Engineering method

# **\*\*Problem definition:\*\***

We have a set of cities connected by roads, and each city has a specific price for fuel. We need to find the most economical way to travel from one city to another (start to destination), considering the price of fuel in each city and the fuel tank capacity of the vehicle. Also, we need to find another route that is also economical considering the price of fuel in each city.

# **\*\*Gathering of necessary information:\***

To solve this problem, we need the following data:

* Number of cities.
* Number of roads.
* The price of fuel in each city.
* The roads connecting the cities and the distance from each road.
* The fuel tank capacity of the vehicle.
* Start and destination city.

# **\*\*Functional requirements for solving the problem:\*\***

* The programme must be able to handle up to 1000 cities and 10000 roads.
* The program must be able to perform up to 100 queries.
* The program must be able to handle different fuel prices for each city.
* The program must be able to handle different fuel capacities for each query.
* The program must be able to handle different start and end cities for each query.

# **\*\*Search for creative solutions:\*\***

The six possible solutions:

## 1. **\*\*Dijkstra Algorithm:\*\***

We can use Dijkstra's algorithm to find the cheapest route between two cities. This algorithm works best when a single query is needed.

## 2. **\*\*Floyd-Warshall’s Algorithm:\*\***

This algorithm can pre-calculate the cheapest routes between all city pairs, which can be useful if multiple queries are needed.

## 3. **\*\*Bellman-Ford’s Algorithm:\*\***

This is another algorithm we can use to find the shortest route between two cities. It has a worse time complexity than Dijkstra and Floyd-Warshall, but it can handle edges with negative weights.

## **4. \*\* The breadth-first search (BFS) algorithm:\*\***

We can modify the BFS algorithm to account for fuel costs and tank capacity. This solution can work well if the network is shallow and densely connected.

## **5. \*\*Depth-first search (DFS ) algorithm :\*\***

Similar to BFS, we can modify the DFS algorithm to solve this problem. This solution can work well if the network is deep and sparsely connected.

## **6. \*\*Priority Queue with Dijkstra:\*\***

Instead of using an array to store the path costs in Dijkstra's algorithm, we can use a priority queue to store the nodes, which can improve the efficiency of the algorithm if the network is sparse.

## **7.\*\*Transition from brainstorming to preliminary designs:\*\***

From these ideas, we can formulate preliminary designs for each. For each algorithm, we will need to define an appropriate network structure, functions to add and remove nodes and edges, and a function to calculate the cheapest route given a tank capacity and a start and end city.

## **8.\*\*Evaluation and selection of the best solution:\*\***

To select the best solution, we can create a decision matrix and evaluate each solution based on criteria such as time efficiency, space efficiency, ease of implementation, and adaptability to different types of input data. We can then select the solution with the best overall score.

## **9.\*\*Preparation of Reports and Specifications:\*\***

Once we have selected the best solution, we should prepare a report detailing our choice and explaining how the solution works. We should also write specifications for the programme, including functional requirements and how data should be entered.

**10.\*\*Implementation and design:\*\***

Finally, we need to implement our solution in Java and test it to make sure it works correctly. We could design unit tests to verify each part of the program and system tests to verify that the whole program works correctly as a whole.

Sure, we will elaborate a decision matrix to evaluate each proposed solution based on criteria such as time efficiency, space efficiency, ease of implementation and adaptability to different types of input data. As mentioned, we will not consider the Bellman-Ford algorithm for the evaluation.

Scores shall be assigned from 1 to 5 for each criterion, with 5 being the best score and 1 the worst.

| ***Solution*** | ***Efficiency (time)*** | ***Efficiency (space)*** | ***Easy to implement*** | ***Adaptability*** |
| --- | --- | --- | --- | --- |
| *Dijkstra* | 3 | 4 | 4 | 3 |
| *Floyd-Warshall* | 2 | 3 | 3 | 5 |
| *BFS* | 4 | 5 | 5 | 2 |
| *DFS* | 4 | 5 | 5 | 2 |
| *PriorityQueue with Dijkstra* | 5 | 4 | 5 | 4 |

If we add up the scores, we get:

- Dijkstra's algorithm: 14

- Floyd-Warshall algorithm: 13

- Breadth first search strategy (BFS): 16

- Deep First Search Strategy (DFS): 16

- Using a priority queue with Dijkstra: 16

According to this assessment, the "Breadth First Search Strategy (BFS)", the "Depth First Search Strategy (DFS)" and the "Use of a Priority Queue with Dijkstra" seem to be the best solutions for this problem. In deciding between them, we might consider additional factors, such as the specific nature of the dataset or the preferences of the development team.

# TAD

## **\*\*Representation\*\***

We can represent this problem by means of a weighted directed graph. The nodes in the graph represent the cities, and the edges between the nodes represent the roads connecting the cities. Each edge has a weight, which corresponds to the cost of travelling along that road (the distance multiplied by the price of fuel in the starting city).

## **\*\*Invariant\*\***

For any pair of nodes u and v in the network, the weight of the cheapest route from u to v is always less than or equal to the sum of the weights of the cheapest routes from u to any intermediate node w and from w to v. In mathematical terms, if we denote the cost of the cheapest route between two nodes u and v as d(u, v), the invariant of the problem can be expressed as:

∀u, v, w ∈ V (d(u, v) ≤ d(u, w) + d(w, v))

## **\*\*Primitive operations\*\***

1. `agregarCiudad(int precioCombustible)`:

- {Pre: The price of fuel is a positive integer.}

- {Pos: The network contains an additional node representing the new city, and the fuel price in that city is set correctly}.

2. `agregarCarretera(int u, int v, int distancia)`:

-{Pre: The city indices u and v are non-negative integers representing existing cities, and the distance is a positive integer.}

-{Pos: The network contains an additional edge representing the new road, and the distance of that road is set correctly.}

3. `rutaMasBarata(int capacidad, int ciudadInicio, int ciudadFin)`:

- {Pre: The capacity is a positive integer, and the start and end city indices are non-negative integers representing existing cities.}

- {Pos: The cost of the cheapest route is returned, or a special value (such as -1) if no route is possible.}

## **\*\*Explanation of methods\*\***

`agregarCiudad(int fuelPrice)`:

"This method adds a city to the network".

{pre:int fuelPrice}

{post: The network contains an additional node representing the new city, and the fuel price in that city is set correctly}.

`agregarCarretera(int u, int v, int distancia)`:

"This method adds a road to the network".

{pre: int u, int v, int distancia}

{post: The network contains an additional edge representing the new road, and the distance of that road is set correctly }.

`rutaMasBarata(int capacidad, int ciudadInicio, int ciudadFin)`:

"This method finds the cheapest route between two cities for a car with a given fuel capacity".

{pre: int capacidad, int ciudadInicio y int ciudadFin}

{post: After the method is executed, the cost of the cheapest route is returned, or a special value if no route is possible.}