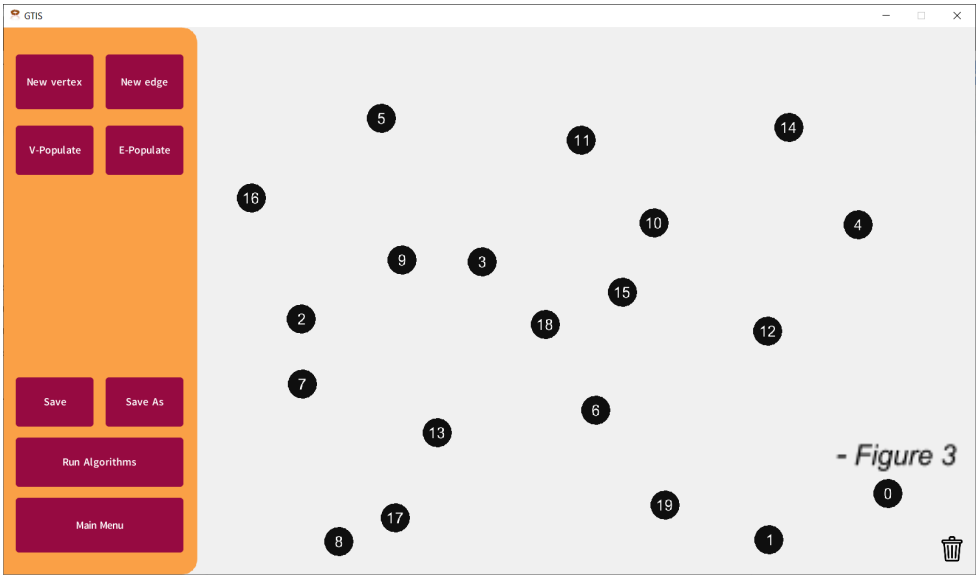
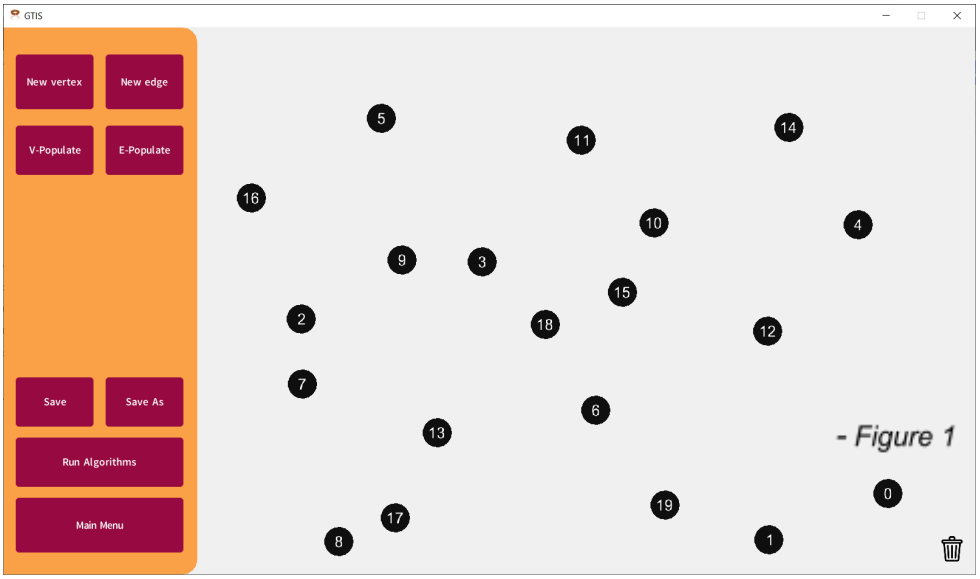
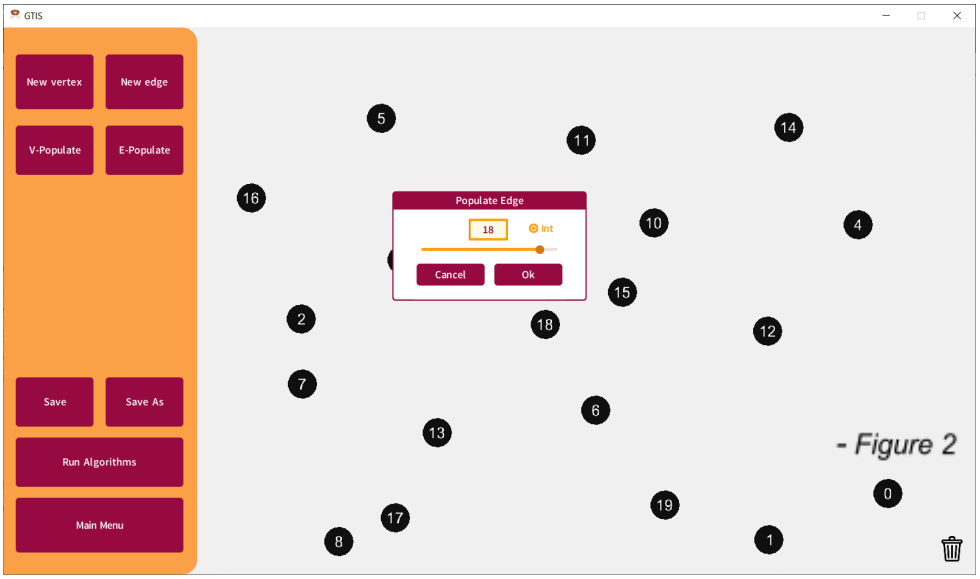


Figure 1 : The editor with a new empty graph loaded
Figure 2 : The popup after pessing on the V-Populate button
Figure 3 : The graph after selecting 20 vertices and confirming

As we can see from figure 3, the number of vertices selected in figure 2 (20) have been added to the screen with random coordinates

Test S3.1:


GTIS
New vertex
New edge
V-Populate
E-Populate
Save
Save As
Run Algorithms
Main Menu
- Figure 1


GTIS
New vertex
New edge
V-Populate
E-Populate
Save
Save As
Run Algorithms
Main Menu
Populate Edge
18
Int
Cancel
Ok
- Figure 2

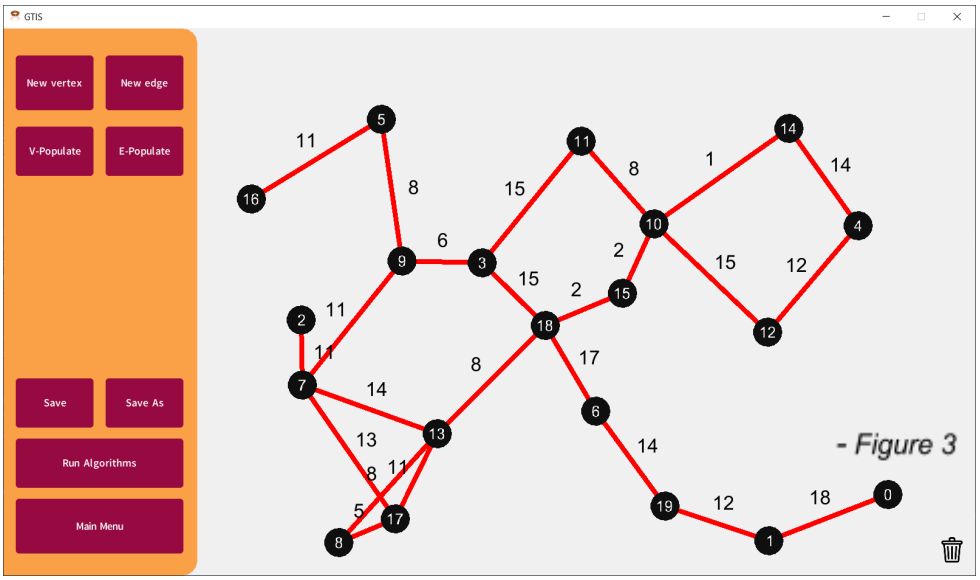
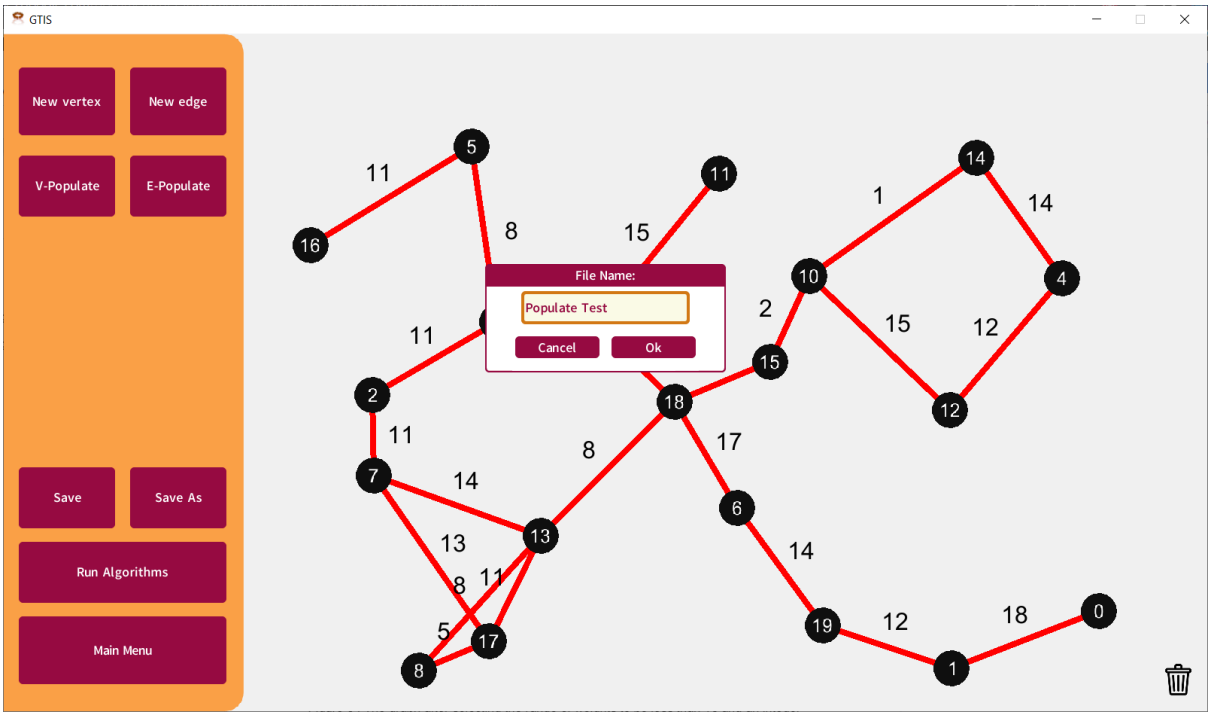

GTIS
New vertex
New edge
V-Populate
E-Populate
Save
Save As
Run Algorithms
Main Menu
- Figure 3

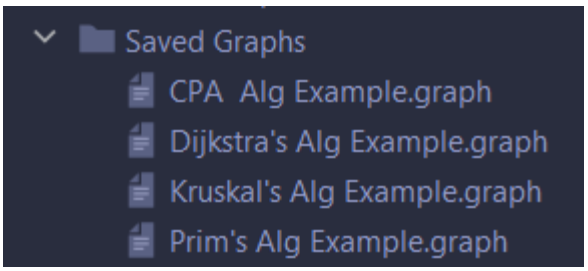
Figure 1 : A graph with no edges
Figure 2 : The popup after pressing the E-Populate button
Figure 3 : The graph after selecting the range of weights to be less than 18 and an integer and confirming

As we can see from figure 3, the graph has been fully connected and the weights of the edges are integers and fall within the range of 0 and 18

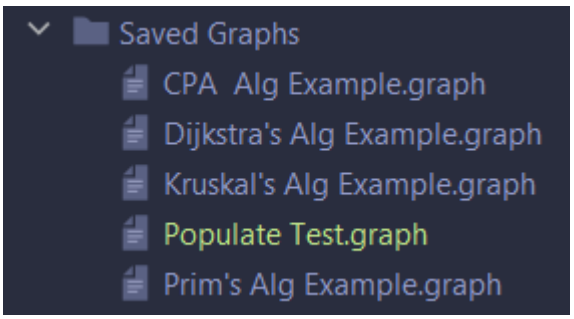
Test S4.1:



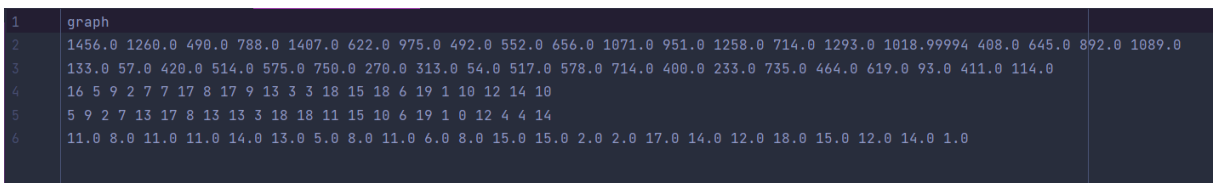
- Figure 1



- Figure 2



- Figure 3



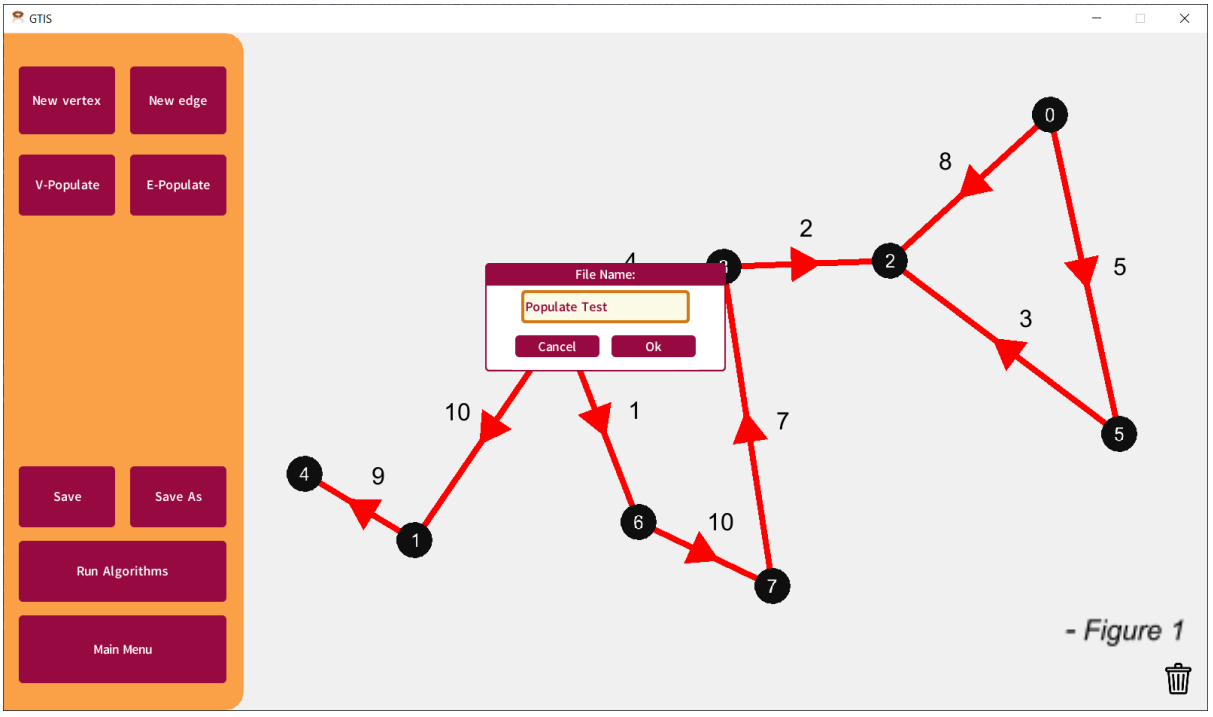
- Figure 4

Figure 1 : The screen after the Save As button was pressed with the name of the graph input
*Figure 2 : The Saved Graphs folder **before** the file was saved as*
*Figure 3 : The Saved Graphs folder **after** the file was saved as*
Figure 4 : The content of the saved graph file (Populate Test.Graph)

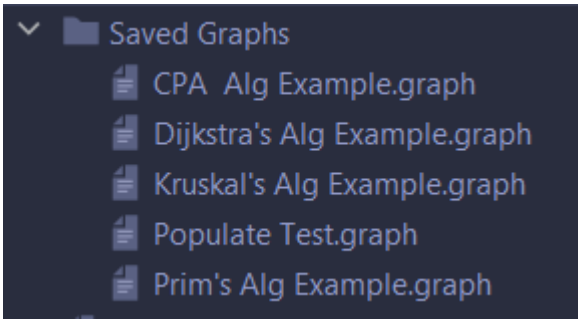
As we can see from figure 3, the graph has been saved ito the Saved Graphs folder under the name that was input in figure 1
As we can see in figure 4, the graph has been saved with the following data:

- *Type of graph (unidirectional or bidirectional)*
- *The X coordinate of each vertex*
- *The Y coordinate of each vertex*
- *The vertex that each edge starts at*
- *The vertex that each edge ends at*
- *The weight of each edge*

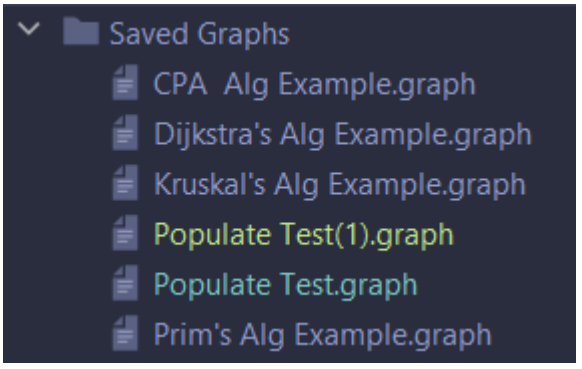
Test S4.2:



- *Figure 1*



- *Figure 2*

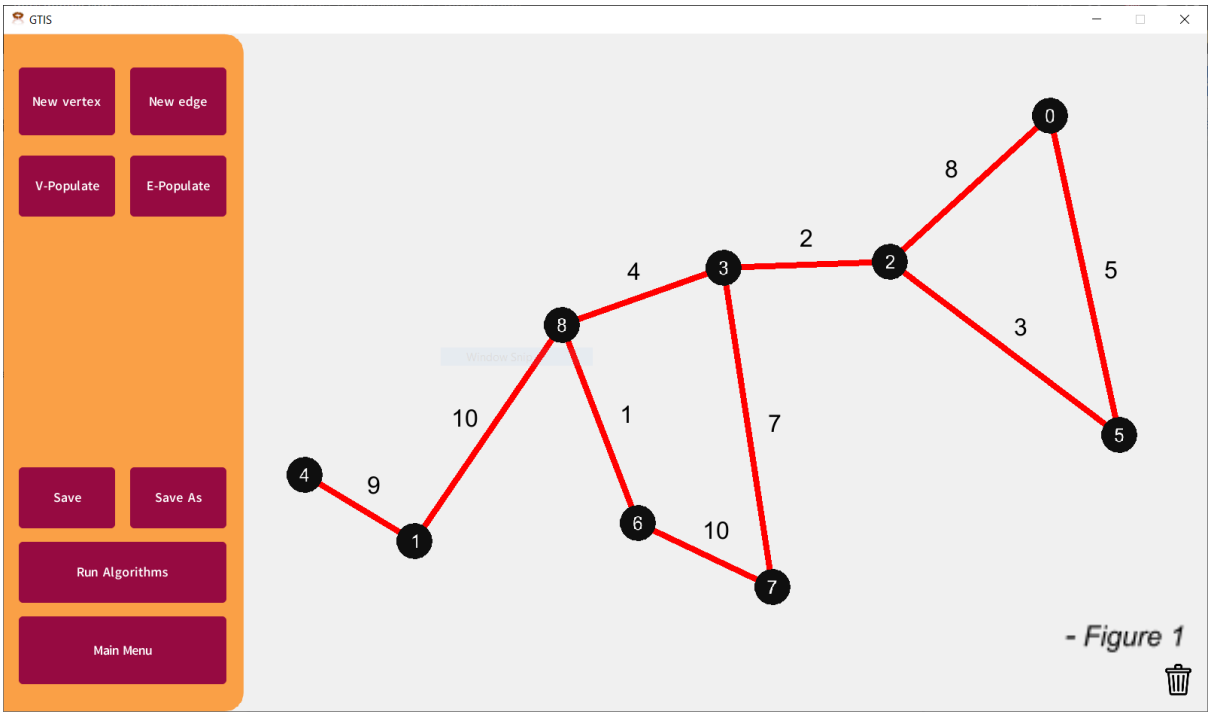


- *Figure 3*

Figure 1 : A new graph being saved under the same name as an existing graph
*Figure 2 : The Saved Graphs folder **before** saving the new graph*
*Figure 3 : The Saved Graphs folder **after** saving the new graph*

As we can see in figure 3, the new graph gets saved under a new name despite the name the user input

Test S4.3:



- Figure 1

```
graph
1456.0 1260.0 490.0 788.0 1407.0 622.0 975.0 492.0 552.0 656.0 1071.0 951.0 1258.0 714.0 1293.0 1018.99994 408.0 645.0 892.0 1089.0
133.0 57.0 420.0 514.0 575.0 750.0 270.0 313.0 54.0 517.0 578.0 714.0 400.0 233.0 735.0 464.0 619.0 93.0 411.0 114.0
16 5 9 2 7 7 17 8 17 9 13 3 3 18 15 18 6 19 1 10 12 14 10
5 9 2 7 13 17 8 13 13 3 18 18 11 15 10 6 19 1 0 12 4 4 14
11.0 8.0 11.0 11.0 14.0 13.0 5.0 8.0 11.0 6.0 8.0 15.0 15.0 2.0 2.0 17.0 14.0 12.0 18.0 15.0 12.0 14.0 1.0
```

- Figure 2

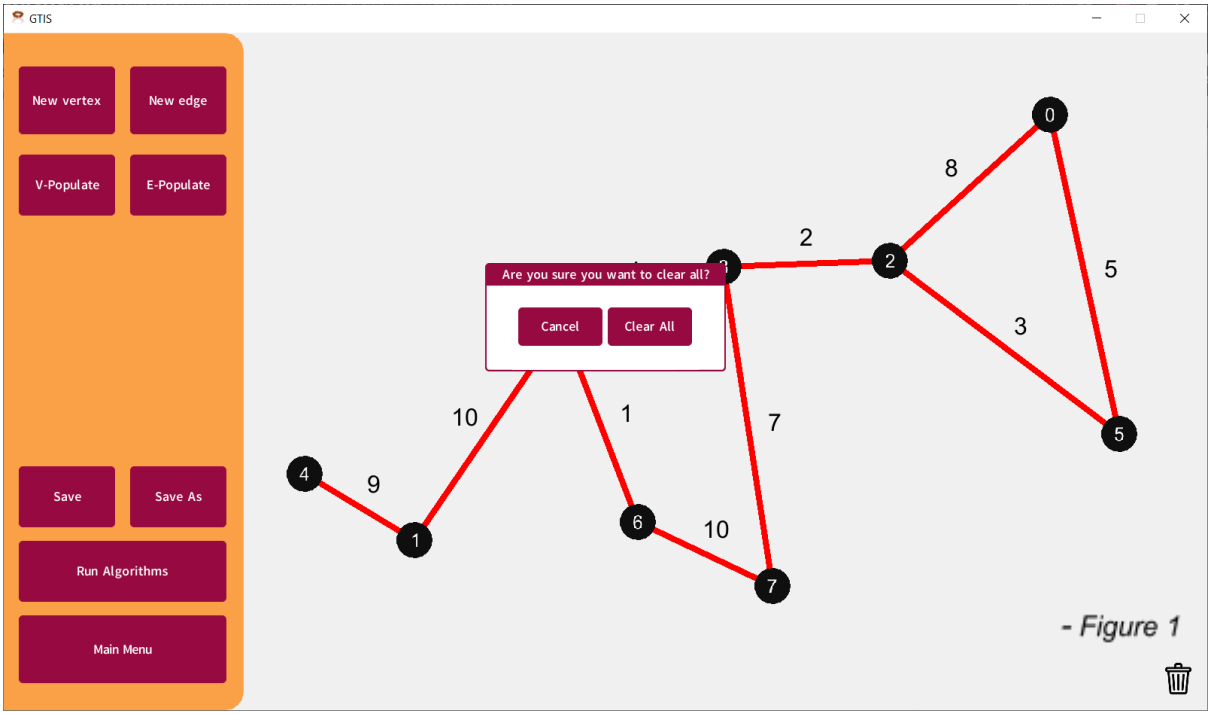
```
graph
1391.0 546.0 1178.0 957.0 400.0 1483.0 843.0 1022.0 742.0
791.0 226.0 597.0 589.0 314.0 367.0 250.0 165.0 513.0
0 1 3 5 6 8 8 7 0 3
2 4 2 2 7 6 1 3 5 8
8.0 9.0 2.0 3.0 10.0 1.0 10.0 7.0 5.0 4.0
```

- Figure 3

Figure 1 : The Populate Test graph where a new graph has been made
*Figure 2 : The contents of the Populate Test.Graph folder **before** pressing the Save button*
*Figure 3 : The contents of the Populate Test.Graph folder **after** pressing the Save button*

As we can see in figure 3, the new graph's data has overwritten the original graph's data

Test S5.1:



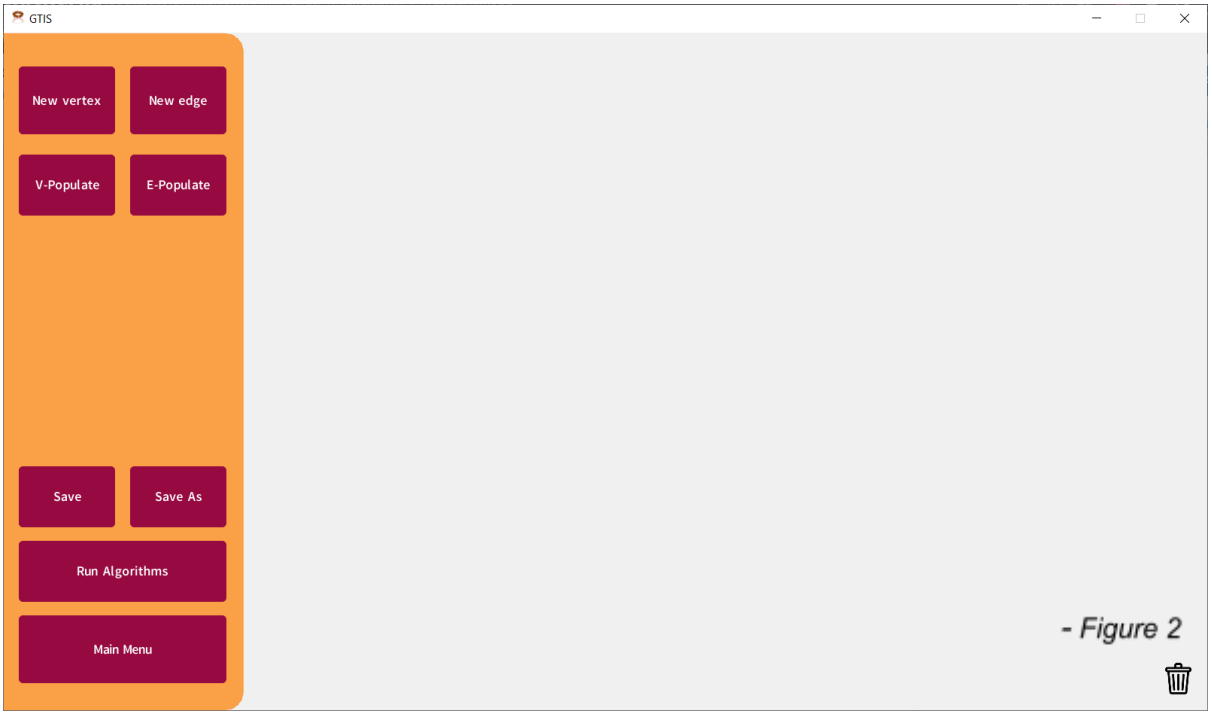
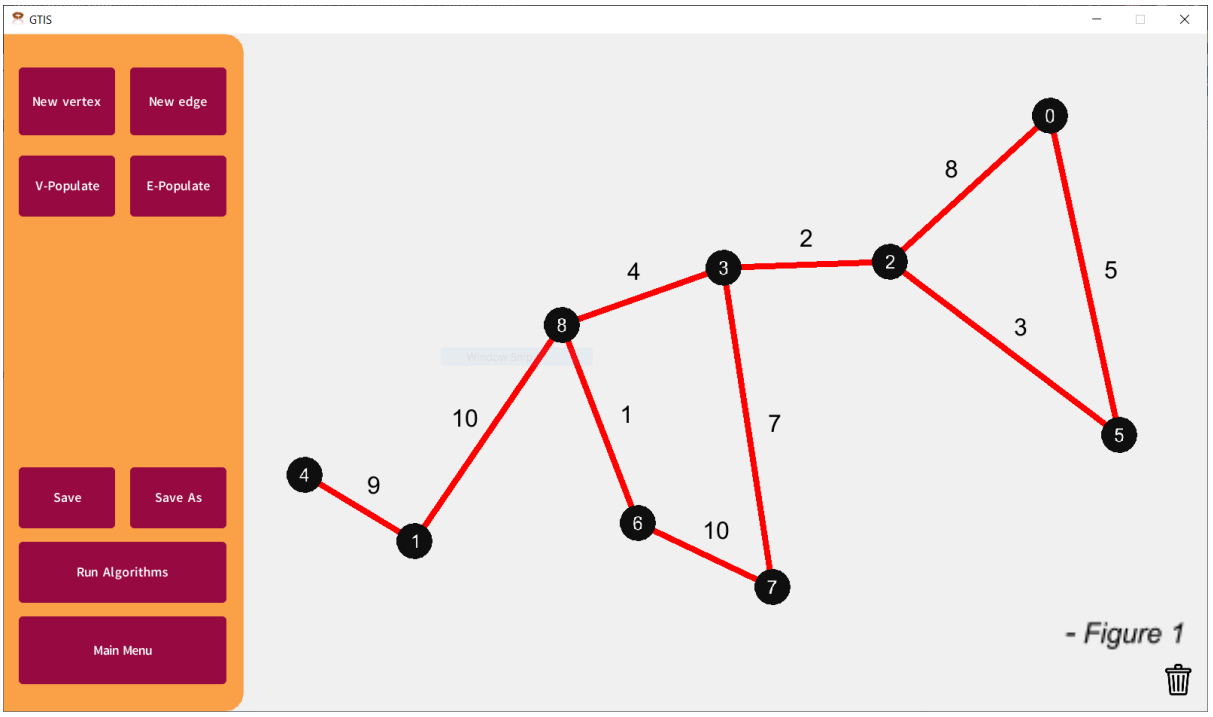


Figure 1 : The screen after pressing on the clear all icon in the bottom right
Figure 2 : The screen after confirming that the user want to clear all

As we can see in figure 2, the clear all button clears all vertices and edges off the graph

Test S5.1:



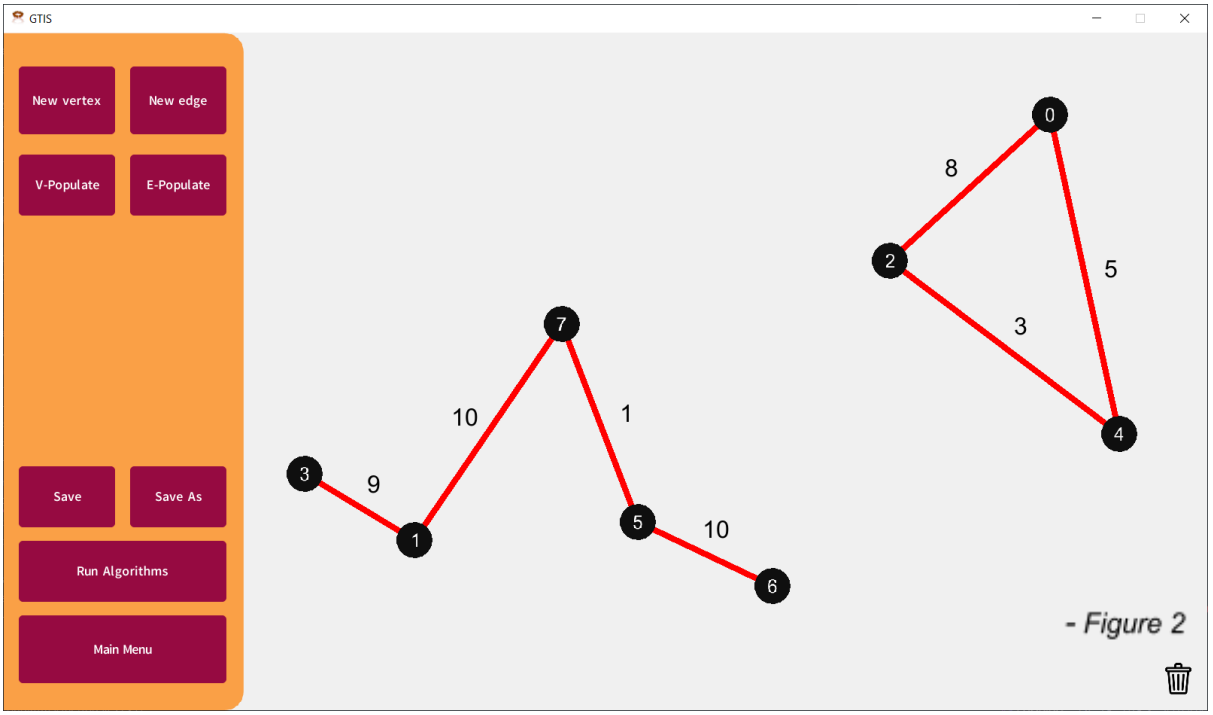
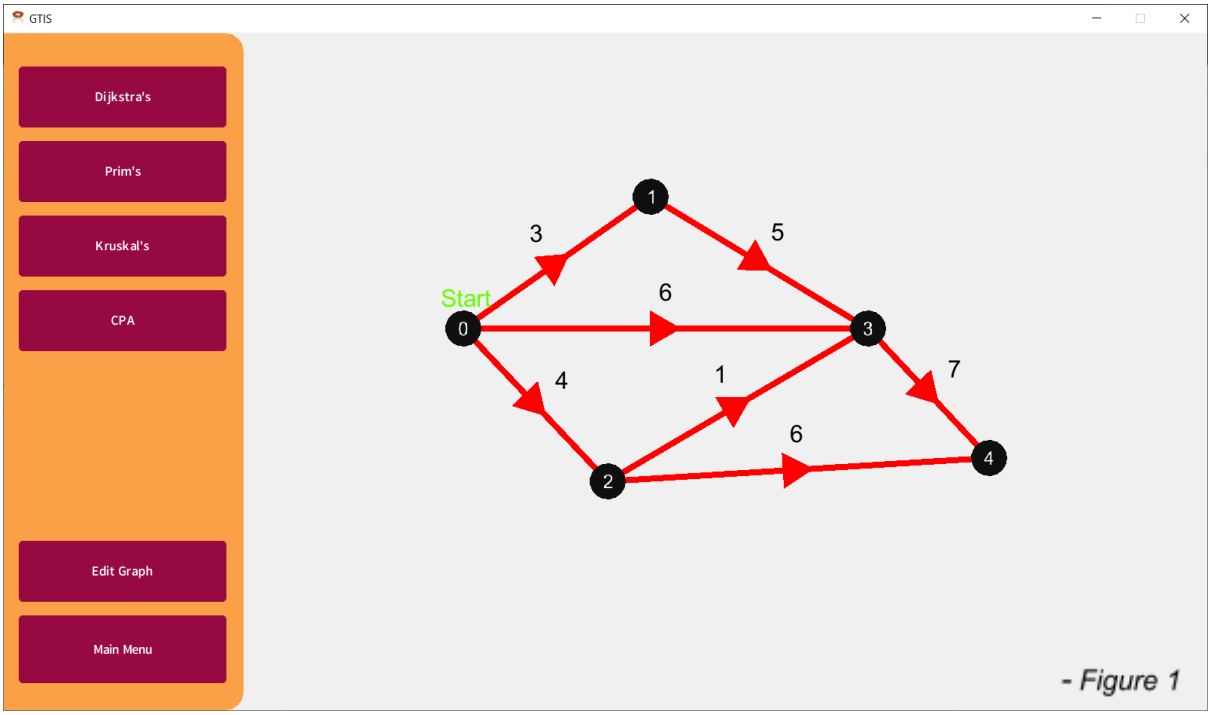


*Figure 1 : The graph **before** right clicking vertex 3*
*Figure 2 : The graph **after** right clicking vertex 3*

As we can see from figure 2, once right clicked vertex 3 and all the edges connected to it are removed from the graph and all other vertex numbers are changed to accomodate

Algorithm Executor:

Test A1.1:



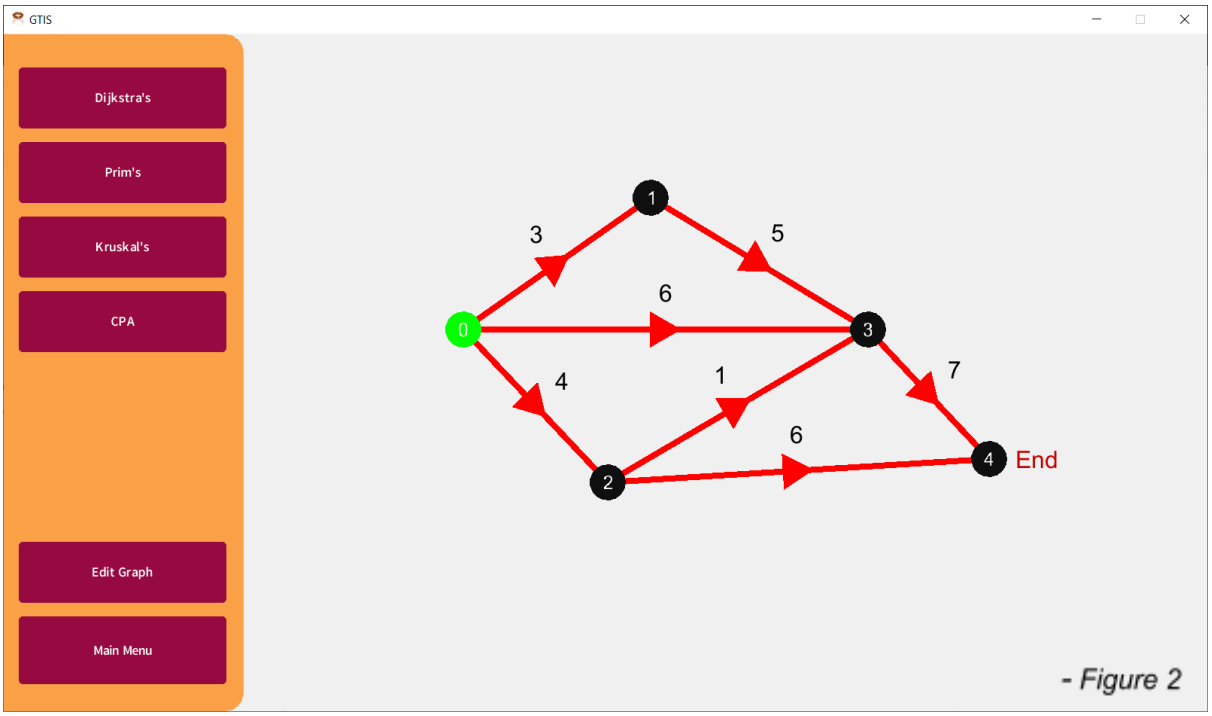


Figure 1 : The program prompting the user to pick the start vertex after pressing the Djikstra's button
Figure 2 : The program prompting the user to pick the end vertex after selecting the start vertex

As we can see from figure 1, when the Djikstra's button is pressed the user is prompted to pick a start vertex. Once this start vertex has been selected, the user is then prompted to pick an end vertex

Test A1.2 & A1.3:

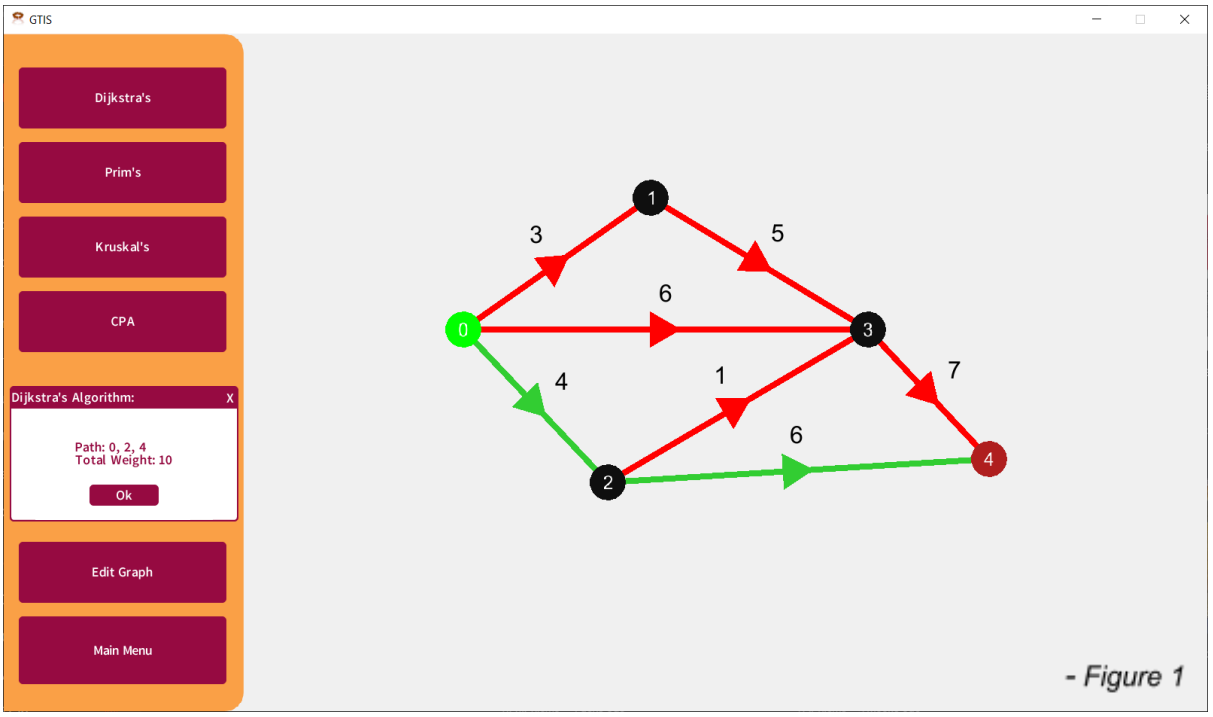


Figure 1 : The graph after selecting both the start and end vertex

As we can see from figure 1, the program wrote the vertices in the shortest path and the total weight of the shortest path to the screen. We can also see that this path is highlighted in green

Test A1.4:

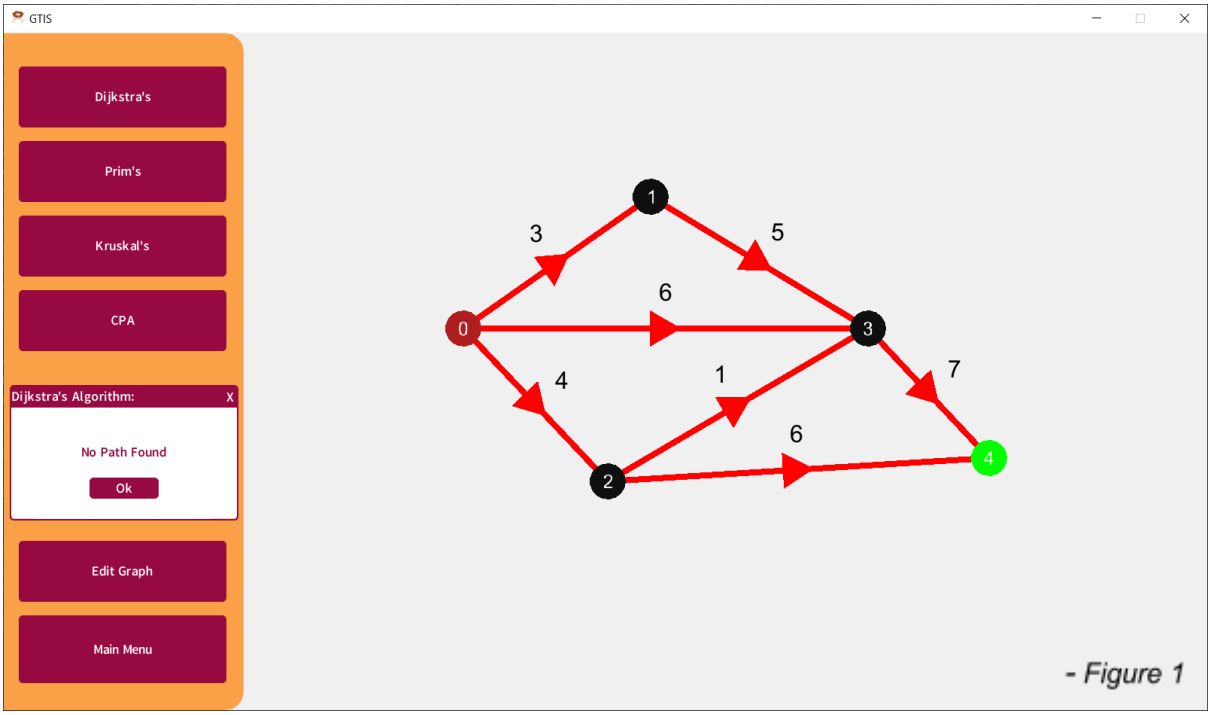


Figure 1 : The graph after selecting a start and end vertex where there is no path between the two

As we can see from figure 1 the program will notify the user if there is no possible path between the two vertices

Test A2.1:

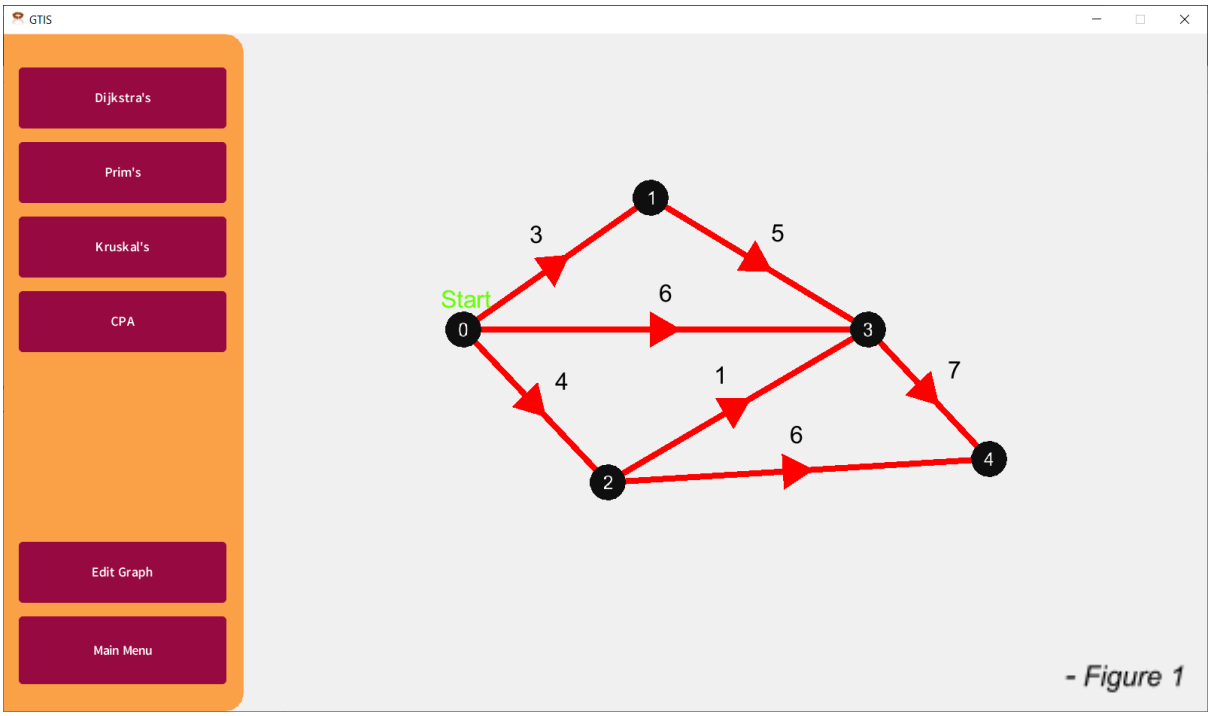


Figure 1 : The program prompting the user to pick the start vertex after pressing the Prim's button

As we can see from figure 1, when the Djikstra's button is pressed the user is prompted to pick a start vertex

Test A2.2 & A2.3:

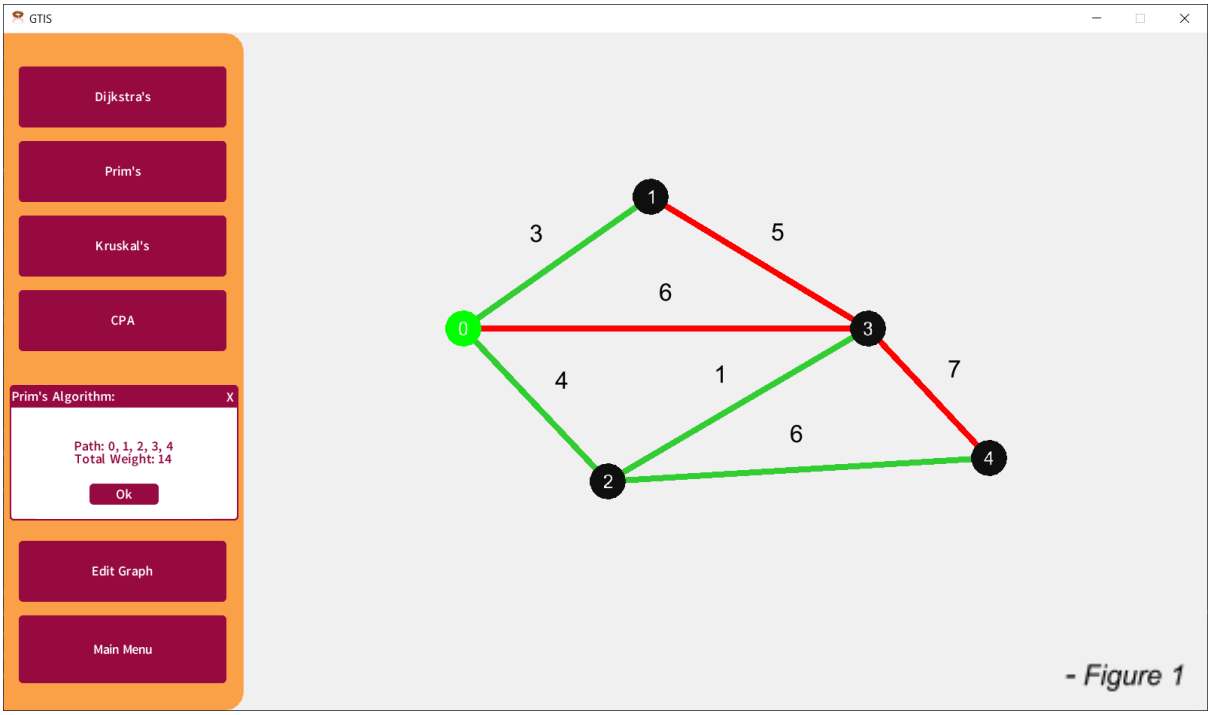


Figure 1 : The graph after selecting the start vertex

As we can see from figure 1, when the user selects the start vertex the program writes the vertices in the minimum spanning tree and the total weight of the subgraph.We can also see that this path is highlighted in green.

Test A3.1 & A3.2 :

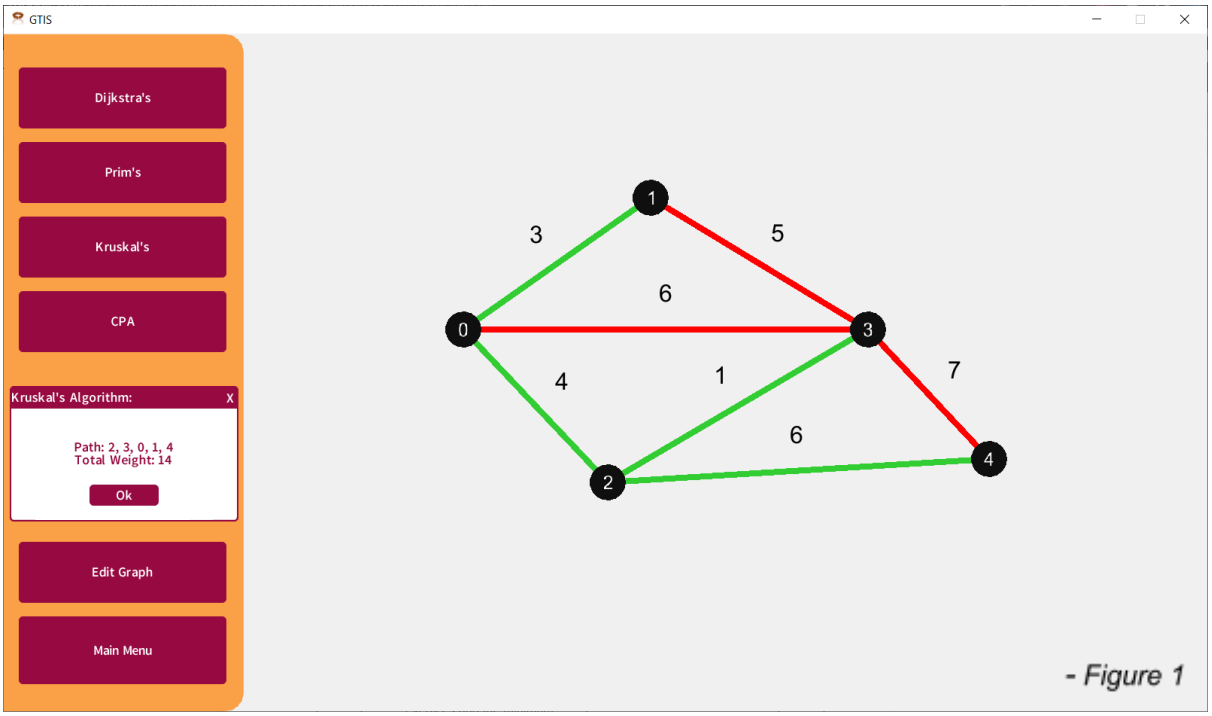


Figure 1 : The graph after pressing the Kruskal's button

As we can see from figure 1, when the user presses the Kruskal's button the program writes the vertices in the minimum spanning tree and the total weight of the subgraph. We can also see that this path is highlighted in green.

Test A4.1:

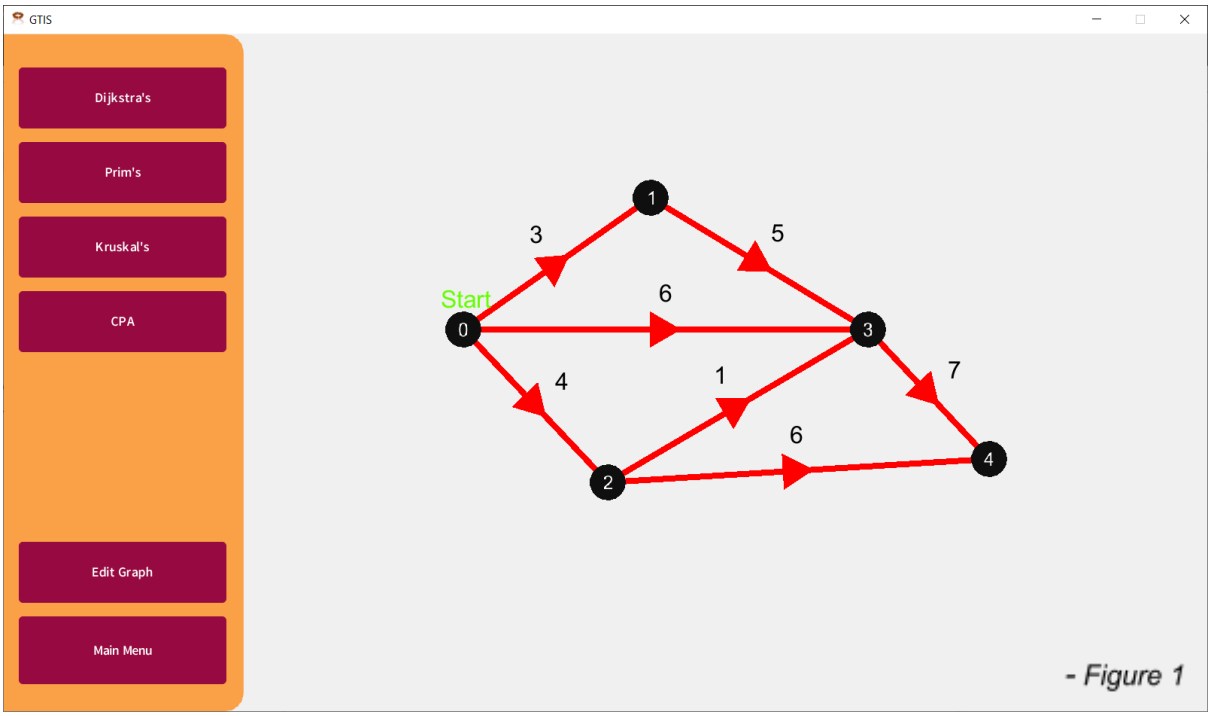


Figure 1 : The graph after pressing the Kruskal's button

As we can see from figure 1, the user if prompted to choose a start vertex when the CPA button is pressed

Test A4.2 & A4.3 & A4.4:

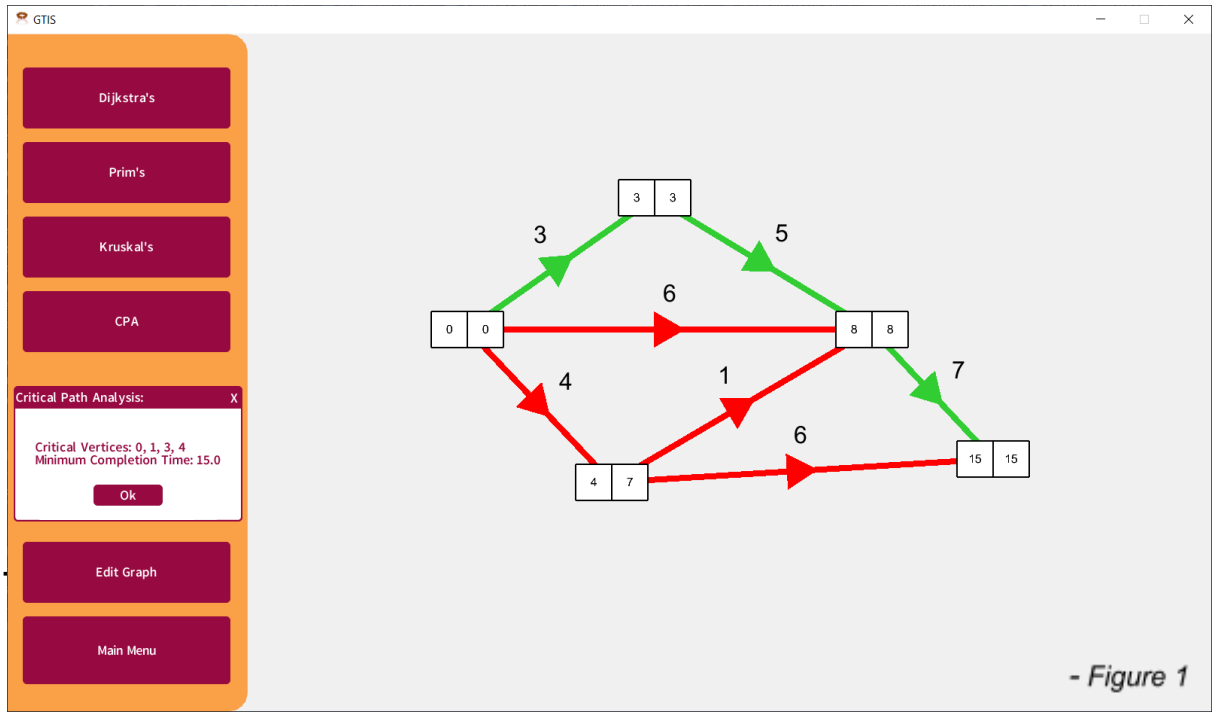
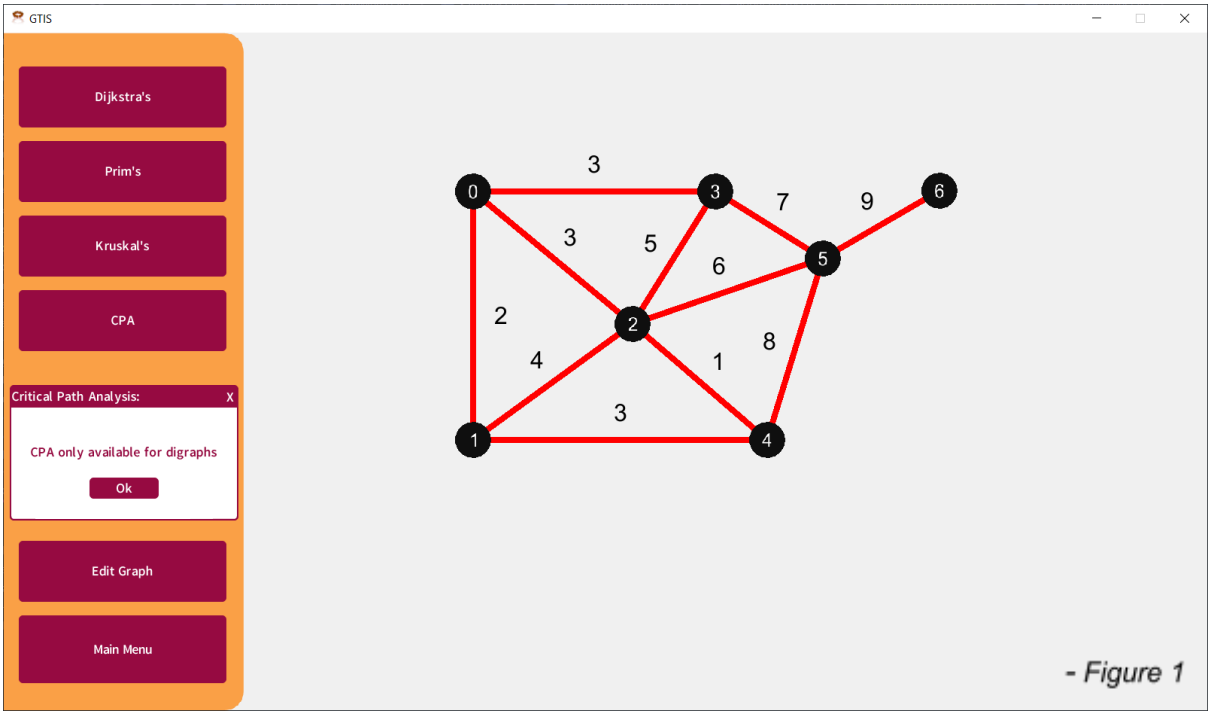


Figure 1 : The graph after the CPA button was pressed and the first vertex was chosen

As we can see from figure 1, when the user presses the CPA button the program writes the vertices in the critical path and the total weight of the critical path to the screen. We can also see that the critical path is highlighted in green and the earliest and latest event times are displayed at each vertex.

Test A4.2:



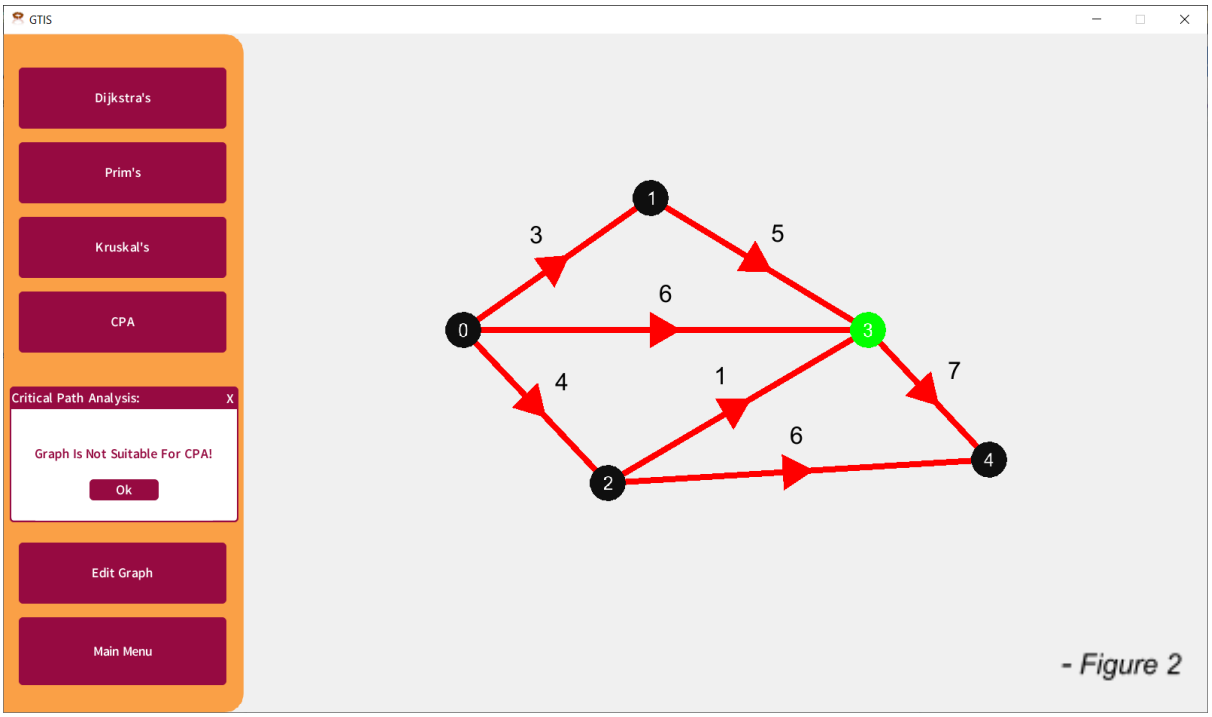


Figure 1 : An attempt to run CPA on an undirected graph
Figure 2 : An attempt to run CPA on an directed graph, but not at the activity network's start vertex

As we can see from figure 1 and figure 2, the program will notify the user if the graph is unsuitable for critical path analysis

Evaluation:

Objective Summary:

Obj	Objective Summary:	Done:
1.1	The user should have the choice to create either unidirectional or bidirectional graphs	✔
1.2	The user should be able to edit these graphs by adding new vertices and edges with custom weights	✔
1.3	The program should be able to to create its own connected graph with a user chosen number of vertices and random edge weights	✔
1.3.1	The user should be able to choose a number of vertices between 1 and 400	✔
1.3.2	The user should be able to choose the range of the weights between 0 and 20 and whether they are integer or float values	✔
1.4	The user should be able to create as many graphs as they want from scratch or from an existing graph	✔
2.1	The user should be able to save their own graph with custom file names	✔
2.1.1	The program should store whether the graph is unidirectional or bidirectional, the coordinates of each vertex, where each edge starts and ends and the weight of each edge	✔
2.2	The user should be able to edit any graph and overwrite that graph's save file	✔
2.3	The user should be able to share their own graphs with other users by sending them the graph's file	✔
3.1	The user should be able to run Dijkstra's algorithm on any (valid) graph	✔
3.1.1	The shortest path should be displayed on the screen in a different colour and with its total weight	✔
3.2	The user should be able to run Prim's algorithm on any (valid) graph	✔
3.2.1	The minimum spanning tree should be displayed on the screen in a different colour with its total weight	✔
3.3	The user should be able to run Kruskal's algorithm on any (valid) graph	✔
3.3.1	The minimum spanning tree should be displayed on the screen in a different colour with its total weight	✔
3.4	The user should be able to run the CPA algorithm on any (valid) graph	✔
3.4.1	The critical path should be displayed on screen along with the earliest and latest event time for each vertex and the earliest completion time	✔
4.1	The GUI should be simple and user friendly	✔
4.2	The Algorithms should be fast and effective	✔
4.3	The user should be notified if the chosen algorithm cannot be executed on their graph	✔

User Feedback:

I have shown my project to one of the further maths teachers and one of the further maths students who I interviewed before I started the project to get their thoughts on how the application has lived up to their expectations and what there is still to improve. Here is what they said:

T1: Further Maths Teacher 1 **S2:** Further Maths Student 2

Q: How has the application lived up to your expectations?

T1: This project is exactly what I was hoping for when you first came to me with the idea. The final product totally lives up to my expectations about being simple and intuitive. I especially like the way that you can change the names of vertices and edges to what you prefer and change the size of the vertices as that would help the students visibility when I'm demonstrating something in a classroom.

S2: The application is exactly what was explained to me when you first told me about this project. The UI is very user friendly and creating my own graphs is very quick and easy. I am happy that you took my idea of removing vertices and included it as I think that copying graphs from paper would be a lot more frustrating when you make a mistake.

Q: Is this application something that you can see yourself using?

T1: Yes, definitely! This program is exactly what was needed and I am very excited to use it in the classroom. I am sure that many other teachers would find this a very useful tool and I can see it being used widely in many schools.

S2: Yes. This program would be very helpful in my revision for checking my method and answers to exam questions. It would also be very useful to be able to share the graph file with other students when working together on a question.

Q: What could still be improved / added?

T1: I like the range of algorithms available, but you could always add more. I also think that it would be nice to see the algorithm being executed rather than just the end result as that would really benefit the students who struggle with understanding how to execute the algorithms.

S2: I really like the way that you can remove vertices by right clicking on them, but I still think that a ‘history’ tab would be very useful like in photoshop. That way you can easily change a small part of the graph without needing to add and remove vertices manually.

Response to user feedback:

I am very happy with the response to how my program turned out as these are the end users and will be the ones using it. I am also glad with the fact that both of the interviewees would use this application as a tool to help students understand more about graph theory as that was always the intended use of the product.

I agree that adding more algorithms would only be a good thing and that being able to see the steps of the algorithm would also benefit the student's understanding of how to execute the algorithms. I can also see how adding a history function would be very useful to the user and would make their experience with the product feel a lot more effortless and might be something that I will consider in the future of this application.

End note:

Overall I am very happy with how my application turned out. It was exactly what I had hoped to accomplish and I felt that there was nothing that stood out as incomplete. I have met all of my objectives and I have developed as much as I had planned for.