Algorithms and data structures & VIP

2019-2

Christian Flor - A00355624, Cesar Canales -A00345026, Carlos Restrepo - A00355028

Networks

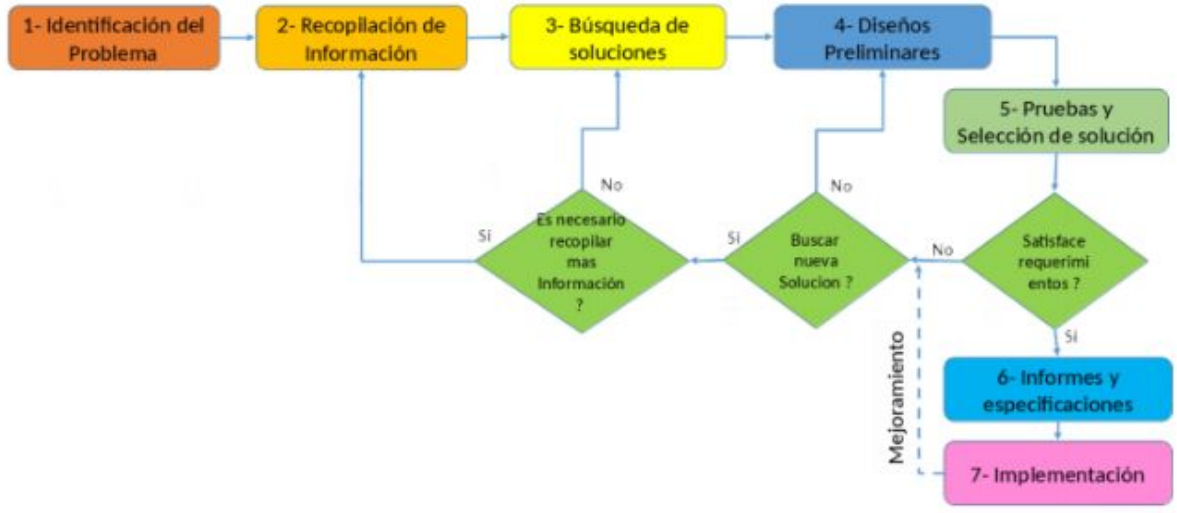
Context of the Problem.

Networks are used to represent and analyze a lot of complex systems, the analysis of these include figuring out optimal pathing, max flow, least cost, etc. Despite the importance of networks, there is not a trivial way of teaching them, this project aims to create an educational software which allows students to visualize networks and treat problems like the ones mentioned in order for them to have an easier learning experience.

Solution development.

To solve the previous situation, the Engineering Method was chosen to develop the solution, following a systematic approach in accordance with the problematic situation.

Based on the description of the Engineering Method of the book "Introduction to Engineering" by Paul Wright, it presents the following flow chart, whose steps we will follow in the development of the solution.



Step 1. Problem Identification

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

Identification of needs and symptoms

- ➢ Currently, there is a limited amount of resources from where to learn networks.
- ➢ A functionality that allows to visualize the network.
- ➢ A functionality that allows the user to add a new network and choose the type of problem to be solved.

Definition of the problem

It is required to develop a tool to visualize networks in order to make teaching easier. Also, a functionality which solves some classic problems involving networks and shows the step by step of the solution.

Step 2. Recompilation of information

To have total clarity in the concepts involved, a search is made of the different problems that are to be treated and their solutions. In addition to that, some general information about graph theory which is a natural way of representing networks in computer science.

Definitions

Graph

In graph theory, a graph is a structure amounting to a set of objects in which some pairs of the objects are "related". The objects are often referred to as vertices(also called nodes or points) and each of the related pairs of vertices is called an edge. These edges may be directed or undirected which means that they can have a direction, vertex A points to vertex B, or not, vertex A points to vertex B and vertex B points to vertex A. Finally, these edges can have an associated weight, which can be thought of as the "cost" that has to be paid in order to reach vertex B from vertex A.

Source:
https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

Shortest path problem

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized. For our purposes, we're going to be treating the single source shortest path problem, in which we have to find the shortest paths from a source vertex v to all other vertices in the graph.

Source:
https://en.wikipedia.org/wiki/Shortest_path_problem

Minimum spanning tree

A minimum spanning tree (MST) or minimum weight spanning tree is 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. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.

Source:
https://en.wikipedia.org/wiki/Minimum_spanning_tree

Maximum flow problem

Maximum flow problems involve finding a feasible flow through a flow network that is maximum. Given a source s and a sink t the maximum flow problem is to maximize the flow, that is, to route as much flow as possible from s to t. The flow network can be represented as a directed graph in which each edge has a maximum capacity and the goal is to reach t from s with a maximum flow value.

Source:
https://en.wikipedia.org/wiki/Maximum_flow_problem

Step 3. Looking for creative solutions

In order to find the most efficient solution for the single source shortest paths problem, various ideas are considered. Considering that the result of the algorithm is to be shown in real time, we require one which is efficient. The solutions are given below:

Alternative 1.
Generate every single path from the source vertex to each of the other vertices and for each one, consider the one with the minimum total sum of weights as the answer.

Alternative 2.
Use Floyd Warshall's algorithm in order to find every single source shortest path, whenever a query is made for a source vertex, just output the answer given the precomputation done beforehand with Floyd Warshall's.

Alternative 3.
Use dijkstra's algorithm for single source shortest paths, which uses a priority queue that uses the distance between the vertices as it's sorting criteria. It computes the shortest paths greedily, by taking the closest vertex and comparing it with each one of it's adjacent vertices, updating the distance if necessary.

★ Minimum spanning tree
Alternative 1.
Generate every possible subgraph which has n-1 edges where n is the number of vertices in the graph, once we have a subgraph we check if it is connected which guarantees that it doesn't have any cycles, if this condition is met, then we compute the sum of weights of the edges and check if it's minimum. We repeat this process for all combinations of n-1 edges and this way we get the minimum spanning tree.

Alternative 2.
Use kruskal's algorithm which allows us to find the minimum spanning tree in O(nlogn) time complexity, it also allows to find the minimum spanning forest is the graph happens to not be connected.

★ Maximum flow problem
Before we can tackle this problem, we shall give some basic definitions. First off, the **residual capacity** of a directed edge is the capacity minus the flow. It should be noted that if there is a flow along some directed edge
(u,v), then the reversed edge has capacity 0 and we can define the flow of it as $f((v,u))=-f((u,v))$. This also defines the residual capacity for all reversed edges. From all these edges we can create a **residual network**, which is just a network with the same vertices

and edges, but we use the residual capacities as capacities. Finally, an augmenting path is simple path in the residual graph, i.e. along the edges whose residual capacity is positive.

Alternative 1.
Generate every possible augmenting path, from the source to the sink, then test all possible combinations of augmenting paths, calculating the respective flow in the process, the one where the flow is maximum is the answer.

For the following alternatives, we consider using the Ford-Fulkerson method to solve this problem, this method consists of choosing an augmenting path at random and adding the maximum flow possible through this path, and repeating the process until the maximum flow is reached. Some of the most common algorithms which exploit this method are presented below.

Alternative 2.
Use Edmonds-Karp algorithm in order to solve this problem, this algorithm is just an implementation of the Ford-Fulkerson method which uses a BFS to find augmenting paths.

Alternative 3.
Use Dinic's algorithm in order to solve this problem, this algorithm , as well as the last one, is an implementation of the Ford-Fulkerson method, it uses a BFS and a DFS to the maximum flow in a flow network.

Step 4. Transition from ideas formulation to preliminary designs

In this step we discard unfeasible options.

It's easy to see that some of the presented ideas are brute force approaches so they take a really long time to execute, that is, their time complexity is exponential, making them not viable for the project at hand. In that order of ideas, the following alternatives are discarded:

- Alternative 1 in the single source shortest paths problem.
- Alternative 1 in the minimum spanning tree problem.
- Alternative 1 in the maximum flow problem.

This said, we consider the remaining alternatives:

For the single source shortest paths problem:

Floyd Warshall

- ❖ It allows the graph to have negative cycles, detecting them in the process..

- ❖ In addition to calculating every single source shortest path, it also calculates every path in the process, that is, the matrix of predecessors for every vertex in a path.
- ❖ The time complexity or running time of the algorithm is O(n^3) where n is the number of vertices in the graph.

Disjktra's algorithm

- ❖ It doesn't support negative weights in edges.
- ❖ Calculates the shortest path from a given source vertex to every other vertex in the graph, calculating the predecessors in the process.
- ❖ The time complexity or running time of the algorithm is O(n log n) where n is the number of vertices in the graph.

For the minimum spanning tree problem we have:

Kruskal's algorithm

- ❖ Works for not disconnected graphs, finding the minimum spanning forest in this case.
- ❖ The time complexity or running time of the algorithm is O($n \log n$) where n is the number of vertices in the graph.

Prim's algorithm

- ❖ It only works in connected graphs.
- ❖ Calculates the minimum spanning tree from a given source vertex.
- ❖ The time complexity or running time of the algorithm is O(n log n) where n is the number of vertices in the graph.

Finally, for the maximum flow problem:

Edmonds-Karp algorithm

- ❖ Uses a BFS to find augmenting paths.
- ❖ Has a certain corner case which makes this algorithm not find the maximum flow everytime, it's very rare.
- ❖ The time complexity or running time of the algorithm is O(VE^2) where V is the number of vertices and E is the number of edges in the graph.

Dinics 's algorithm

- ❖ Uses a BFS to check whether or not the sink can be reached from the source vertex.
- ❖ Uses a DFS to find augmenting paths.
- ❖ The time complexity or running time of the algorithm is O(n log n) where n is the number of vertices in the graph.

Step 5. Evaluation and Selection

★(A) Single source shortest paths

For this problem, the wisest alternative to choose is the second one because it's more efficient and it supports the goal of visualizing the algorithm as it is doing it.

Alternative 1.

- Given that the point of the software is to be able to visualize Dijkstra's algorithm, doing Floyd Warshall would make the visualization more difficult.
- Having precomputed distances means that the algorithm has to be executed one time at the start, making it very handy.
- Given the time complexity a graph with an amount of vertices greater than 1000 can take a long time to complete.

Alternative 2.

- By doing Dijkstra we can take into account every step the algorithm takes, in order to simulate it in the application.
- Doesn't allow negative edges.
- Given the time complexity of the algorithm, we can apply it plenty of times and it'll still be efficient.

(B) Minimum spanning tree

We select alternative 1 because it works in graphs that are disconnected as well as graphs that are connected, given that there is no guarantee that the graph given in the input is connected.

Alternative 1.

- It has the advantage of working when the graph is disconnected, finding the minimum spanning forest.
- The advantage mentioned above allows to calculate the minimum cost for each independent network.
- It's time complexity allows for it to be run a lot of times with the execution time still being low.

Alternative 2.

- This alternative requires the graph to be connected in order to not fail.
- An alternative is to run this algorithm in each connected component in order to find the minimum spanning forest.
- This last alternative would take O(k n logn) where k is the number of connected components and n is the number of vertices in the graph.

★(C) Maximum flow problem

For the maximum flow problem we chose alternative 2 because it is more correct at the time of calculating the max flow, as mentioned before Edmonds-Karp algorithm has some corner cases which don't work.

Alternative 1.

- In this alternative it will be easy to add new bars below, from the first one since the user is only asked the type of block and the quantity, the new bar will be created following the same idea selected from the features bar mentioned above

- Improve the user experience by creating more bars.

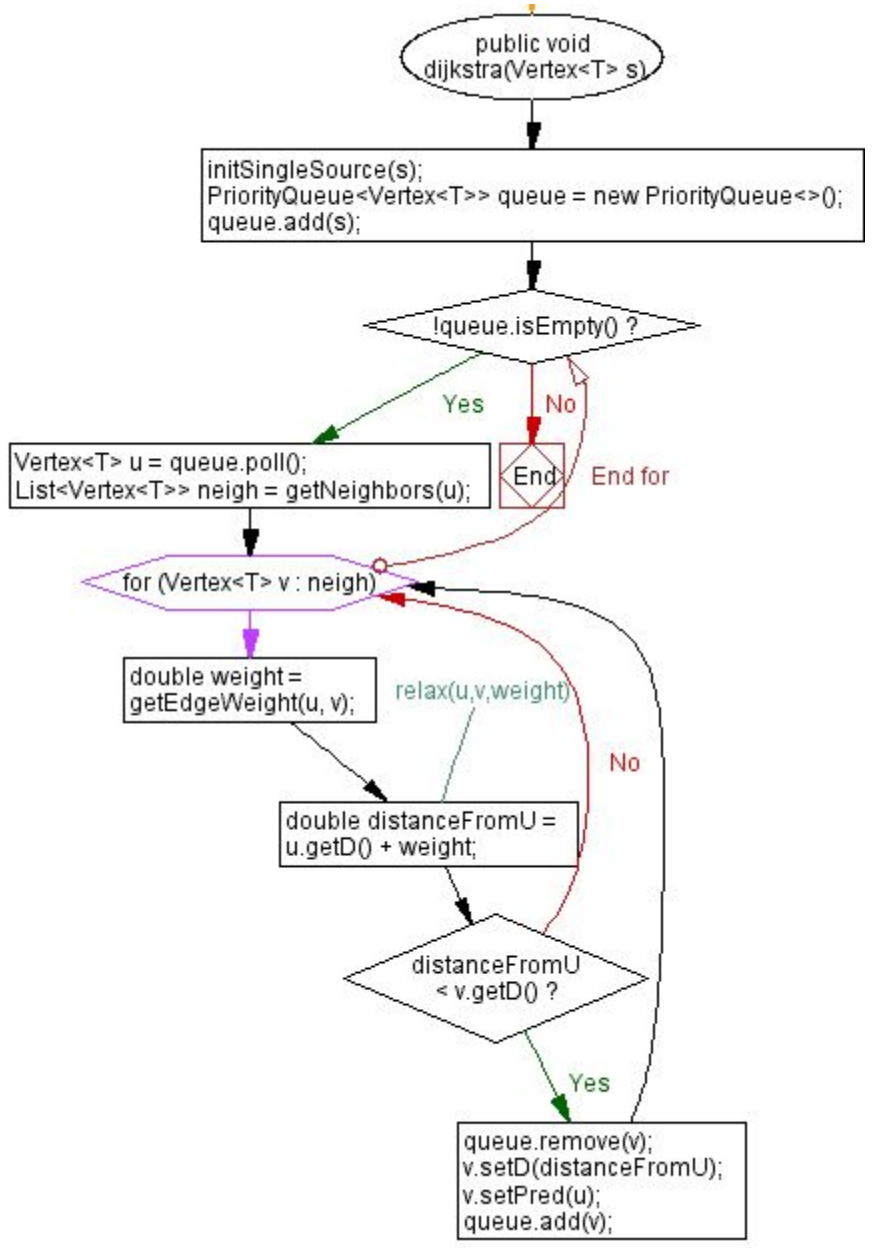
Alternative 2.

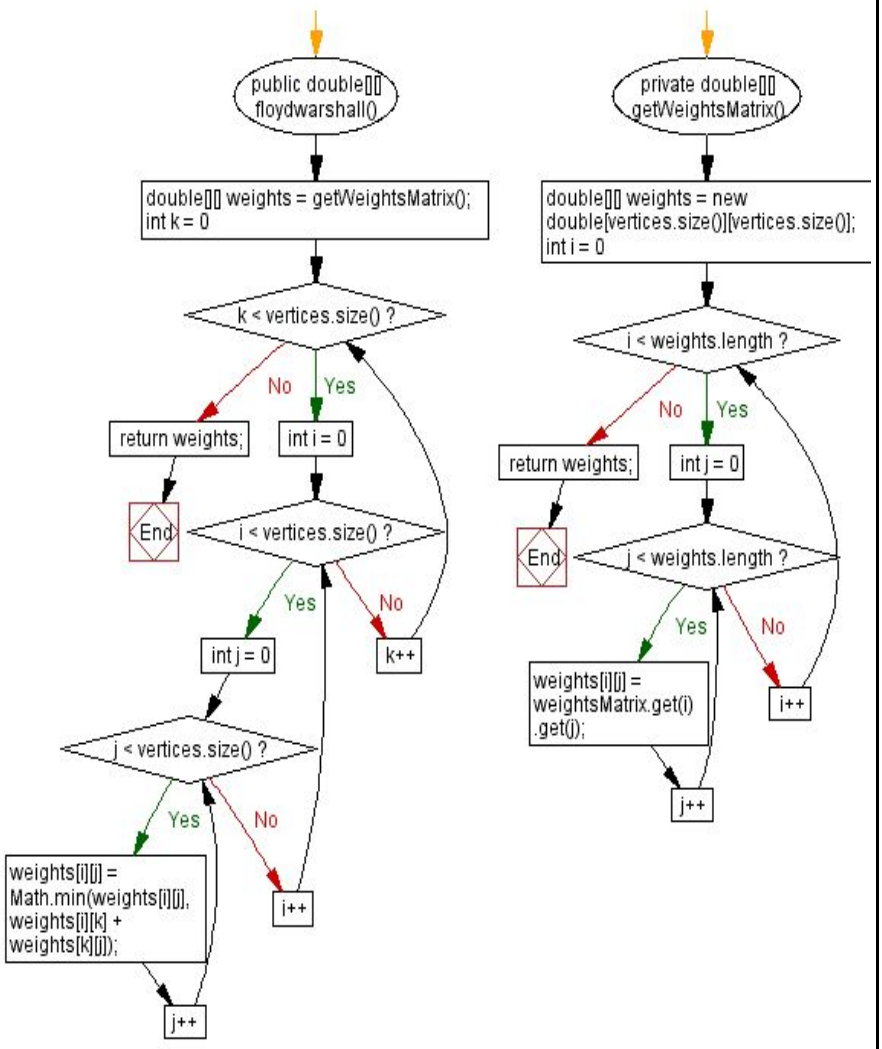
- The user is given more freedom to decide how many quick access bars he wants but at the same time he is restricted since this can only be done once.
- The user can type a large number of quick access bars.

Step 6. Preparation of Reports and Specifications

1. **ADT Design**
2. **Class Diagram**
3. **Test case Design**

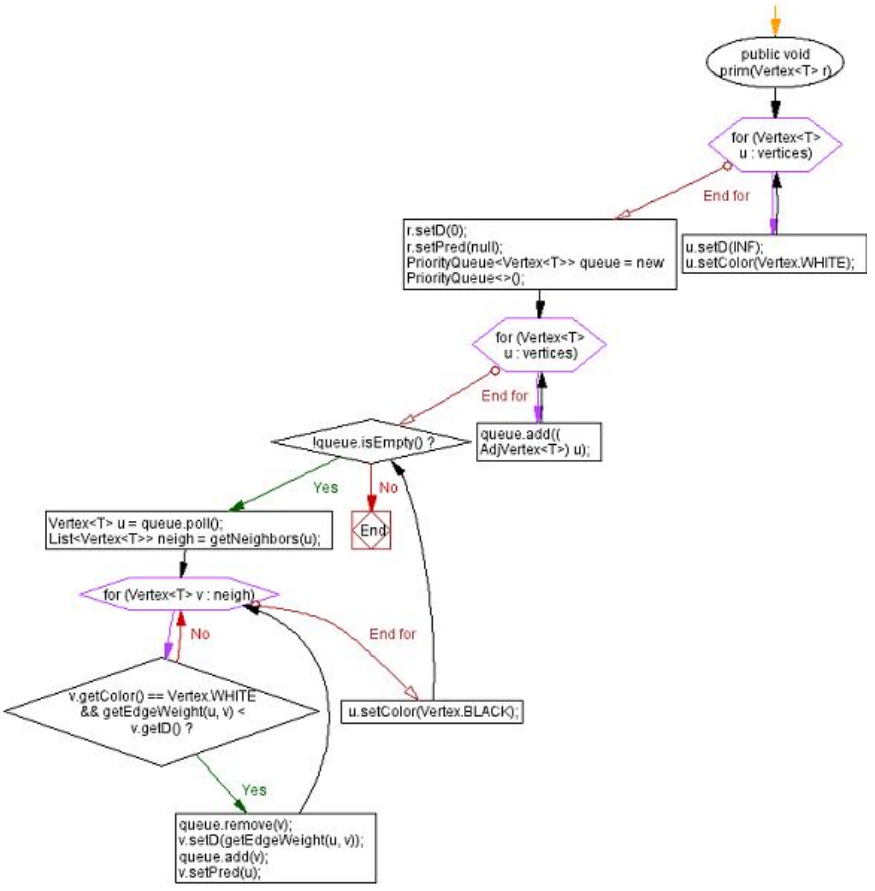
4. Pseudocode of the most relevant algorithms

Algorithm flow chart	Pseudocódigo
	**Dijkstra** 1 **function** Dijkstra(*Graph*, *source*): 2 dist[*source*] ← 0 3 4 create vertex set Q 5 6 **for each** vertex *v* in *Graph*: 7 **if** *v* ≠ *source* 8 dist[*v*] ← INFINITY 9 prev[*v*] ← UNDEFINED 10 11 Q.add_with_priority(*v*, dist[*v*]) 12 13 14 **while** Q is not empty: 15 *u* ← Q.extract_min() 16 **for each** neighbor *v* of *u*: 17 *alt* ← dist[*u*] + length(*u*, *v*) 18 **if** *alt* < dist[*v*] 19 dist[*v*] ← *alt* 20 prev[*v*] ← *u* 21 Q.decrease_priority(*v*, *alt*) 22 23 **return** dist, prev



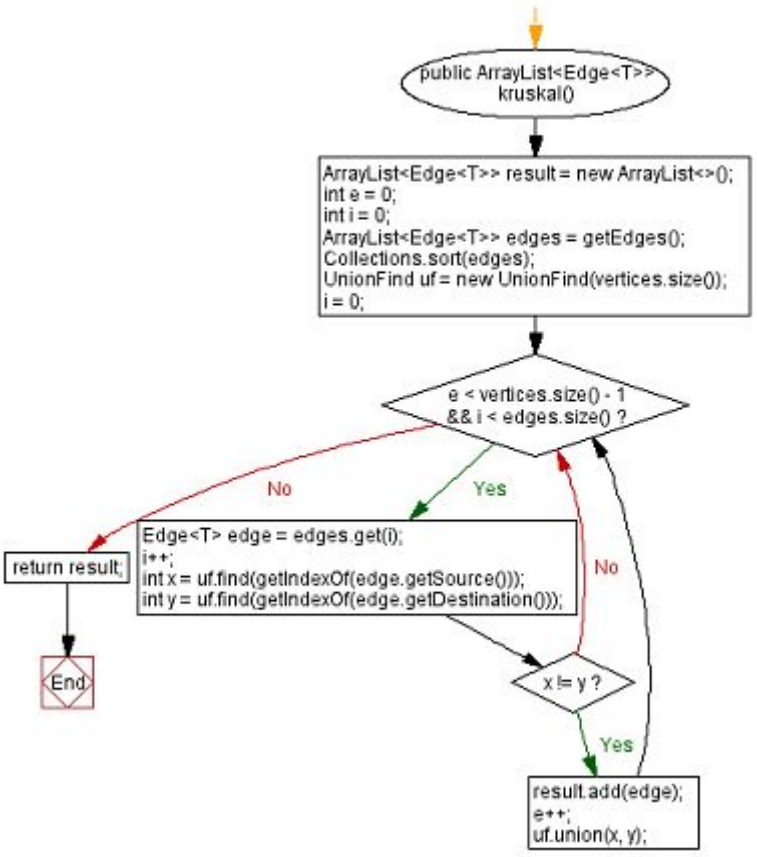
Floyd Warshall

```
1  let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinit
2  for each vertex v
3      dist[v][v] ← 0
4  for each edge (u,v)
5      dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
6  for k from 1 to |V|
7      for i from 1 to |V|
8          for j from 1 to |V|
9              if dist[i][j] > dist[i][k] + dist[k][j]
10                 dist[i][j] ← dist[i][k] + dist[k][j]
11 end if
```



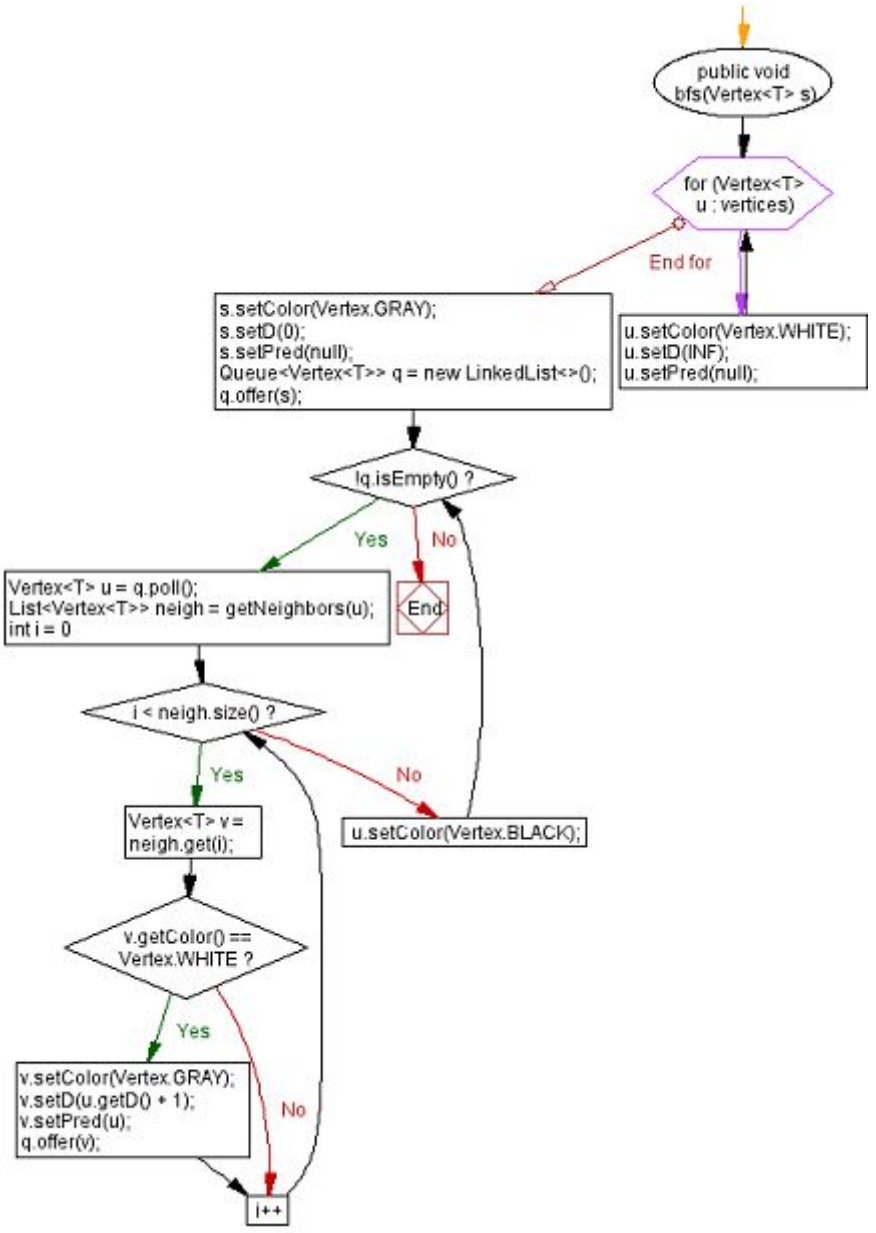
Prim

```
Prim(G,w,r) {
  for each u ∈ V {
    key[u] := +∞;
    color[u] := W;
  }
  key[r] := 0;
  pred[r] := NIL;
  Q = new PriorityQueue(V);
  while (Q is nonempty) {
    u = Q.extractMin();
    for each (v ∈ adj[u]) {
      if (color[v] = W ∧ w[u,v] < key[v]) {
        key[v] := w[u,v];
        Q.decreaseKey(v,key[v]);
        pred[v] := u;
      }
    }
    color[u] := B;
  }
}
```



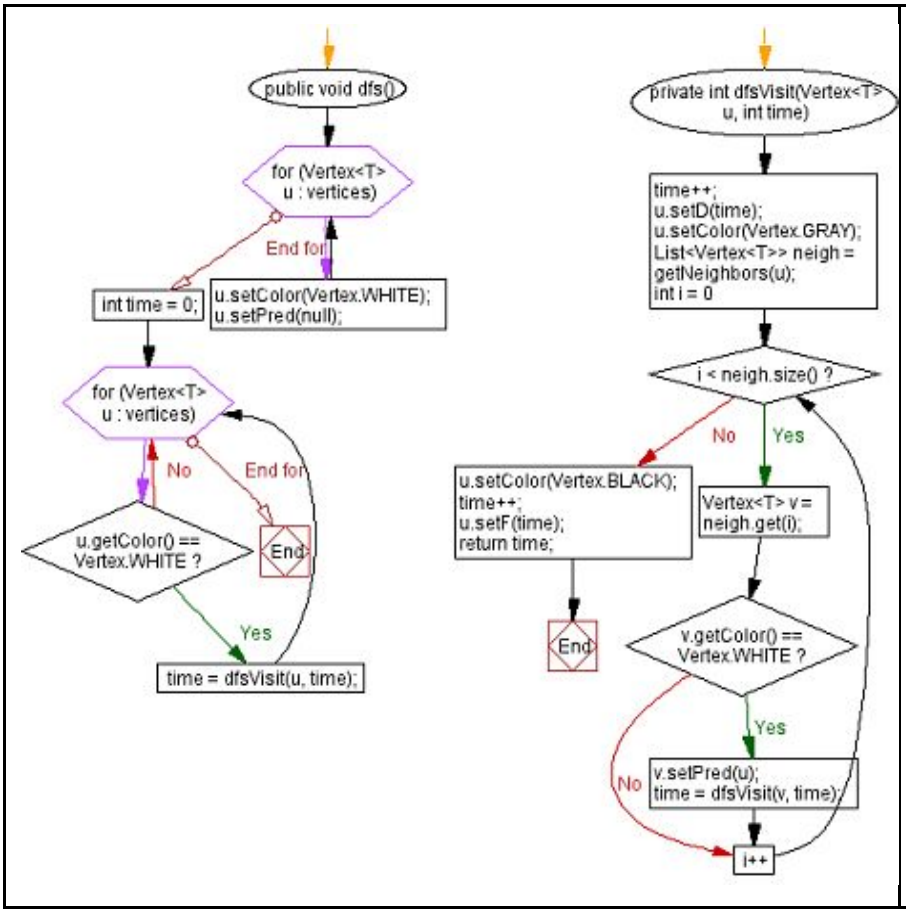
Kruskal

```
1 A = ∅
2 foreach v ∈ G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u) ≠ FIND-SET(v):
6         A = A ∪ {(u, v)}
7         UNION(u, v)
8 return A
```



BFS

```
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```



DFS

DFS(G)

```
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

Step 7. Design implementation

The solution to the problem is in the following repository:
https://github.com/Carlosches/alienbackend/tree/test/src/main

Specification of Functional Requirements. (in terms of input / output)

Name	**F.R# 1. Allow the user to construct a graph**
Summary	The user will have the ability to build a graph with the following properties: • Directed or undirected. • Number of vertices in the graph. • Connections between the vertices with their respective weights.
Input	
Number of vertices of the graph, connections between vertices and whether the graph is directed or undirected.	
Output	
Graph visualized on screen.	

Name	**F.R# 2. Allow a clear visualization of the graph**
Summary	The software will allow to visualize a graph constructed by the user, with values of his choice and interact with it by way of moving it with the mouse.
Input	
None	
Output	
Interactive graph on screen.	

Name	**F.R# 3. Allow a clear visualization of Dijkstra's algorithm**
Summary	The system will allow the user to run Dijkstra's algorithm in the build graph and visualize each step as it happens.
Input	
Graph built by the user.	
Resultados	
Dijkstra visualization on screen.	

Name	**F.R# 4. Allow the user to visualize the shortest path from a source vertex to another vertex.**
Summary	The system will allow the user to visualize the path from a source vertex to a destination vertex chosen by the user as well as the total distance of the path.
Input	
Source and destination vertices.	
Output	
Path and distance visualization on screen.	

Name	**F.R# 5. Allow the user to visualize the construction of a minimum spanning tree.**
Summary	The system will allow the user to visualize the construction of a minimum spanning tree given the initial graph built by the same user.
Input	
Graph built by the user.	
Output	
Minimum spanning tree on screen, or forest if the case is met.	

Name	**F.R# 6. Allow the user to visualize the total sum of the weights of the edges that belong to the minimum spanning tree.**
Summary	The system will allow the user to visualize the sum of the weights of the edges that resulted from the construction of the minimum spanning tree.
Input	
Minimum spanning tree construction.	
Output	
Total sum of the weights of the edges on screen.	

Name	**F.R# 7. Allow the user to visualize the maximum flow from a source to a sink.**
Summary	The system will allow the user to visualize the maximum possible flow in a flow network or graph built by the user.
Input	
Graph built by the user.	
Output	
Max flow shown on screen.	

Name	**F.R# 8. Allow the user to visualize the finding of the max flow in the flow network.**
Summary	The system will allow the user to visualize the finding of augmenting paths with Dinic's algorithm and the come around on the max flow value.
Input	
Graph built by the user.	
Output	
Dinic's algorithm visualization on screen.	

Specification of Non Functional Requirements

Name	**N.F.R# 1. The functionalities must be shown in the icesi vip project web page**
Summary	Given that the functionalities must be shown online, the visualization of graphs must be done in javascript with the assistance of html and css for the visual aspect.