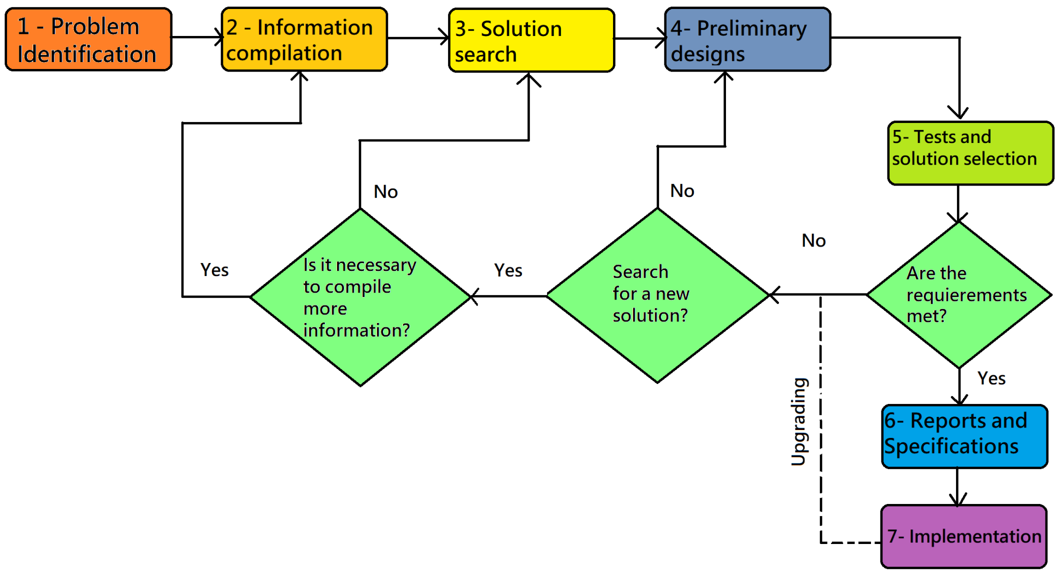
Engineering Method - Metrost App

Problem context[1]:

Metro Trost, is in charge of the construction and operation of the transportation system in Trost, as well as the design, construction and start-up of the Integrated System of Mass Transportation (ISTM), of passengers for the city, based on the definitions technical, legal and financial provided by investment banking. Metrost will be the ISTM of Trost. The system is operated by articulated, standard and complementary buses, which move through trunk, pre-trunk and complementary corridors covering trunk, pre-trunk and feeder routes. Metro Trost requires a first version of a software solution that allows them to: plan the design of the stations' connections, modify the number of stations and the connections between them, as well as provide the shortest route from a specific station to another, taking into account the already existing stations. If, at the moment of a query, there are unconnected stations, the user must be warned.

Development of the solution:

To resolve the above situation, the Engineering Method was chosen to develop the solution following a systematic approach and in line with the problematic situation established.
Based on the description of Paul Wright's "Introduction to Engineering", the following flowchart was defined, the steps of which we will follow in the development of the solution.


1 - Problem Identification
2 - Information compilation
3- Solution search
4- Preliminary designs
5- Tests and solution selection
Are the requierements met?
No
Yes
6- Reports and Specifications
7- Implementation
Upgrading
Search for a new solution?
No
Yes
Is it necessary to compile more information?
No
Yes

[1] Fictional problem context.

Step 1: Problem Identification

The specific needs of the problematic situation are recognized as well as their symptoms and conditions under which it must be resolved.

Identification of needs and symptoms:

- The solution to the problem must be able to find the shortest path between stations efficiently.
- The program must work with the already existing Metrost stations.

Functional requirements:

The program must be able to:

Name	**FR1**: Add stations.
Description	The program allows the user to add stations to the system by interface or in mass by text file.
Inputs	Name of the station, connections with other stations and distance with said stations.
Output	A new station is added to the system.

Name	**FR2**: Delete stations.
Description	The program allows the user to delete stations from the system.
Inputs	Name of the station to delete.
Output	The station is deleted from the system.

Name	**FR3**: Modify a station’s connection.
Description	The program allows the user to modify a station’s connections.
Inputs	The stations to which the specific station will connect and the distances with said stations.
Output	The station overwrites its connections with the new ones.

Name	**FR4**: Save information through csv files.

Description	The program can read and write csv files with a specific format in order to modify and save information.
Inputs	The file's location.
Output	The file is read or written on saving the information in the program.

Name	**FR5**: Find shortest route.
Description	The program can find the shortest route between two stations.
Inputs	The starting station and ending station.
Output	The shortest route between them.

Name	**FR6**: Display the stations' connections.
Description	The program displays the stations and their connections according to what the user chooses.
Inputs	Whether to display the connections as an adjacency list or matrix.
Output	The connections between the stations are displayed.

Name	**FR7**: Display the stations' data.
Description	The program displays the information of every station: Name, connections and distance between them.
Inputs	Nothing.
Output	The information of the stations is displayed.

Name	**FR8**: Notify unconnected stations.
Description	The program warns the user if there are unconnected stations at the moment of a query.
Inputs	Nothing.
Output	The user is warned about the unconnected stations.

Problem Definition:

Metro Trost requires the development of a software application that allows users to add, delete and modify stations and their connections; as well as finding the shortest path between two stations.

Step 2. Information Collection

In order to have complete clarity in the concepts involved, a search is made for the definitions of the terms most closely related to the problem established. It is important to perform this search with recognized and reliable sources to know which elements are part of the problem and which are not.

Sources:

Introduction to Algorithms. Cormen et al. Chapter 22.2 BFS
Introduction to Algorithms. Cormen et al. Chapter 22.3 DFS
Introduction to Algorithms. Cormen et al. Chapter 23 Minimum Spanning Trees
Introduction to Algorithms. Cormen et al. Chapter 24.3 Dijkstra's algorithm

Definitions:

● **Dijkstra**: it's an algorithm for finding the shortest paths between nodes in a graph.

● **Breadth-first search (BFS)**: it's an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

● **Depth-first search (DFS)**: it's an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

● **Minimum spanning tree (MST)**: it's a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Step 3. Finding Creative Solutions

For this step, even if we can think of our own solutions, we look at specialized texts for various ways to implement the different abstract data structures/types (ADTs) necessary and that best suit the different cases of each stage of the process of finding the shortest paths between stations. The methods considered as alternatives, gathered through a brainstorm session, are as follows:

<u>Alternative 1 - Use Weighted Graph</u>

The vertices would represent the stations and the values of the edges would represent the distance between them.

The ADT Weighted Graph:

A weighted graph is a graph where each edge has a value or label associated with it, to represent cost, weight, length, etc.

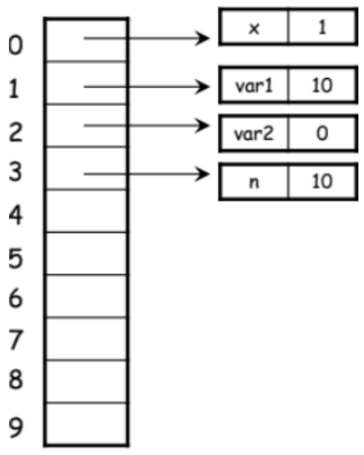
<u>Alternative 2 - Use Hash Table</u>

The keys would be the stations and the values would be the stations with which the station is associated with their respective distances

The ADT Hash Table:

The ADT hash table is a data structure that associates keys with values. The main operation that it efficiently supports is search:

- Allows access to stored items from a generated key.
- It works by transforming the key with a hash function into a hash, a number that the hash table uses to locate the desired value.



Access Procedures:

Constructor operations to construct a Hash Table:

i. createTable() → creates an empty table.

ii. tableInsert(newItem) → inserts newItem into a table in its proper sorted order according to the item’s search key.

Predicate operations to test Hash Tables:

i. isEmpty( ) → determines whether a table is empty.

Selector operations to select items of a Hash Table:

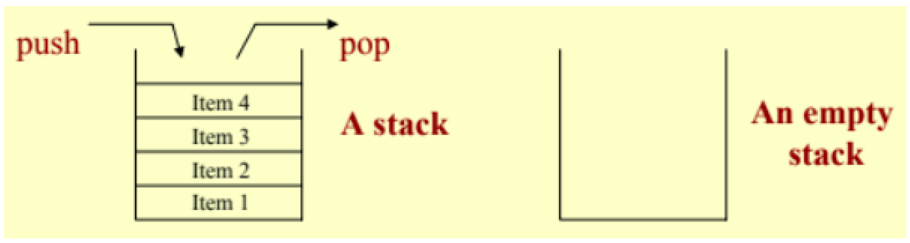
i. tableLength( ) → determines the number of items in the table.
ii. tableDelete(searchKey) → deletes an item with a given search key from the table.
iii. tableRetrieve(searchKey) → retrieves an item with a given search key from a table.

<u>Alternative 3 - Use Stack</u>

The stack would store the stations with their respective relationships and distances

The ADT Stack:

The ADT Stack is a linear sequence of an arbitrary number of items, together with access procedures. The access procedures permit insertions and deletion of items only at one end of the sequence (the "top"). The stack is a list structure, sometimes called a last-in-first-out (or LIFO) list. A stack is either empty, or it consists of a sequence of items. Access is limited to the "top" item on the stack at all times.



Access Procedures:

Operations to construct a stack constructor:

i. createStack( ) → creates an empty Stack.
ii. push(item) → adds an item at the top of the stack.

Preach operations to test Stacks:

i. isEmpty( ) → determine whether a stack is empty.

Selector operations to select items of a Stack:

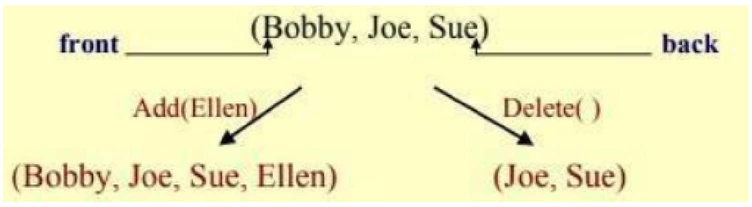
i. top( ) → returns the top item of the stack. It does not change the stack.
ii. pop( ) → changes the stack by removing the top item.

Alternative 4 - Use Queue

The queue would store the stations with their respective connections and distances.

The ADT Queue:

The ADT Queue is a linear sequence of an arbitrary number of items, together with access procedures. The access procedures permit addition only at the back of the queue and deletion of items only at the front of the queue. The queue is a list structure sometimes called a first-in-first-out (or FIFO) list. A queue is either empty, or it consists of a sequence of items. Manipulations or accesses to these items are only permitted at the two ends of the queue.



Access Procedures:

Constructor operations to construct a queue:

i. createQueue( ) → creates an empty queue.
ii. enqueue(item) → adds an item at the end of the queue.

Preach operations to test Queues:

i. isEmpty( ) → determine whether a queue is empty.

Selector operations to select items of a queue:

i. front( ) → returns the item at the front of the queue. It does not change the queue.
ii. dequeue( ) → retrieves and removes the item at the front of the queue.

Alternative 5 - Use Array

The array would store the stations with their respective connections and distances

The ADT Array:

The array is a basic abstract data type that holds an ordered collection of items accessible by an integer index. These items can be anything from primitive types such as integers to more complex types like instances of classes. Since it's an ADT, it doesn't specify an implementation, but it is almost always implemented by an array (data structure) or dynamic array.

Operations to construct Arrays constructor:

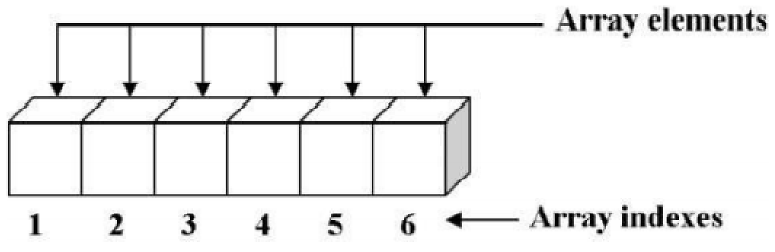
i. createArray( ) → creates an empty array.
ii. add(index, item) → inserts item at position index of an array if 1 ≤ index ≤ size() + 1. If index ≤ size(), items at position index onwards are shifted one position to the right.

Preach operations to test Arrays:

i. isEmpty( ) → determine if an array is empty.

Selector operations to select items of an Array:

i. remove(index) → removes item at position index of an array if 1 ≤ index ≤ size(). Items at position index + 1 onwards are shifted one position to the left.
ii. get(index) → returns item at position index of an array if 1 ≤ index ≤ size().
iii. set(index, newValue) → sets the value of an specified index to a new value.
iv. length( ) → returns number of items in an array.



One-dimensional array with six elements

Alternative 6 - Use List

The list would store the stations with their respective connections and distances.

The ADT List:

The ADT List is a linear sequence of an arbitrary number of items that must be of the same type, together with the following access procedures that can be grouped into three broad categories:

Operations to construct List:

i. createList( ) → creates an empty list.
ii. add(index, item) → inserts item at position index of a list if 1 ≤ index ≤ size() + 1. If index ≤ size(), items at position index onwards are shifted one position to the right.

Preach operations to test Arrays:

i. isEmpty( ) → determine if an array is empty.

Selector operations to select items of an Array:

i. remove(index) → removes item at position index of a list if 1 ≤ index ≤ size(). Items at position index + 1 onwards are shifted one position to the left.
ii. get(index) → returns item at position index of a list if 1 ≤ index ≤ size().
iii. set(index, newValue) → sets the value of an specified index to a new value.
iv. size( ) → returns number of items in a list.

<u>Alternative 7 - Use N-ary Tree</u>

The node would save the stations and the children of the nodes would be the associated stations. At each node the distance from his father and son is kept

A N-nary Tree is a collection of nodes. Each node stores the address of multiple nodes. Every node stores the address of its children and the very first node's address will be stored in a separate pointer called root.

The N-ary trees have the following properties:

- Many children at every node.
- The number of nodes for each node is not known in advance.

Operations to construct N-ary Tree constructor:

i. createTree( ) → creates an empty tree.
ii. insert(key, value) → inserts the value in the list of children of the parent key

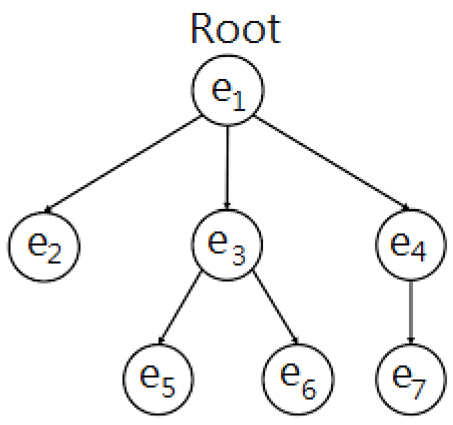
Preach operations to test N-ary Tree:

i. isEmpty( ) → determine if a N-ary Tree is empty by checking the existence of its root.

Selector operations to select items of a N-ary Tree:

i. remove(value) → removes an element from the N-ary Tree through its value.
ii. search(value) → searches an element in a N-ary Tree through its value.
iii. firstChildrenThenRootTraversal() → traverses a tree looking first at the children and then at the root.

iv. firstRootThenChildrenTraversal() → traverses a tree looking first at the root and then at the children.



Step 4. Transitioning Ideas to Preliminary Designs

The first thing we do in this step is to dismiss ideas that are not feasible.

In this sense, the list and array are discarded because it is not necessary to save all stations in a linear fashion when each station already has its connections and distances.

The Queue and the Stack are discarded due to their access procedures (FIFO and LIFO respectively) being impractical in solving this problem.

Careful review of the other alternatives leads to the following:

Alternative 1 - Use Weighted Graph:

➢ The ability to connect any two vertices in the graph and put a value in that connection (weight in the edge) allows for a practical representation of the problem.

Alternative 2 - Use Hash Table:

➢ The use of a hash table would allow for the implementation of a solution but the characteristics and strongpoints of this data structure does not suit the problem at hand since it requires a lot of connections and not necessarily efficient time complexity.

Alternative 3 - Use N-ary Tree:

➢ This type of tree would allow to add an arbitrary amount of stations and take advantage of the connections between them.

Step 5. Evaluation and Selection of the Best Solution

Criteria

The criteria for evaluating solution alternatives should be defined and based on this outcome, the solution that best meets the needs of the problem stated is chosen. The criteria we chose in this case are the ones listed below. A numerical value has been established next to each one in order to establish a weight indicating which of the possible values of each criterion have the most weight (i.e., they are most desirable).

Criterion A. Access procedures time complexity (worst case). A solution that takes less time to access stations and/or its connections.

- [1] $O(n!)$
- [2] $O(2^n)$
- [3] $O(n^2)$
- [4] $O(n \log n)$
- [5] $O(n)$
- [6] $O(\log n)$, $O(1)$

Criterion B. Implementation behavior. A solution that behaves similarly to the elements of the model is preferred. The behavior can be:

- [3] Similar
- [2] Partially similar
- [1] Not similar

Evaluation

By evaluating the above criteria in the alternatives that are maintained, we get the following table:

	Criterion A	**Criterion B**	**Total**
Alternative 1	5 $O(n)$	3 Similar	8
Alternative 2	3 $O(n^2)$	1 Not similar	4

Alternative 3	5 O(n)	2 Partially similar	7

Selection

According to the previous evaluation Alternative 1 should be selected, as it obtained the highest score according to the defined criteria.

Step 6. Preparation of Reports and Specifications

Problem Specification:

Problem: Adding, deleting and modifying stations and their connections; as well as finding the shortest path between two stations.

Input: The already existing stations found in the city of Trost.

Output: Shortest route from one station to another (given); the information from every station.

Considerations

The following cases should be taken into account for testing this process:

1. The stations that already existed are taken into account.
2. The query of the shortest path between two stations may yield no result if there is no route connecting both stations.
3. There may be unconnected stations.
4. Distance between stations must be positive numbers.

ADT Design

ADT Generic Directed Weighted Graph

Graph = (V, E, W)
V = set of vertices = $\{v_1, v_2, v_3, \dots, v_n\} \in$ *Graph*
e_i = pairs of ordered vertices = $(v_i, v_j) \mid v_i, v_j \in V$
E = set of edges = $\{e_1, e_2, e_3, \dots, e_n\} \in$ *Graph*

W = set of associated weights to each edge = $\{w_1, w_2, w_3, \ldots, w_n\} \in Graph \land w_i \in \mathbb{R}$
SizeVertices = $|V|$
SizeEdges = $|E|$
SizeWeights = $|W|$

{ inv: $\forall e_i, e_j \in E \Rightarrow e_i \neq e_j$

$\forall v_i, v_j \in V \Rightarrow (v_i, v_j) \neq (v_j, v_i)$

SizeVertices ≥ 0

SizeEdges ≥ 0

SizeWeights ≥ 0

SizeWeights = *SizeEdges* }

Primitive operations				**Operation type**
CreateGraph		→	Graph	Constructor
AddVertex	Graph x Vertex	→	Graph	Modifier
HasEdge	Graph x Vertex1 x Vertex2	→	Boolean	Analyzer
AddEdge	Graph x Vertex1 x Vertex2	→	Graph	Modifier
GetEdgeWeight	Graph x Edge	→	Number	Analyzer
DeleteVertex	Graph x Vertex	→	Graph	Modifier
Dijkstra	Graph x Vertex1 x Vertex2	→	String	Analyzer
DeleteEdge	Graph x Edge	→	Graph	Modifier
ShowGraph	Graph x Vertex	→	String	Analyzer

CreateGraph()

"Creates an empty Graph"

{ pre: *TRUE* }

{ post: Graph = Ø }

AddVertex(vertex, Graph)

"Adds a new vertex in the Graph"

{ pre: ∃ *Graph*, vertex ∉ *Graph* }

{ post: |V| + 1, vertex ∈ *Graph* }

HasEdge(vertex1,vertex2, Graph)

“Checks if there is an edge between the given vertices”

{ pre: ∃ Graph, Graph ≠ Ø, ∃ Vertex1, ∃ Vertex2, Vertex1 ∧ Vertex2 ∈ Graph }

{ post: True if the edge exists, otherwise False }

GetEdgeWeight(edge, Graph)

“Returns the weight of the given edge”

{ pre: ∃ Graph, Graph ≠ Ø, ∃ Edge, Edge ∈ Graph }

{ post: Edge.weight ∈ W }

AddEdge(Graph, Vertex1, Vertex2, weight)

“Creates a new edge between two given vertices”

{ pre: ∃ Graph, Graph ≠ Ø, ∃ Vertex1, ∃ Vertex2, (∄ e_i = (Vertex1,Vertex2)) }

{ post: (∃ e_i = (Vertex1,Vertex2)), |E| + 1, |W| + 1 }

DeleteVertex(Graph, vertex)

“Deletes a vertex from the Graph with its incident edges”

{ pre: ∃ Graph, Graph ≠ Ø, vertex ∈ *Graph* }

{ post: vertex ∉ Graph, |V| - 1, ∄ edge | vertex ∈ edge }

Dijkstra(Graph,Vertex1,Vertex2)

“Returns a list of the vertices to follow from Vertex1 to Vertex2 using the path with the smallest sum of weight in its edges”

{ pre: ∃ Graph, Graph ≠ Ø, ∃ Vertex1, ∃ Vertex2, Vertex1 ∧ Vertex2 ∈ Graph }

{ post: String ∈ Text }

DeleteEdge(Graph, Edge)

“Deletes an edge from the Graph”

{ pre: ∃ Graph, Graph ≠ Ø, ∃ Edge, Edge ∈ Graph}

{ post: Edge ∉ Graph, |E| - 1, |W| - 1 }

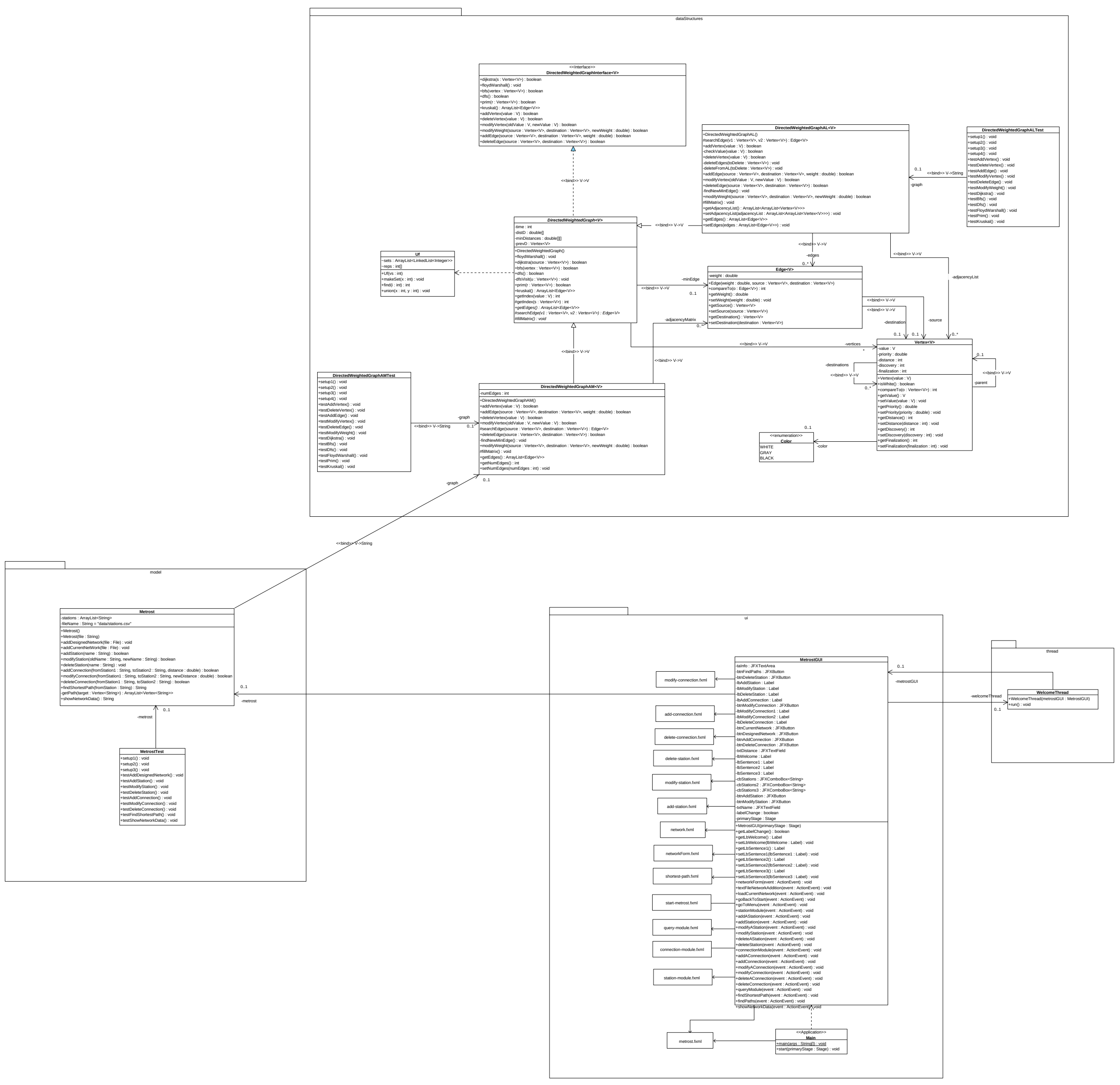
ShowGraph(Graph, Vertex)

“Displays the graph’s information: each vertex, its adjacent vertices and the weight in its edges”

{ pre: ∃ Graph, Graph ≠ Ø }

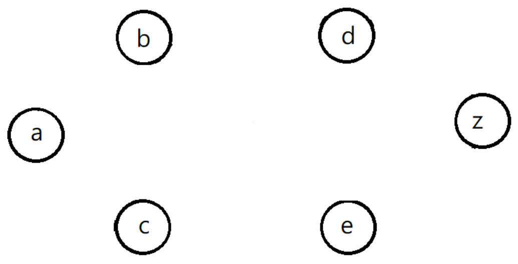
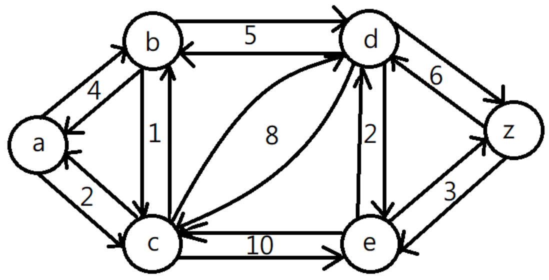
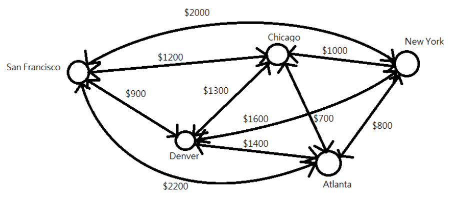
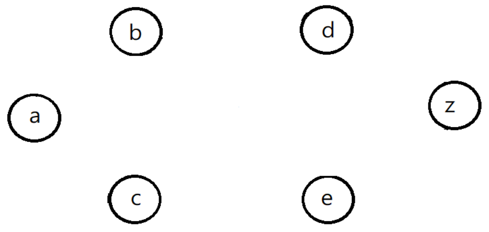
{ post: String ∈ Text }

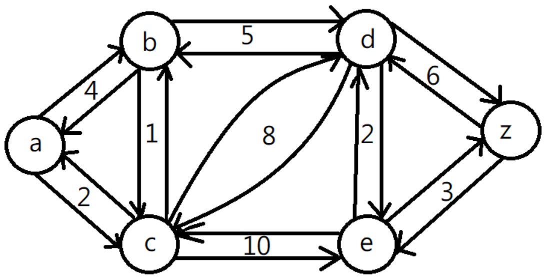
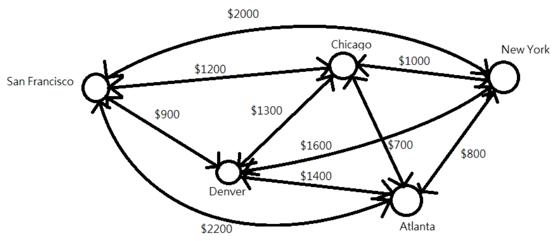
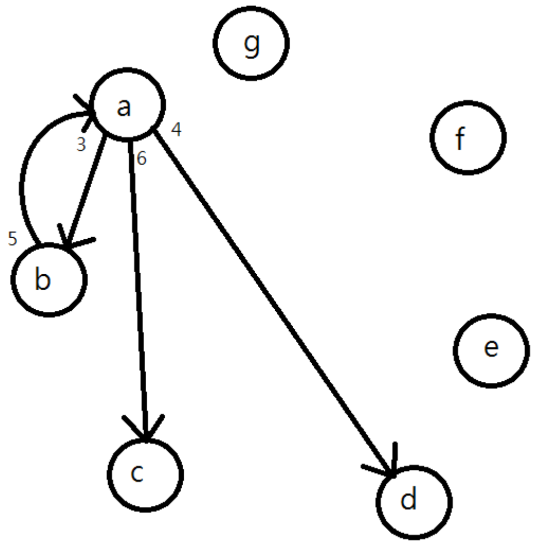
Class Diagram

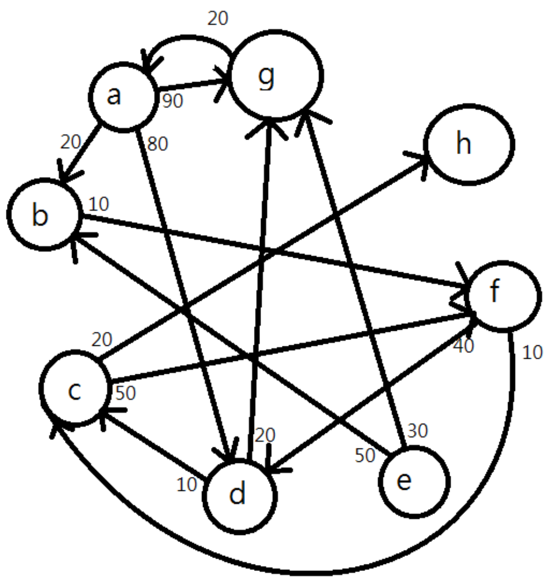

dataStructures
<<Interface>>
DirectedWeightedGraphInterface<V>
+dijkstra(s : Vertex<V>) : boolean
+floydWarshall() : void
+bfs(vertex : Vertex<V>) : boolean
+dfs() : boolean
+prim(r : Vertex<V>) : boolean
+kruskal() : ArrayList<Edge<V>>
+addVertex(value : V) : boolean
+deleteVertex(value : V) : boolean
+modifyVertex(oldValue : V, newValue : V) : boolean
+modifyWeight(source : Vertex<V>, destination : Vertex<V>, newWeight : double) : boolean
+addEdge(source : Vertex<V>, destination : Vertex<V>, weight : double) : boolean
+deleteEdge(source : Vertex<V>, destination : Vertex<V>) : boolean
<<bind>> V->V
DirectedWeightedGraphAL<V>
+DirectedWeightedGraphAL()
#searchEdge(v1 : Vertex<V>, v2 : Vertex<V>) : Edge<V>
+addVertex(value : V) : boolean
-checkValue(value : V) : boolean
+deleteVertex(value : V) : boolean
-deleteEdges(toDelete : Vertex<V>) : void
-deleteFromAL(toDelete : Vertex<V>) : void
+addEdge(source : Vertex<V>, destination : Vertex<V>, weight : double) : boolean
+modifyVertex(oldValue : V, newValue : V) : boolean
+deleteEdge(source : Vertex<V>, destination : Vertex<V>) : boolean
-findNewMinEdge() : void
+modifyWeight(source : Vertex<V>, destination : Vertex<V>, newWeight : double) : boolean
#fillMatrix() : void
+getAdjacencyList() : ArrayList<ArrayList<Vertex<V>>>
+setAdjacencyList(adjacencyList : ArrayList<ArrayList<Vertex<V>>>) : void
+getEdges() : ArrayList<Edge<V>>
+setEdges(edges : ArrayList<Edge<V>>) : void
DirectedWeightedGraphALTest
+setup1() : void
+setup2() : void
+setup3() : void
+setup4() : void
+testAddVertex() : void
+testDeleteVertex() : void
+testAddEdge() : void
+testModifyVertex() : void
+testDeleteEdge() : void
+testModifyWeight() : void
+testDijkstra() : void
+testBfs() : void
+testDfs() : void
+testFloydWarshall() : void
+testPrim() : void
+testKruskal() : void
0..1
<<bind>> V->String
-graph
<<bind>> V->V
<<bind>> V->V
-edges
0..*
<<bind>> V->V
-adjacencyList
DirectedWeightedGraph<V>
-time : int
-distD : double[]
-minDistances : double[][]
-prevD : Vertex<V>
+DirectedWeightedGraph()
+floydWarshall() : void
+dijkstra(source : Vertex<V>) : boolean
+bfs(vertex : Vertex<V>) : boolean
+dfs() : boolean
-dfsVisit(u : Vertex<V>) : void
+prim(r : Vertex<V>) : boolean
+kruskal() : ArrayList<Edge<V>>
+getIndex(value : V) : int
#getIndex(s : Vertex<V>) : int
+getEdges() : ArrayList<Edge<V>>
#searchEdge(v1 : Vertex<V>, v2 : Vertex<V>) : Edge<V>
#fillMatrix() : void
Uf
~sets : ArrayList<LinkedList<Integer>>
~reps : int[]
+Uf(vs : int)
+makeSet(x : int) : void
+find(i : int) : int
+union(x : int, y : int) : void
-minEdge
<<bind>> V->V
0..1
Edge<V>
-weight : double
+Edge(weight : double, source : Vertex<V>, destination : Vertex<V>)
+compareTo(o : Edge<V>) : int
+getWeight() : double
+setWeight(weight : double) : void
+getSource() : Vertex<V>
+setSource(source : Vertex<V>)
+getDestination() : Vertex<V>
+setDestination(destination : Vertex<V>)
-adjacencyMatrix
0..*
<<bind>> V->V
<<bind>> V->V
-destination
-source
0..1
0..1
0..*
<<bind>> V->V
-vertices
*
<<bind>> V->V
Vertex<V>
-value : V
-priority : double
-distance : int
-discovery : int
-finalization : int
+Vertex(value : V)
+isWhite() : boolean
+compareTo(o : Vertex<V>) : int
+getValue() : V
+setValue(value : V) : void
+getPriority() : double
+setPriority(priority : double) : void
+getDistance() : int
+setDistance(distance : int) : void
+getDiscovery() : int
+setDiscovery(discovery : int) : void
+getFinalization() : int
+setFinalization(finalization : int) : void
-destinations
<<bind>> V->V
0..*
0..1
<<bind>> V->V
-parent
0..1
<<enumeration>>
Color
WHITE
GRAY
BLACK
-color
<<bind>> V->V
DirectedWeightedGraphAMTest
+setup1() : void
+setup2() : void
+setup3() : void
+setup4() : void
+testAddVertex() : void
+testDeleteVertex() : void
+testAddEdge() : void
+testModifyVertex() : void
+testDeleteEdge() : void
+testModifyWeight() : void
+testDijkstra() : void
+testBfs() : void
+testDfs() : void
+testFloydWarshall() : void
+testPrim() : void
+testKruskal() : void
-graph
<<bind>> V->String
0..1
DirectedWeightedGraphAM<V>
-numEdges : int
+DirectedWeightedGraphAM()
+addVertex(value : V) : boolean
+addEdge(source : Vertex<V>, destination : Vertex<V>, weight : double) : boolean
+deleteVertex(value : V) : boolean
+modifyVertex(oldValue : V, newValue : V) : boolean
#searchEdge(source : Vertex<V>, destination : Vertex<V>) : Edge<V>
+deleteEdge(source : Vertex<V>, destination : Vertex<V>) : boolean
-findNewMinEdge() : void
+modifyWeight(source : Vertex<V>, destination : Vertex<V>, newWeight : double) : boolean
#fillMatrix() : void
+getEdges() : ArrayList<Edge<V>>
+getNumEdges() : int
+setNumEdges(numEdges : int) : void
-graph
0..1
<<bind>> V->String
model
Metrost
-stations : ArrayList<String>
-fileName : String = "data/stations.csv"
+Metrost()
+Metrost(file : String)
+addDesignedNetwork(file : File) : void
+addCurrentNetWork(file : File) : void
+addStation(name : String) : boolean
+modifyStation(oldName : String, newName : String) : boolean
+deleteStation(name : String) : void
+addConnection(fromStation1 : String, toStation2 : String, distance : double) : boolean
+modifyConnection(fromStation1 : String, toStation2 : String, newDistance : double) : boolean
+deleteConnection(fromStation1 : String, toStation2 : String) : boolean
+findShortestPath(fromStation : String) : String
-getPath(target : Vertex<String>) : ArrayList<Vertex<String>>
+showNetworkData() : String
0..1
-metrost
-metrost
0..1
MetrostTest
+setup1() : void
+setup2() : void
+setup3() : void
+testAddDesignedNetwork() : void
+testAddStation() : void
+testModifyStation() : void
+testDeleteStation() : void
+testAddConnection() : void
+testModifyConnection() : void
+testDeleteConnection() : void
+testFindShortestPath() : void
+testShowNetworkData() : void
ui
MetrostGUI
-taInfo : JFXTextArea
-btnFindPaths : JFXButton
-btnDeleteStation : JFXButton
-lbAddStation : Label
-lbModifyStation : Label
-lbDeleteStation : Label
-lbAddConnection : Label
-btnModifyConnection : JFXButton
-lbModifyConnection1 : Label
-lbModifyConnection2 : Label
-lbDeleteConnection : Label
-btnCurrentNetwork : JFXButton
-btnDesignedNetwork : JFXButton
-btnAddConnection : JFXButton
-btnDeleteConnection : JFXButton
-txtDistance : JFXTextField
-lbWelcome : Label
-lbSentence1 : Label
-lbSentence2 : Label
-lbSentence3 : Label
-cbStations : JFXComboBox<String>
-cbStations2 : JFXComboBox<String>
-cbStations3 : JFXComboBox<String>
-btnAddStation : JFXButton
-btnModifyStation : JFXButton
-txtName : JFXTextField
-labelChange : boolean
-primaryStage : Stage
+MetrostGUI(primaryStage : Stage)
+getLabelChange() : boolean
+getLbWelcome() : Label
+setLbWelcome(lbWelcome : Label) : void
+getLbSentence1() : Label
+setLbSentence1(lbSentence1 : Label) : void
+getLbSentence2() : Label
+setLbSentence2(lbSentence2 : Label) : void
+getLbSentence3() : Label
+setLbSentence3(lbSentence3 : Label) : void
+networkForm(event : ActionEvent) : void
+textFileNetworkAddition(event : ActionEvent) : void
+loadCurrentNetwork(event : ActionEvent) : void
+goBackToStart(event : ActionEvent) : void
+goToMenu(event : ActionEvent) : void
+stationModule(event : ActionEvent) : void
+addAStation(event : ActionEvent) : void
+addStation(event : ActionEvent) : void
+modifyAStation(event : ActionEvent) : void
+modifyStation(event : ActionEvent) : void
+deleteAStation(event : ActionEvent) : void
+deleteStation(event : ActionEvent) : void
+connectionModule(event : ActionEvent) : void
+addAConnection(event : ActionEvent) : void
+addConnection(event : ActionEvent) : void
+modifyAConnection(event : ActionEvent) : void
+modifyConnection(event : ActionEvent) : void
+deleteAConnection(event : ActionEvent) : void
+deleteConnection(event : ActionEvent) : void
+queryModule(event : ActionEvent) : void
+findShortestPath(event : ActionEvent) : void
+findPaths(event : ActionEvent) : void
+showNetworkData(event : ActionEvent) : void
modify-connection.fxml
add-connection.fxml
delete-connection.fxml
delete-station.fxml
modify-station.fxml
add-station.fxml
network.fxml
networkForm.fxml
shortest-path.fxml
start-metrost.fxml
query-module.fxml
connection-module.fxml
station-module.fxml
metrost.fxml
<<Application>>
Main
+main(args : String[]) : void
+start(primaryStage : Stage) : void
0..1
-metrostGUI
thread
-welcomeThread
0..1
WelcomeThread
+WelcomeThread(metrostGUI : MetrostGUI)
+run() : void

Design of scenarios and test cases

Scenario configuration

Name	Class	Scenario
setup1	DirectedWeightedGraphALTest	graph, a DirectedWeightedGraphAL of String is initialized as DirectedWeightedGraphAL<>().
setup2	DirectedWeightedGraphALTest	
setup3	DirectedWeightedGraphALTest	 b 5 d 4 6 a 1 8 2 z 2 3 c 10 e
setup4	DirectedWeightedGraphALTest	 $2000 Chicago $1000 New York San Francisco $1200 $900 $1300 $1600 $700 $800 Denver $1400 $2200 Atlanta
setup1	DirectedWeightedGraphAMTest	graph, a DirectedWeightedGraphAM of String is initialized as DirectedWeightedGraphAM<>().
setup2	DirectedWeightedGraphALTest	

setup3	DirectedWeightedGraphAMTest	 a b c d e z 4 2 1 5 8 10 2 6 3
setup4	DirectedWeightedGraphAMTest	 $2000 San Francisco $1200 Chicago $1000 New York $900 $1300 $1600 $700 $800 $1400 Denver $2200 Atlanta
setup1	metrostTest	metrost is initialized and is told to write in file “test/stations.csv”, no stations or connections are loaded from any file
setup2	metrostTest	metrost is initialized and is told to write in file “test/stations.csv”, it loads from file "test/inputExample.csv", the network loaded is as follows 
setup3	metrostTest	metrost is initialized and is told to write in file “test/stations.csv”, it loads from file "test/inputExample2.csv", the network

		loaded is as follows  20 a g 90 h 20 80 b 10 f 20 40 10 c 50 20 30 50 e 10 d

Test case design

Test Objective: Verify that a new vertex is successfully added to the graph				
Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAL	addVertex	setup1	“a” “b” “c” “d” “e” “z”	graph’s adjacency list and vertices list now has 6 elements, graph does not have edges. All 6 elements in the adjacency list are empty lists since the vertices are not connected. graph’s first vertex has value “a” graph’s second vertex has value “b” graph’s third vertex has value “c” graph’s fourth vertex has value “d” graph’s fifth vertex has value “e” graph’s sixth vertex has value “z”

Test Objective: Verify that a vertex is successfully deleted from the graph.				
Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeight	deleteVertex	setup2	“m”	The method returns false and graph

edGraphAL				still has 6 vertices.
DirectedWeight edGraphAL	deleteVertex	setup2	“d”	The method returns true and graph now has 5 vertices. graph’s first vertex has value “a” graph’s second vertex has value “b” graph’s third vertex has value “c” graph’s fourth vertex has value “e” graph’s fifth vertex has value “z”. graph’s adjacency list has 5 elements now, all of them empty lists.
DirectedWeight edGraphAL	deleteVertex	setup3	“a”	The method returns true. graph now has 5 vertices. graph’s first vertex has value “b” graph’s second vertex has value “c” graph’s third vertex has value “d” graph’s fourth vertex has value “e” graph’s fifth vertex has value “z”. graph’s adjacency list has 5 elements now. the first vertex’s adjacency list has two elements and does not contain a vertex with value “a”. the second vertex’s adjacency list has three elements and does not contain a vertex with value “a”. The other vertices’ adjacency lists remain unchanged.

Test Objective: Verify that a new edge is successfully added to the graph				
Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeight edGraphAL	addEdge	setup2	the first vertex in graph, the first vertex in graph, 10 the second vertex in graph, the first vertex in graph, -1 the third vertex in graph, the second vertex in graph, 0 the first vertex in graph, the second vertex in graph, 10 the first vertex in graph, the second	The method returns false The method returns false The method returns false The method returns true The method returns false The method returns true The method returns true The method returns true The method returns true

			vertex in graph, 10 the second vertex in graph, the third vertex in graph, 5 the third vertex in graph, the fourth vertex in graph, 2 the fourth vertex in graph, the fifth vertex in graph, 7 the fifth vertex in graph, the sixth vertex in graph, 12 the sixth vertex in graph, the first vertex in graph, 13 the sixth vertex in graph, the fourth vertex in graph, 16	The method returns true The method returns true graph now has 7 edges. the following is reflected in the adjacencyList: the first vertex has 1 adjacent vertex the second vertex has 1 adjacent vertex the third vertex has 1 adjacent vertex the fourth vertex has 1 adjacent vertex the fifth vertex has 1 adjacent vertex the sixth vertex has 2 adjacent vertices. the fourth edge's source vertex and destination vertex have value "d" and "e" respectively, as well as a weight of 7

Test Objective: Verify that a vertex's value is successfully modified

Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAL	modifyVertex	setup2	"a","b" "b","f" "a","b"	method returns false method returns true method returns true graph's first vertex now has value "b" graph's second vertex now has value "f"
DirectedWeightedGraphAL	modifyVertex	setup3	"z","c" "z","f"	method returns false method returns true None of graph's edges have a vertex (source or destination) with value "z"

Test Objective: Verify that an edge is successfully deleted from the graph.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAL	deleteEdge	setup3	graph's first vertex, graph's sixth vertex graph's fourth vertex, null graph's fifth vertex, a new Vertex with value "h" graph's first vertex, graph's second vertex graph's first vertex, graph's second vertex graph's first vertex, graph's third vertex graph's second vertex, graph's third vertex	method returns false method returns false method returns false method returns true method returns false method returns true method returns true graph now has 15 edges graph's first vertex now has no adjacent vertices graph's second vertex now has two adjacent vertices this is reflected in graph's adjacency list

Test Objective: Verify that an edge's weight is successfully modified				
Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAL	modifyWeight	setup3	graph's first vertex, graph's sixth vertex, 12.8 null, graph's sixth vertex, 12.8 graph's first vertex, graph's second vertex, 0 graph's first vertex, graph's third vertex, -1 graph's second vertex, graph's fourth vertex, 5.7	method returns false method returns false method returns false method returns false method returns true method returns true method returns true graph's eleventh edge has weight 5.7 graph's sixth edge has weight 0.8 graph's second edge has

			graph's second vertex, graph's third vertex, 0.8 graph's second vertex, graph's first vertex, 2.7	weight 2.7 graph has 18 edges graph's second vertex has 3 adjacent vertices, this is reflected in graph's adjacency list

Test Objective: Verify that a minimum weight path can be found successfully between two vertices.

Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAL	dijkstra	setup1	a new vertex with value "x"	method returns false
DirectedWeightedGraphAL	dijkstra	setup3	null a new vertex with value x vertex with value x (added to graph)	method returns false method returns false method returns true, graph's distD={∞,∞,∞,∞,∞,0}, graph's prevD={null,null,null,null,null, null}
DirectedWeightedGraphAL	dijkstra	setup3	graph's first vertex	method returns true, graph's distD={0,3,2,8,10,13}, graph's prevD={nil,c,a,b,d,e}

Test Objective: Verify that the connected vertices in a graph can be found given a source vertex.

Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAL	bfs	setup2	a new vertex with value "x" graph's first vertex	method returns false method returns true graph's first vertex is black, the rest are white.
DirectedWeightedGraphAL	bfs	setup2	graph's first vertex after adding edges between vertices: 0 and 1 (weight 1), 0 and 2 (weight 2),5 and 4 (weight 3), 5 and 3 (weight 4)	method returns true, graph's first three vertices are black their distances are 0, 1 and 1 and their parents are null, first vertex and first vertex respectively vertices 3 through 5 are black, their distances are ∞ and their parents are null

DirectedWeightedGraphAL	bfs	setup3	graph’s second vertex	method returns true and all of graph’s vertices are black graph’s second vertex’s distance is 0 and parent is null graph’s first vertex’s distance is 1 and parent is the second vertex graph’s third vertex’s distance is 1 and parent is the second vertex graph’s fourth vertex’s distance is 1 and parent is the second vertex graph’s fifth vertex’s distance is 2 and parent is the third vertex graph’s sixth vertex’s distance is 2 and parent is the fourth vertex

Test Objective: Verify that the connected vertices in a graph can be found given a source vertex.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAL	dfs	setup1		method returns false.
DirectedWeightedGraphAL	dfs	setup2		method returns true. First vertex’s discovery time and finalization time are 1 and 2 respectively, second vertex’s discovery time and finalization time are 3 and 4 respectively, third vertex’s discovery time and finalization time are 5 and 6 respectively, fourth vertex’s discovery time and finalization time are 7 and 8 respectively, fifth vertex’s discovery time and finalization time are 9 and 10 respectively and sixth vertex’s discovery time and finalization time are 11 and 12 respectively. all vertices are black
DirectedWeightedGraphAL	dfs	setup3		method returns true. First vertex’s discovery time and finalization time are 1 and 12 respectively, second vertex’s discovery time and finalization time are 2 and 11 respectively, third vertex’s discovery time and

				finalization time are 3 and 10 respectively, fourth vertex's discovery time and finalization time are 5 and 8 respectively, fifth vertex's discovery time and finalization time are 4 and 9 respectively and sixth vertex's discovery time and finalization time are 6 and 7 respectively. all vertices are black

Test Objective: Verify that a minimum weight path can be found successfully between all vertices.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAL	floydWarshall	setup2		The matrix representing the minimum distances between each vertex has 0 in its diagonal and ∞ in every other cell.
DirectedWeightedGraphAL	floydWarshall	setup3		The matrix representing the minimum distances between each vertex has the correct values

Test Objective: Verify that a minimum spanning tree can be found successfully in the graph.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAL	prim	setup4	graph's fifth vertex ("Atlanta")	method returns true the fifth vertex's parent is null the second and third vertices' parent is the fifth vertex the first vertex's parent is the second vertex the fourth vertex's parent is the first vertex the weight of the minimum spanning tree is 3600 the fifth vertex's key is 0 the second vertex's key is 700 the third vertex's key is 800 the first vertex's key is 1200 the fourth vertex's key is 900
DirectedWeightedGraphAL	kruskal	setup4		the method returns 4 edges the first edge connects

				Chicago and Atlanta with weight 700 the second edge connects New York and Atlanta with weight 800 the third edge connects San Francisco and Denver with weight 900 the fourth edge connects Chicago and San Francisco with weight 1200

Test Objective: Verify that a new vertex is successfully added to the graph

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAM	addVertex	setup1	“a” “b” “c” “d” “e” “z”	graph’s vertices list now has 6 elements and its adjacency matrix has 6 rows and 6 columns, each cell holds null. graph does not have edges. All 6 elements in the adjacency list are empty lists since the vertices are not connected. graph’s first vertex has value “a” graph’s second vertex has value “b” graph’s third vertex has value “c” graph’s fourth vertex has value “d” graph’s fifth vertex has value “e” graph’s sixth vertex has value “z”

Test Objective: Verify that a vertex is successfully deleted from the graph.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAM	deleteVertex	setup2	“m”	The method returns false and graph still has 6 vertices and its adjacency matrix has 6 rows and 6 columns.
DirectedWeightedGraphAM	deleteVertex	setup2	“d”	The method returns true and graph now has 5 vertices and its adjacency matrix has 5 rows and 5 columns graph’s first vertex has value “a” graph’s second vertex has value “b”

				graph’s third vertex has value “c” graph’s fourth vertex has value “e” graph’s fifth vertex has value “z”.
DirectedWeightedGraphAM	deleteVertex	setup3	“a”	The method returns true. graph now has 5 vertices. graph’s first vertex has value “b” graph’s second vertex has value “c” graph’s third vertex has value “d” graph’s fourth vertex has value “e” graph’s fifth vertex has value “z”. graph’s adjacency matrix has 5 rows and 5 columns, it now contains 14 edges: the first row has two edges. the second row has three edges. The other rows remain unchanged.

Test Objective: Verify that a new edge is successfully added to the graph

Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAM	addEdge	setup2	the first vertex in graph, a new vertex with value “h”,7.9 the first vertex in graph, the first vertex in graph, 10 the second vertex in graph, the first vertex in graph, -1 the third vertex in graph, the second vertex in graph, 0 the first vertex in graph, the second vertex in graph, 10 the first vertex in graph, the second vertex in graph, 10 the second vertex in graph, the third vertex in graph, 5 the third vertex in graph, the fourth	The method returns false The method returns false The method returns false The method returns false The method returns true The method returns false The method returns true The method returns true The method returns true The method returns true The method returns true The method returns true graph now has 7 edges. the following is reflected in the adjacency matrix: the first row has 1 edge the second row has 1 edge

			vertex in graph, 2 the fourth vertex in graph, the fifth vertex in graph, 7 the fifth vertex in graph, the sixth vertex in graph, 12 the sixth vertex in graph, the first vertex in graph, 13 the sixth vertex in graph, the fourth vertex in graph, 16	the third row has 1 edge the fourth row has 1 edge the fifth row has 1 edge the sixth row has 2 edges. there's an edge located in the matrix in row 3, column 4, its source vertex and destination vertex have value "d" and "e" respectively, and it has a weight of 7

Test Objective: Verify that a vertex's value is successfully modified

Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAM	modifyVertex	setup2	"a","b" "b","f" "a","b"	method returns false method returns true method returns true graph's first vertex now has value "b" graph's second vertex now has value "f"
DirectedWeightedGraphAM	modifyVertex	setup3	"z","c" "z","f"	method returns false method returns true All the source vertices in the sixth row of the adjacency matrix have value "f"

Test Objective: Verify that an edge is successfully deleted from the graph.

Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAM	deleteEdge	setup3	graph's first vertex, graph's sixth vertex graph's fourth vertex, null	method returns false method returns false method returns false method returns true

			graph’s fifth vertex, a new Vertex with value “h” graph’s first vertex, graph’s second vertex graph’s first vertex, graph’s second vertex graph’s first vertex, graph’s third vertex graph’s second vertex, graph’s third vertex	method returns false method returns true method returns true graph now has 15 edges the adjacency matrix’s first row now has 0 edges the adjacency matrix’s second row now has two edges

Test Objective: Verify that an edge’s weight is successfully modified				
Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAM	modifyWeight	setup3	graph’s first vertex, graph’s sixth vertex, 12.8 null, graph’s sixth vertex, 12.8 graph’s first vertex, graph’s second vertex, 0 graph’s first vertex, graph’s third vertex, -1 graph’s second vertex, graph’s fourth vertex, 5.7 graph’s second vertex, graph’s third vertex, 0.8 graph’s second vertex, graph’s first vertex, 2.7	method returns false method returns false method returns false method returns false method returns true method returns true method returns true The edge located in row 1, column 3 of the matrix has weight 5.7 The edge located in row 1, column 2 of the matrix has weight 0.8 The edge located in row 1, column 0 of the matrix has weight 2.7 graph has 18 edges the matrix’s second row has 3 edges

Test Objective: Verify that a minimum weight path can be found successfully between two vertices.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAM	dijkstra	setup1	a new vertex with value “x”	method returns false
DirectedWeightedGraphAM	dijkstra	setup3	null a new vertex with value x vertex with value x (added to graph)	method returns false method returns false method returns true, graph’s distD={∞,∞,∞,∞,∞,0}, graph’s prevD={null,null,null,null,null, null}
DirectedWeightedGraphAM	dijkstra	setup3	graph’s first vertex	method returns true, graph’s distD={0,3,2,8,10,13}, graph’s prevD={nil,c,a,b,d,e}

Test Objective: Verify that the connected vertices in a graph can be found given a source vertex.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAM	bfs	setup2	a new vertex with value “x” graph’s first vertex	method returns false method returns true graph’s first vertex is black, the rest are white.
DirectedWeightedGraphAM	bfs	setup2	graph’s first vertex after adding edges between vertices: 0 and 1 (weight 1), 0 and 2 (weight 2),5 and 4 (weight 3), 5 and 3 (weight 4)	method returns true, graph’s first three vertices are black their distances are 0, 1 and 1 and their parents are null, first vertex and first vertex respectively vertices 3 through 5 are black, their distances are ∞ and their parents are null
DirectedWeightedGraphAM	bfs	setup3	graph’s second vertex	method returns true and all of graph’s vertices are black graph’s second vertex’s distance is 0 and parent is null graph’s first vertex’s distance is 1 and parent is the second vertex graph’s third vertex’s distance is 1 and parent is the second

				vertex graph’s fourth vertex’s distance is 1 and parent is the second vertex graph’s fifth vertex’s distance is 2 and parent is the third vertex graph’s sixth vertex’s distance is 2 and parent is the fourth vertex

Test Objective: Verify that the connected vertices in a graph can be found given a source vertex.

Class	Method	Scenario	Input values	Result
DirectedWeightedGraphAM	dfs	setup1		method returns false.
DirectedWeightedGraphAM	dfs	setup2		method returns true. First vertex’s discovery time and finalization time are 1 and 2 respectively, second vertex’s discovery time and finalization time are 3 and 4 respectively, third vertex’s discovery time and finalization time are 5 and 6 respectively, fourth vertex’s discovery time and finalization time are 7 and 8 respectively, fifth vertex’s discovery time and finalization time are 9 and 10 respectively and sixth vertex’s discovery time and finalization time are 11 and 12 respectively. all vertices are black
DirectedWeightedGraphAM	dfs	setup3		method returns true. First vertex’s discovery time and finalization time are 1 and 12 respectively, second vertex’s discovery time and finalization time are 2 and 11 respectively, third vertex’s discovery time and finalization time are 3 and 10 respectively, fourth vertex’s discovery time and finalization time are 5 and 8 respectively, fifth vertex’s discovery time and finalization time are 4 and 9 respectively and sixth vertex’s discovery time and finalization time are 6 and 7 respectively.

				all vertices are black

Test Objective: Verify that a minimum weight path can be found successfully between all vertices.				
Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAM	floydWarshall	setup2		The matrix representing the minimum distances between each vertex has 0 in its diagonal and ∞ in every other cell.
DirectedWeightedGraphAM	floydWarshall	setup3		The matrix representing the minimum distances between each vertex has the correct values

Test Objective: Verify that a minimum spanning tree can be found successfully in the graph.				
Class	**Method**	**Scenario**	**Input values**	**Result**
DirectedWeightedGraphAM	prim	setup4	graph's fifth vertex ("Atlanta")	method returns true the fifth vertex's parent is null the second and third vertices' parent is the fifth vertex the first vertex's parent is the second vertex the fourth vertex's parent is the first vertex the weight of the minimum spanning tree is 3600 the fifth vertex's key is 0 the second vertex's key is 700 the third vertex's key is 800 the first vertex's key is 1200 the fourth vertex's key is 900
DirectedWeightedGraphAM	kruskal	setup4		the method returns 4 edges the first edge connects Chicago and Atlanta with weight 700 the second edge connects New York and Atlanta with weight 800 the third edge connects San Francisco and Denver with weight 900 the fourth edge connects

				Chicago and San Francisco with weight 1200

Test Objective: Verify that metrost can import a network given a file with the correct format.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	addDesigned Network	setup1	file = new File("test/input Example.csv");	metrost now has 7 stations with values “a”,”b”,”c”,”d”,”e”,”f” and “g” respectively. metrost’s graph’s adjacency matrix has 7 rows and columns. metrost’s graph has 7 vertices with values “a”,”b”,”c”,”d”,”e”,”f” and “g” respectively. metrost’s graph’s adjacency matrix holds: an edge with weight 3 in row 0 column 1, an edge with weight 6 in row 0 column 2, an edge with weight 5 in row 1 column 0 and an edge with weight 4 in row 0 column 3 and null everywhere else.

Test Objective: Verify that metrost can add a station properly.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	addStation	setup1	“Prueba”	metrost now has 1 station with value “Prueba” and no connections. This is reflected in the file where metrost keeps its data ("test/stations.csv" in this case)
Metrost	addStation	setup2	“Prueba”	metrost now has 8 stations with values “a”,”b”,”c”,”d”,”e”,”f”,”g” and “Prueba” respectively. metrost has connections between

				stations “a” and “b” with weight 3, “a” and “c” with weight 6, “b” and “a” with weight 5 and “a” and “d” with weight 4. This is reflected in the file where metrost keeps its data ("test/stations.csv" in this case)

Test Objective: Verify that metrost can modify a station properly.

Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	modifyStation	setup2	“a”, “b” “b”, “h” “a”, “b”	False True, the name of station b changes to h. The file test/station.csv also changes the name True, the name of station a changes to b. The file test/station.csv also changes the name

Test Objective: Verify that metrost can delete a station properly.

Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	deleteStation	setup2	“a”	metrost now has 6 stations with values ”b”,”c”,”d”,”e”,”f” and “g” respectively. metrost has no connections between any stations. This is reflected in the file where metrost keeps its data ("test/stations.csv" in this case)

Test Objective: Verify that metrost can add a connection properly.

Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	addConnectio	setup2	“a”, “a”, 12.0	False

	n		"a", "b", 11.0 "a", "f", 0 "a", "f", 7.5 "f", "g", 10.2	False False True metrost now has a new edge with source "a", destination "f" and weight 7.5. The file test/station.csv also add the edge True metrost now has a new edge with source "f", destination "g" and weight 10.2. The file test/station.csv also add the edge

Test Objective: Verify that metrost can modify a connection properly.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	modifyConnection	setup2	"e", "f", 15 "a", "b", 3.5	False True. The connection between stations "a" and "b" now has weight 3.5. This is reflected in metrost's graph and in the file where metrost keeps its data ("test/stations.csv" in this case)

Test Objective: Verify that metrost can delete a connection properly.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	deleteConnection	setup2	"a", "f" "a", "b"	False True. The connection between stations "a" and "b" does not exist now. Metrost still has 7 stations with values "a","b","c","d","e","f" and "g" respectively. This is reflected in metrost's graph and in the file

				where metrost keeps its data ("test/stations.csv" in this case)

Test Objective: Verify that metrost can find the shortest path properly.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	findShortestPath	setup3	“a”	Method returns the following String: “To b: Distance: 20.0 Path: a——>b To c: Distance: 40.0 Path: a——>b——>f——> To d: Distance: 70.0 Path: a——>b——>f——>d To e: Distance: N/A Path: Non-existent To f: Distance: 30.0 Path: a——>b——>f To g: Distance: 90.0 Path: a——>g To h: Distance: 60.0 Path: a——>b——>f——>c——> h”

Test Objective: Verify that metrost displays network data properly.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Metrost	showNetworkData	setup3		Method returns the following String: “Number of stations 8 Stations a b c d e f g h Station,Connected station,Distance

				between them a,b,20.0 a,d,80.0 a,g,90.0 b,f,10.0 c,f,50.0 c,h,20.0 d,c,10.0 d,g,20.0 e,b,50.0 e,g,30.0 f,c,10.0 f,d,40.0 g,a,20.0”

Step 7 - Design implementation

Link to the source folder of the design implementation: https://github.com/TheLordJuanes/metrost-app