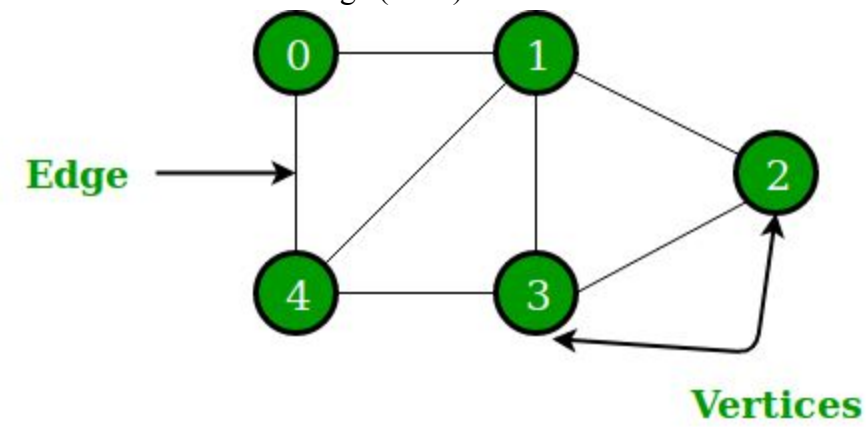
Graph ADT

Graphs are data structures where every node is known as a vertex and the connection between vertices receives the name of edge (aristi).



Recovered from: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

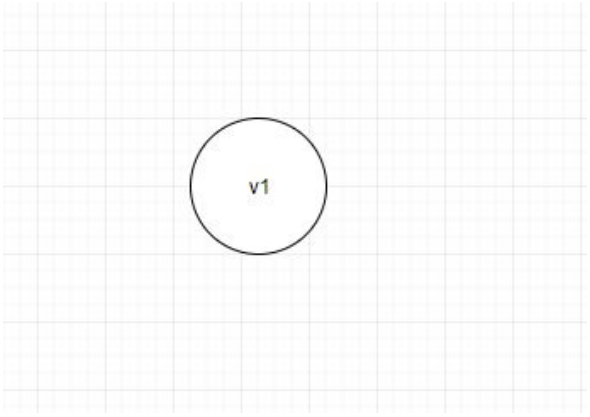
Let G be a Graph:

- $G.vertices \neq \varnothing$
- Let G be a non-directed Graph, e the number of edges and δ the sum of vertices
  - $2e = \sum_{v \varepsilon V} \delta(v)$
- Let G be a directed Graph, δ^- the sum of entering vertices and δ^+ outgoing vertices
  - $\sum_{v \in V} \delta(v)^- = \sum_{v \in V} \delta(v)^+$

Graph	------	-----
Insert vertex:	$G \; x \; v \rightarrow$	$v \in G.vertices$ (modifier)
Insert edge:	$G \; x \; e \rightarrow$	$e \in G.edges$ (modifier)
Delete vertex:	$G \; x \; v \rightarrow$	$v \notin G.vertices$ (modifier)
Delete edge:	$G \; x \; e \rightarrow$	$e \notin G.edges$ (modifier)
Search vertex:	*vertex value*	*vertex* (analyzer)
Minimum distance:	$v \in G$	*void* (analyzer)
Search edge	*edge value*	edge(analyzer)
get neighbors:	$v \in G$	*void* (analyzer)

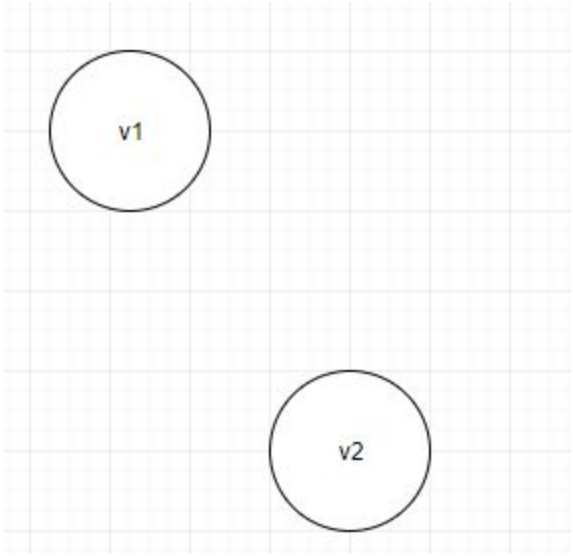
Graph()
“Creates a new graph”
{Pre: null}
{Post: Graph G}

InsertVertex()
“Inserts a new vertex”
{Pre: an empty graph}
{Post:



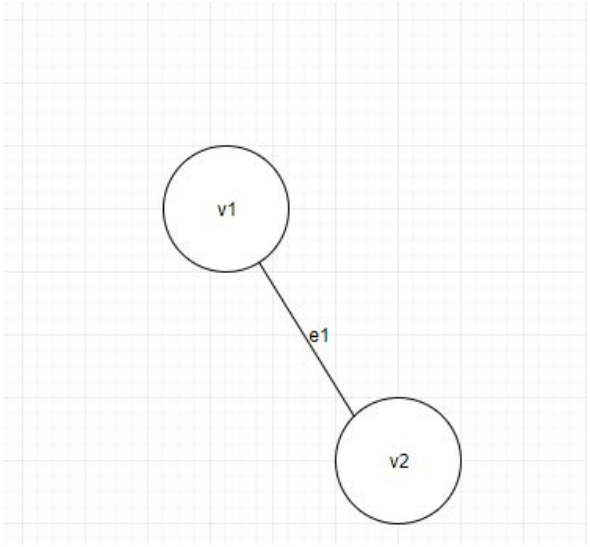
}

InsertEdge()
“Creates a new edge between two vertices”
{Pre:



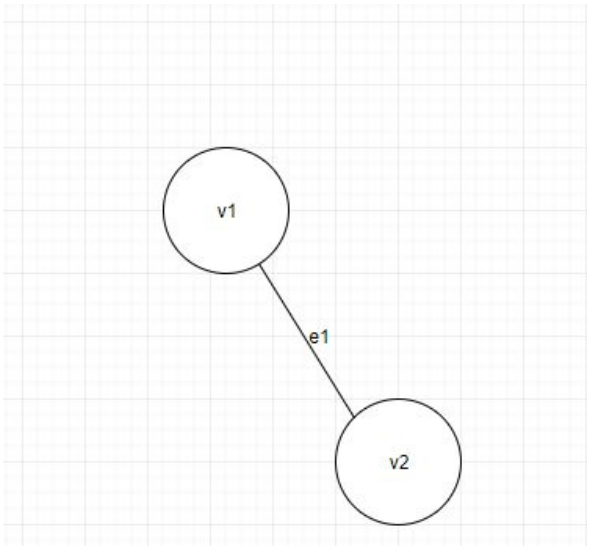
}

{Post:

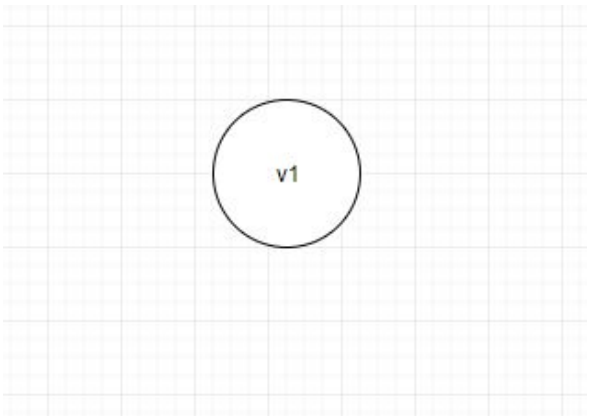


}

DeleteVertex()
"Deletes a vertex from the Graph"
{Pre:

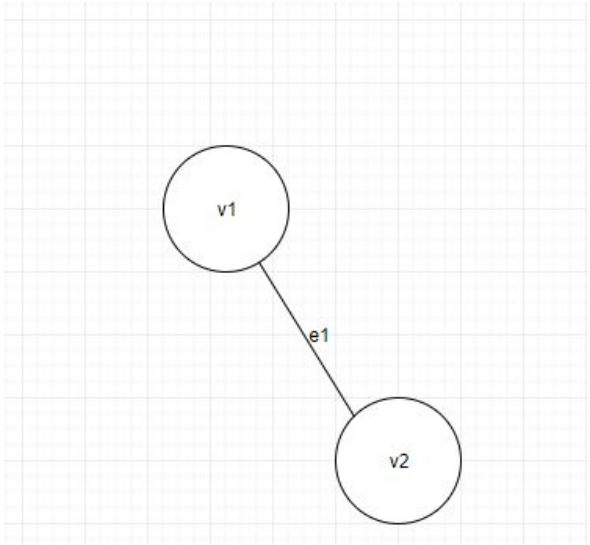


}
{Post:

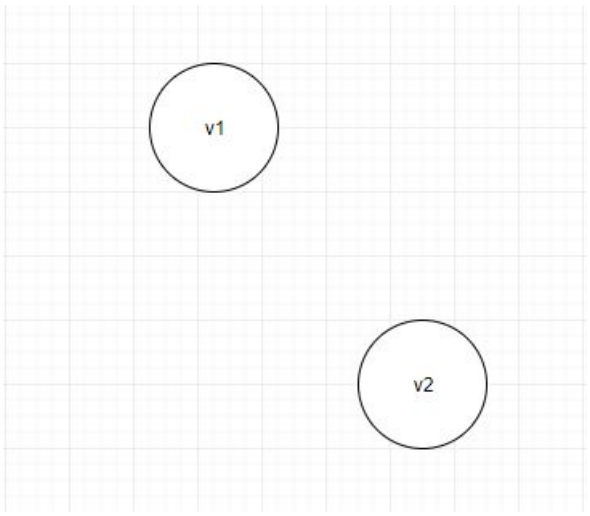


}

DeleteEdge()
“Deletes an edge from the Graph”
{Pre:

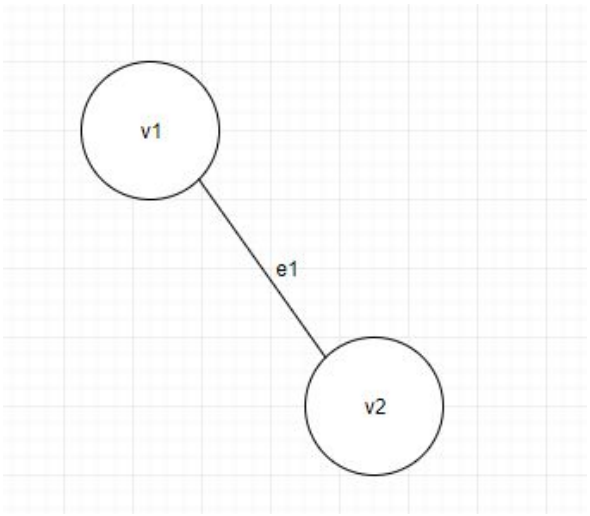


}
{Post:

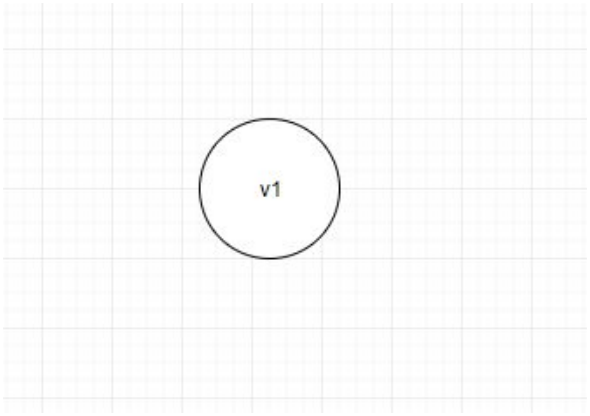


}

SearchVertex()
“Searches for a vertex in the graph using the value given. For example, following the structure, the user searches for v1”
{Pre:

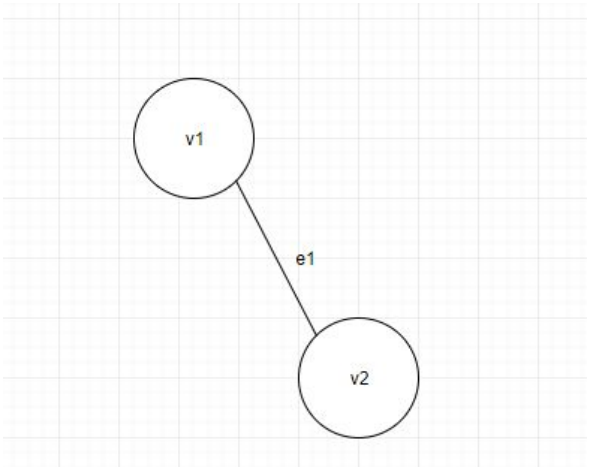


}
{Post: returns the needed vertex:

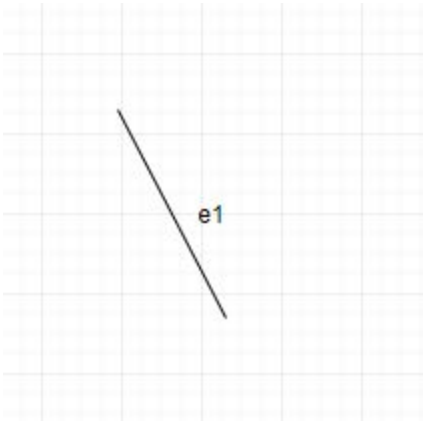


}

SearchEdge()
“Searches for an edge in the graph using the value given. For example, following the structure, the user searches for e1”
{Pre:

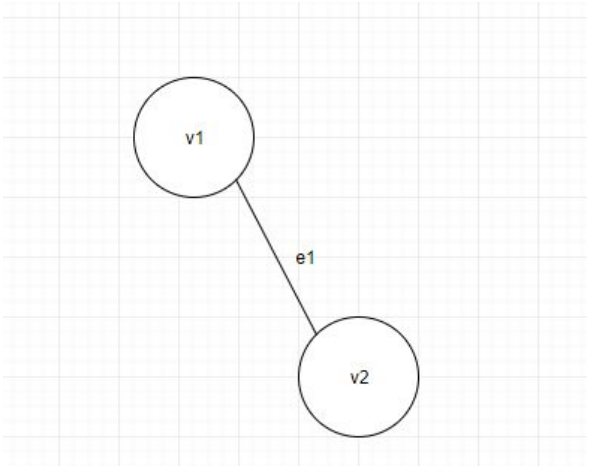


}
{Post: returns the needed edge, this case:

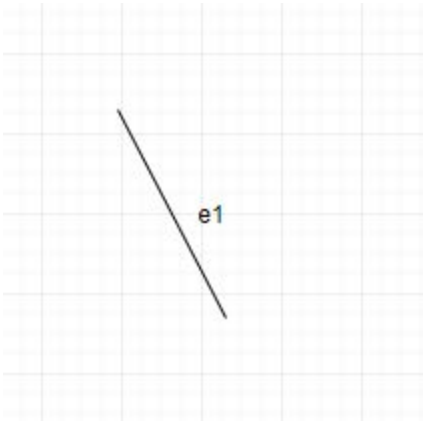


}

GetNeighbors()
“Get the adyacentes of a specific node”
{Pre:



}
{Post: returns the needed edge, this case:



}

Unitary tests design for AdjacencyListGraph:

TestCase	1
Objective	Inserting the first vertex of a graph
Scene	An empty graph
Input	new Vertex()
Output	assertFalse(vertices.isEmpty()) to know if the arrayList containing the vertices is not empty after inserting one

TestCase	2
Objective	testInsertEdge
Scene	A graph with 4 vertices
Input	4 new vertices to insert in the graph connecting: 0-1 1-3 3-4 3-1
Output	assertTure() that will verify that the vertices are indeed connected after inserting the edges

TestCase	3
Objective	testDeleteVertex
Scene	A graph with two vertices and an edge connecting them
Input	
Output	assertTure verifying that the size of vertices is 1 and that the adjaceny list of the remaining vertex is empty

TestCase	4
Objective	testIsAdjacent
Scene	A graph with vertices and edges connecting them
Input	
Output	assertTrue verifying that the method isAdjacent returns true for two of the adjacent vertices

TestCase	5
Objective	testDeleteEdge
Scene	A graph with vertices and edges connecting them
Input	The edge to delete
Output	assertFalse verifying that the method isAdjacent returns false for the vertices that were connected through the deleted edge

TestCase	6
Objective	testBFS
Scene	A graph with vertices and edges connecting them
Input	
Output	assertEquals for an arrayList containing the verified vertices in the right order and the return of the BFS method

TestCase	7
Objective	testKruskal
Scene	A graph with vertices and edges connecting them
Input	
Output	assertEquals() with the number of edges the kruskal return graph should have and the kruskal return graph number of edges

TestCase	8
Objective	testDFS
Scene	A graph with vertices and edges connecting them
Input	
Output	assertEquals for an arrayList containing the verified vertices in the right order and the return of the BFS method

TestCase	9
Objective	testFloyd
Scene	A graph with vertices and edges connecting them
Input	
Output	We verify if the return from the floydwarshall method is equal to an array with the right values and order for the given graph after using floydwarshall

TestCase	10
Objective	testDijkstra
Scene	A graph with vertices and edges connecting them
Input	
Output	We verify if the return from the floydwarshall method is equal to a matrix with the right values and order for the given graph after using floydwarshall

TestCase	11
Objective	testPrim
Scene	A graph with vertices and edges connecting them
Input	
Output	We verify if the return from the floydwarshall method is equal to an array with the right values and order for the given graph after using floydwarshall

Unitary tests design for AdjacencyMatrixGraph:

TestCase	1
Objective	InsertVertex
Scene	A empty graph
Input	New Vertex()
Output	We verify that the list (vertexOrder) of vertices is not empty

TestCase	2
Objective	InsertEdge

Scene	A graph with vertex already added
Input	new Edge(), new Edge(), new Edge(), new Edge()
Output	We verify that on the positions that were added the edges the queue is not empty

TestCase	3
Objective	DeleteVertex
Scene	A graph with vertices and an edge added
Input	The position of the vertex that we want to delete
Output	We verify that the size of vertexOrder is modificated and we check if the vertex that still on the list is the one that we did not erased

TestCase	4
Objective	DeleteEdge
Scene	A graph with vertices and an edges added
Input	The vertex from and to and the id of the edge
Output	We verify that the size of EdgeOrder is modificated

TestCase	5
Objective	testKruskal
Scene	A graph with vertices and an edges added
Input	
Output	We verify that the number of edges on the new graph is correct

TestCase	6
Objective	testDFS
Scene	A graph with vertices and an edge added
Input	
Output	We verify that the Arraylist of the method is the same that other arraylist with the expected answer

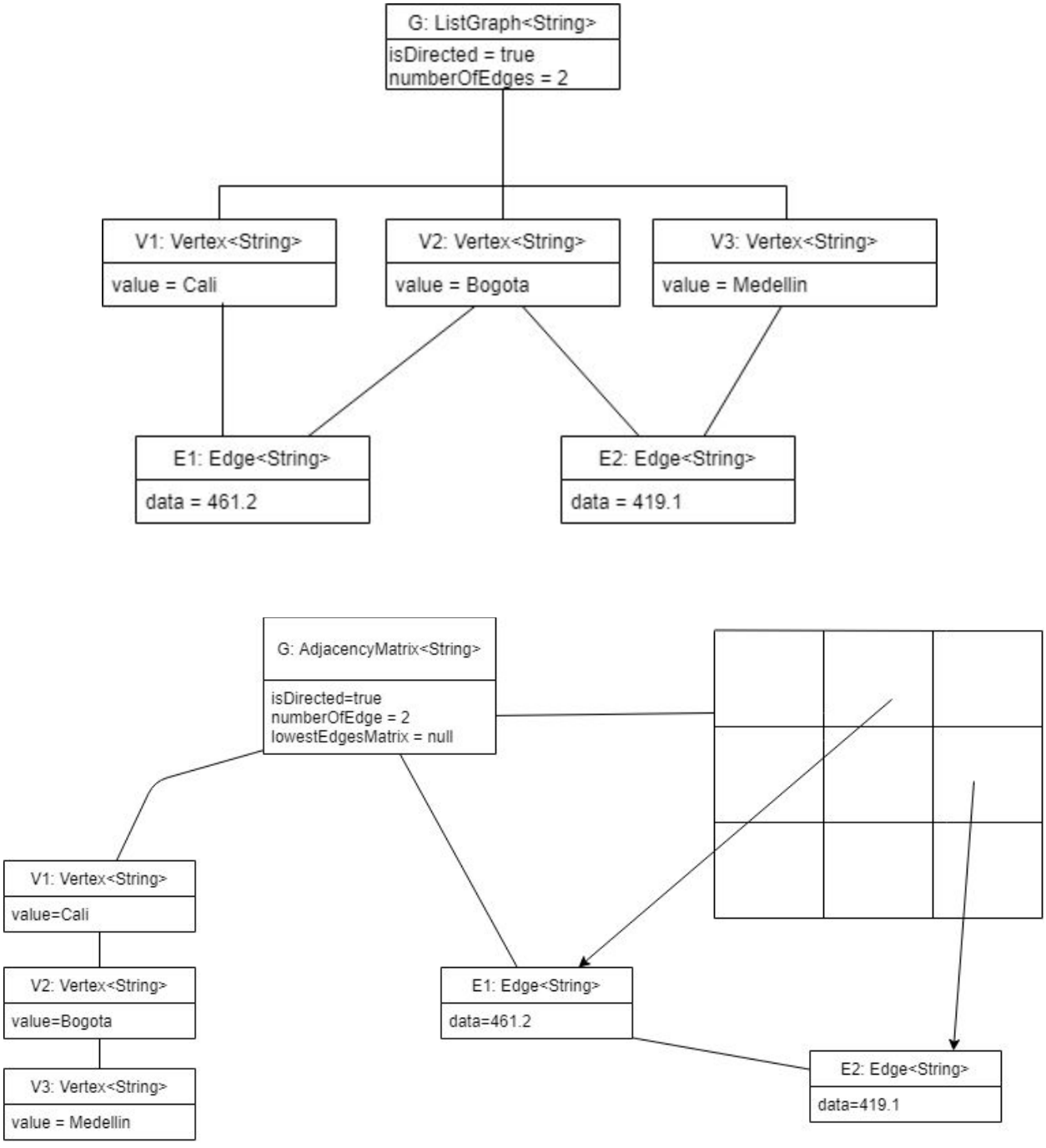
TestCase	7
Objective	testPrim
Scene	A graph with vertices and an edge added
Input	
Output	We verify that the array returned by the method is the same comparing with another with the expected solution

TestCase	8
Objective	testDijkstra
Scene	A graph with vertices and an edge added
Input	
Output	We verify that the kilometers on the matrix are correct

TestCase	9
Objective	testFloyd
Scene	A graph with vertices and an edge added
Input	
Output	We verify that the matrix returned by the method is correct

Class diagram

Object diagram



Engineering method report

Problem to solve: the problem consists of a set of Colombian cities connected by flights. The flights have certain costs and connect one city with another one. All the cities are reachable from

any other city. The task is to find the cheapest way to get from one city to another one but spending the least amount of money as a customer. Also, the airline company providing the flights would like to know if there is a better way to connect the cities by removing some flights and creating a minimum number of flights connecting only the necessary cities that will allow the customers to fly anywhere. The company wants to be able to add a new flight to a new city from any of the already existing cities. They also want to be able to remove a flight connecting two cities.

Step1. problem identification:

- Functional Requirements:
    - R1: The software must allow the user to add a new city to the map by clicking on it and typing the name
    - R2: The software must allow the user to create a flight between two cities by typing in their names and the cost of the flight
    - R3: The software can show the lowest-cost route to visit all the cities connected by flights on the map
    - R4: The software has to allow the user to find the lowest-cost route between two specific cities.
- It's needed a program that can determinate de minimum price and kilometers of a travel between cities on a airline
- The solution must secure that the specified travel is the cheapest or shortest possible
- The solution must be given with two different implementations of graphs
- The solution given with the two different graph must be the same

Step2. Information:

- Graph: is a non-linear data structure consisting of nodes and edges. The nodes are, normally known as vertex are connected to others vertex by edges, there are so many different types of graph but the main idea is the same.
- Airline: a company dedicated to the transport of passengers
- Minimum: that have the less value or quantity

Step3. creative solutions:

<u>Alternative 1: incidence list</u>

Is a form to implement a graph that use a list to identify the adjacents of the edge on the graph. Every single edge have an space on the list and in each space are the vertices related

Alternative 2: adjacency list

Is a form to implement a graph that use a list to identify the adjacents of a node (vertex) on the graph. Every single node have an space on the list and in each vertex space are the vertices related

Alternative 3: incidence matrix

Is another form to implement a graph, this one use a matrix with rows as vertices and columns as the edges and if exist and adjacency on the intersection of the vertex and the edge we put a value to identify the relation

Alternative 4: adjacency matrix

Is a from to implement the graph that use a square matrix with rows and columns as the vertices, and if exist a relation between two vertex in the intersection on the matrix we put the edge

Step4. transition from ideas to design:

Alternative 1: incidence list

- If there are too many edges the list become so long.
- Each edge must be referenced and named to used it on the list.
- Each position has the vertices that are tied by the edge so it is confuser.
- Cannot use floyd warshall by itself, it will need to be transfer to a matrix.

Alternative 2: adjacency list

- Cannot use floyd warshall by itself, it will need to be transfer to a matrix.
- We know with ease what vertex we can visit.
- the edges are not specified on the main structure (the adjacency list).
- Main operations of the matrix are simple.

Alternative 3: incidence matrix

- If there are too many edges the matrix become so long in columns.
- Each edge must be referenced as a column in the matrix.
- Each Vertex must be referenced as a row in the matrix.

- If there are too many edges it spend a lot of space because not every intersection exist, normally a graph have more adjacencies than vertex.
- Can use floyd warshall easily.

Alternative 4: adjacency matrix

- Each Vertex must be referenced as a row in the matrix, so the order should be allusion in another structure.
- The matrix is square, so the operations would be easy to follow.
- The intersection of the adjacency matrix would have the edges organized so it will easy to get the edge information needed.
- Following the idea that normally there are more edges than vertices, the matrix its fuller of information without using more space.
- It is perfect to use floyd warshall.

Step5. Selection of the best solution:

Criteria

- Criterion A. Implementation complexity of the graph.
  - [3] Easy
  - [2] Medium
  - [1] Hard
- Criterion B. Space complexity.
  - [2] Low
  - [1] High
- Criterion C. Time complexity.
  - [3] Low
  - [2] Half
  - [1] High
- Criterion D. Implementation of the methods.
  - [2] Easy
  - [1] Hard

Evaluation

	Criterion A	Criterion B	Criterion C	Criterion D	Total
Alternative 1	2	2	2	1	7
Alternative 2	3	2	2	1	8
Alternative 3	2	1	2	2	7
Alternative 4	3	1	3	2	9

Selecting the best alternative

We concluded that the best options to model our problem were to use an adjacency list and matrix because they scored the highest points in the evaluation. They provide decent time complexity and space complexity, creating a balance between these. On top of that, they are the most frequently used implementations of graphs.

Step6. Preparation of report and specifications:

Considerations

1. The program starts with specific vertices, but you can add more.
2. The implementations will be based on the problem of the kilometers and price.