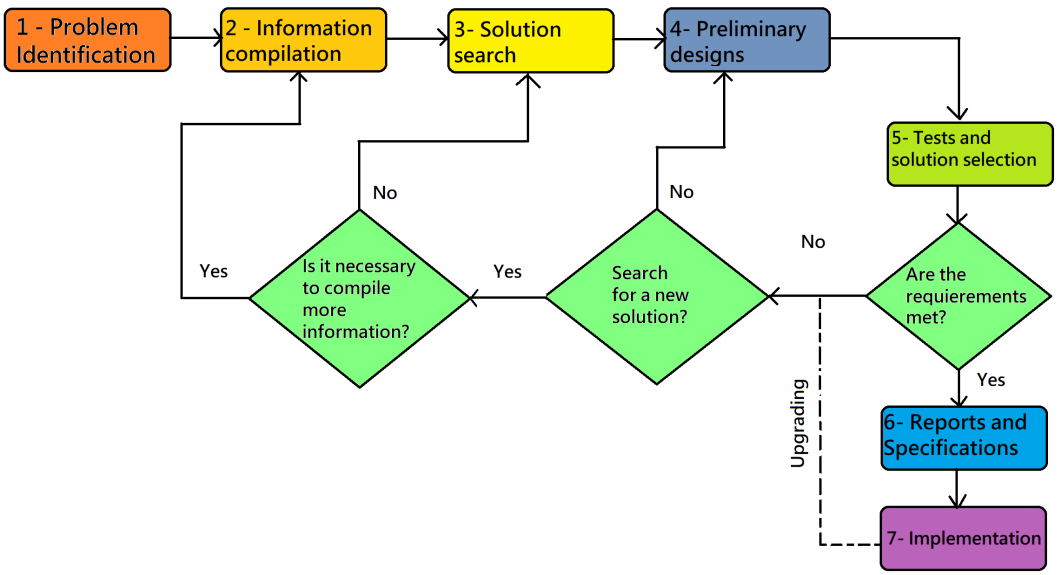
Engineering Method - Metrost App

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] 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:

Alternative 1 - Use Weighted Graph

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.

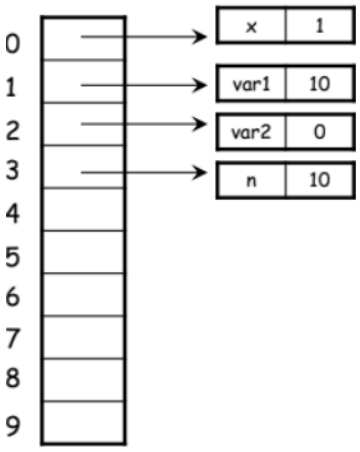
Alternative 2 - Use Hash Table

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:

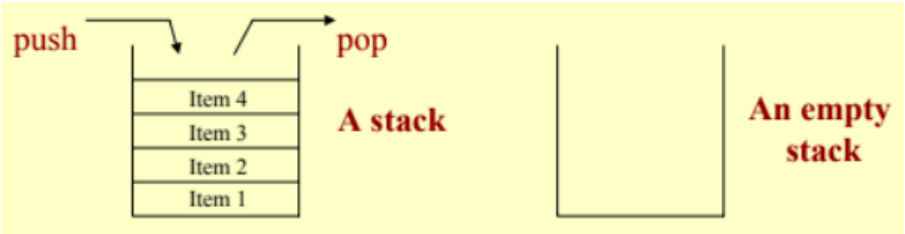
i. tableLength( ) → determines the number of items in the table.
ii. tableDelete(searchKey) → deletes an item with a given search key from the table.
iii. tableRetrieve(searchKey) → retrieves an item with a given search key from a table.

Alternative 3 - Use Stack

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:

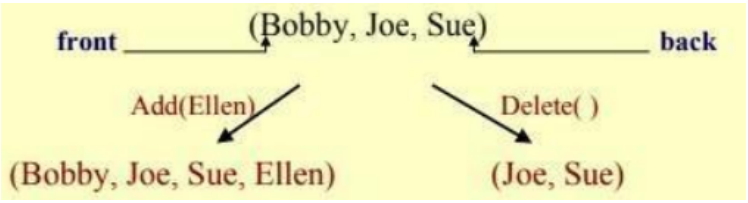
i. top( ) → returns the top item of the stack. It does not change the stack.
ii. pop( ) → changes the stack by removing the top item.

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:

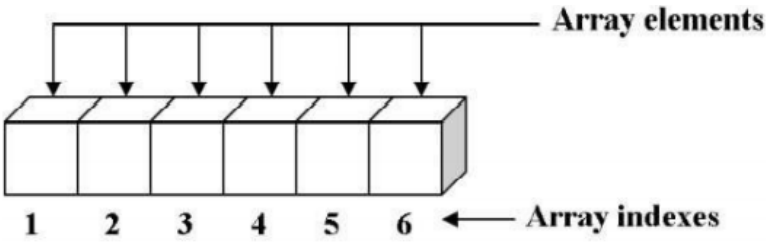
i. createArray( ) → creates an empty array.
ii. add(index, item) → inserts item at position index of an array if 1 ≤ index ≤ size() + 1. If index ≤ size(), items at position index onwards are shifted one position to the right.

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

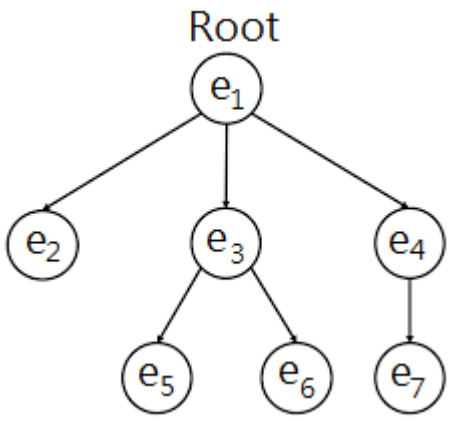
Preach operations to test N-ary Tree:

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, … , v_n }∈ *Graph* e_i = pairs of ordered vertices = (v_i,v_j)| v_i,v_j∈ V E = set of edges = {e_1, e_2, e_3, … , e_n } ∈ *Graph*

W = set of associated weights to each edge = {w_1, w_2, w_3, … , w_n} ∈ *Graph* ∧ w_i ∈ 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
GetEdgeWeigh	Graph x Edge	→ Number	Analyzer
t	Graph x Vertex	→ Graph	Modifier
DeleteVertex	Graph x Vertex1 x Vertex2	→ String	Analyzer
Dijkstra	Graph x Edge	→ Graph	Modifier
DeleteEdge	Graph x Vertex	→ String	Analyzer
ShowGraph			

CreateGraph()

"Creates an empty Graph"

{ pre: *TRUE* }

{ post: Graph = ∅ }

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 ≠ ∅, ∃ 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 ≠ ∅, ∃ Edge, Edge ∈ Graph }

{ post: Edge.weight ∈ W }

AddEdge(Graph, Vertex1, Vertex2, weight)

"Creates a new edge between two given vertices"

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

{ post: String ∈ Text }

DeleteEdge(Graph, Edge)

"Deletes an edge from the Graph"

{ pre: ∃ Graph, Graph ≠ ∅, ∃ 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 }