SPRINT 4

RELATORY
ESINF

1201518 | ANA ALBERGARIA

1201284 | DIOGO VIOLANTE

1200049 | JOÃO WOLFF

1201592 | MARTA RIBEIRO

G54

CLASS 2DF

PROFESSOR: FÁTIMA RODRIGUES

JANUARY 2022

Table of Contents

1. US401 As a Traffic manager I wish to know which ports are more critical (have greater centrality) in this freight network.

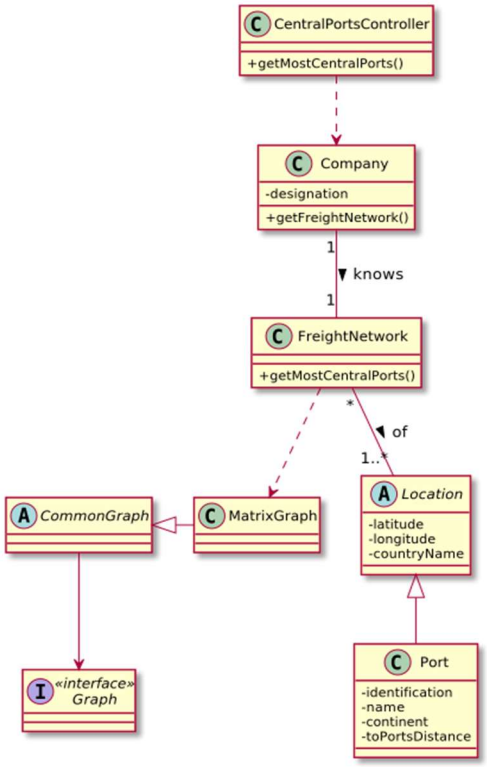
1.1 Problem and justification

US401 As a Traffic manager I wish to know which ports are more critical (have greater centrality) in this freight network.

- **Acceptance criteria:**
  - Return the n ports with greater centrality.
  - The centrality of a port is defined by the number of shortest paths that pass through it.
- **Contributes:**
- **Code and Tests:**
  - João Wolff, 1200049
- **complexity analysis:**
  - Ana Albergaria, 1201518

2.1 Class Diagram Analysis

For the us 401 the used classes were mostly the freight network class, which extends the base graph class and uses an MatrixGraph for representing the locations graph. This can be seen in the below class diagram:



2.3 Complexity Analysis

For solving the problem of returning the most critical ports of the graph, which are the ports with more shortest paths crossing through it, we used one main method called *List<Map.Entry<Location, Integer>> getMostCentralPorts().* Inside the function all the vertex of the network are iterated, and for each one it gets all of it's shortest paths using the Dijkstra's algorithm, after that it goes over the paths and adds or increase the appearence of the Ports in a Map. This can be seen in the following method:

```
public List<Map.Entry<Location, Integer>> getMostCentralPorts() {

    Map<Location, Integer> ports = new HashMap<>();
    ArrayList<LinkedList<Location>> paths = new ArrayList<>();
    ArrayList<Double> dists = new ArrayList<>();

    for (Location location : freightNetwork.vertices()) {
        Algorithms.shortestPaths(freightNetwork, location, Double::compare, Double::sum,  zero: 0.0, paths, dists);
        for (LinkedList<Location> path : paths) {
            for (Location loc : path) {
                if (loc instanceof Port) {
                    ports.merge(loc,  value: 1, Integer::sum);
                }
            }
        }
        paths.clear();
        dists.clear();
    }
    List<Map.Entry<Location, Integer>> toBeSortedMap = new ArrayList<>(ports.entrySet());
    toBeSortedMap.sort(Comparator.comparing(Map.Entry<Location, Integer>::getValue).reversed());
    return toBeSortedMap;
}
```

Analysing the complexity of the method we can conclude that it's mainly based on the O(n) of the Dijkstra's algorithm $O(v^2)$ times the number of vertices (v), leading us to the final complexity of $O(V^3)$.

2. US402 - Know the shortest path between two locals (city and/or port)

2.1 Problem and justification

US402 – know the shortest path between two locals (city and/or port).

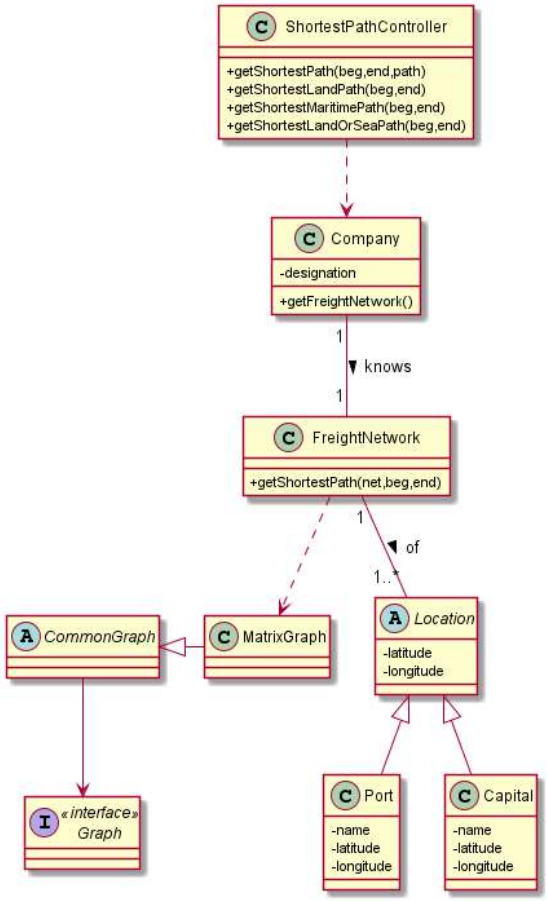
- **Contributes:**
  - **Code:**
    - Marta Ribeiro, 1201592
  - **Tests:**
    - Marta Ribeiro, 1201592
  - **Complexity analysis:**
    - Ana Albergaria, 1201518

To know the shortest path between two locals (city and/or port), the following acceptance criteria were given:

- Land path (only includes land routes, may start/end in port/city).
- Maritime path (only includes ports).
- Land or sea path (may include cities and ports).
- Obligatory passing through n indicated places.

2.2 Class Diagram Analysis

The discussed class organization and abstraction can be seen in the following class diagram:



2.3 Complexity Analysis

The shortest path is obtained by the principal method of the controller class:

```
public List<String> getShortestPath(Location beg, Location end, int path){
    //1: land path | 2: maritime path | 3: land or sea path
    if (path==1){
        return getShortestLandPath(beg,end);
    } else if (path==2){
        return getShortestMaritimePath(beg,end);
    } else if (path==3){
        return getShortestLandOrSeaPath(beg,end);
    }
    return null;
}
```

For each type of path, this method calculates the shortest path between the two locations, considering:

Land Path:

- **Vertices:** Ports and capitals
- **Edges:** Between capitals and between ports and capitals

Maritime Path:

- **Vertices:** Ports
- **Edges:** Only between ports

Land or Sea Path:

- **Vertices:** Ports and capitals
- **Edges:** Between ports, between capitals and between ports and capitals

As can be seen here, for the land path:

```
public List<String> getShortestLandPath(Location beg, Location end){
    FreightNetwork freightNetwork = this.company.getFreightNetwork();
    if (beg==end || beg==null || end==null){...}
    Graph<Location,Double> noPortsEdges = freightNetwork.getFreightNetwork().clone();
    Collection<Edge<Location,Double>> outgoingEdges;
    Collection<Edge<Location,Double>> incomingEdges;
    for (Location loc: noPortsEdges.vertices()) { //O(v)
        if (loc instanceof Port){ //O(1)
            incomingEdges=noPortsEdges.incomingEdges(loc);
            outgoingEdges=noPortsEdges.outgoingEdges(loc);
            for (Edge<Location,Double> edge:outgoingEdges) { //O(E)
                if (edge.getVOrig() instanceof Port && edge.getVDest() instanceof Port)  //O(1)
                    noPortsEdges.removeEdge(edge.getVOrig(),edge.getVDest()); //O(1)
            }
            for (Edge<Location,Double> edge:incomingEdges) { //O(E)
                if (edge.getVOrig() instanceof Port && edge.getVDest() instanceof Port)
                    noPortsEdges.removeEdge(edge.getVOrig(),edge.getVDest());
            }
        }
    }

    LinkedList<Location> result;
    List<String> strings = new ArrayList<>();
    String toAdd;
    result=freightNetwork.getShortestPath(noPortsEdges,beg,end); //Dijkstra
    if (!result.contains(beg) || !result.contains(end)){
        return null;
    }
    for (Location loc:result) {
        if (loc instanceof Port){
            toAdd="Port: " + ((Port) loc).getName();
            strings.add(toAdd);
        } else if (loc instanceof Capital){
            toAdd="Capital: " + ((Capital) loc).getName();
            strings.add(toAdd);
        }
    }
    return strings;
}
```

As the method, for each vertex of the graph – O(V) - , verifies if it's a Port, and if so, goes through its edges – O(E) - and removes all the edges between two Ports, it has complexity of **O(V x E)**.

After it calls the algorithm of Shortest Path by Dijkstra, which has a complexity of **$O(V^2)$**.

The final time complexity of this method is **$O(V^2)$**.

Here, for the maritime path:

```
public List<String> getShortestMaritimePath(Location beg, Location end){
    FreightNetwork freightNetwork = this.company.getFreightNetwork();
    if (beg==null || end==null || beg==end || beg instanceof Capital || end instanceof Capital){...}
    Graph<Location,Double> onlyPorts = freightNetwork.getFreightNetwork().clone();
    Collection<Edge<Location,Double>> outgoingEdges;
    Collection<Edge<Location,Double>> incomingEdges;
    for (Location loc: onlyPorts.vertices()) {
        if (loc instanceof Capital){
            incomingEdges=onlyPorts.incomingEdges(loc);
            outgoingEdges=onlyPorts.outgoingEdges(loc);
            for (Edge<Location,Double> edge:outgoingEdges) {
                onlyPorts.removeEdge(edge.getVOrig(),edge.getVDest());
            }
            for (Edge<Location,Double> edge:incomingEdges) {
                onlyPorts.removeEdge(edge.getVOrig(),edge.getVDest());
            }
            onlyPorts.removeVertex(loc);
        }
    }
    LinkedList<Location> result;
    List<String> strings = new ArrayList<>();
    result=freightNetwork.getShortestPath(onlyPorts,beg,end);
    if (!result.contains(beg) || !result.contains(end)){...}
    String toAdd;
    for (Location loc:result) {...}
    return strings;
}
```

The maritime path has the same logic as the land path:

As the method, for each vertex of the graph – O(V) - , verifies if it's a Capital, and if so, goes through its edges – O(E) - and removes all the edges containing Capitals and removes the Capitals from the Graph, it has complexity of **O(V x E)**.

After it calls the algorithm of Shortest Path by Dijkstra, which has a complexity of **$O(V^2)$**.

The final time complexity of this method is **$O(V^2)$**.

And here, for the land or sea path:

```
public List<String> getShortestLandOrSeaPath(Location beg, Location end){
    FreightNetwork freightNetwork = this.company.getFreightNetwork();
    LinkedList<Location> result;
    List<String> strings = new ArrayList<>();
    String toAdd;
    if (beg==end || beg==null || end==null){
        return null;
    }
    result=freightNetwork.getShortestPath(freightNetwork.getFreightNetwork(),beg,end); //O(V^2)
    if (!result.contains(beg) || !result.contains(end)){
        return null;
    }
    for (Location loc:result) {
        if (loc instanceof Port){
            toAdd="Port: " + ((Port) loc).getName();
            strings.add(toAdd);
        } else if (loc instanceof Capital){
            toAdd="Capital: " + ((Capital) loc).getName();
            strings.add(toAdd);
        }
    }
    return strings;
}
```

This method has similar logic to the previous ones, except it doesn't remove any edges of vertex from the Freight Network.

As it calls the Dijkstra's Shortest Path Algorithm, which has a complexity of $O(V^2)$ and then it iterates through all the vertex – O(V) – the final Time Complexity is **$O(V^2)$**.

All three functions make use of the *getShortestPath(Graph<Location,Double> g, Location origin, Location destination)* method of the *FreightNetwork* class:

```
public LinkedList<Location> getShortestPath(Graph<Location,Double> g, Location origin, Location destination) {
    LinkedList<Location> shortestPath = new LinkedList<>();
    Algorithms.shortestPath(g,origin,destination,Double::compare,Double::sum, zero: 0.0,shortestPath);
    return shortestPath;
}
```

As can be seen in the image above, the *shortestPath(Graph<V, E> g, V vOrig, V vDest, Comparator<E> ce, BinaryOperator<E> sum, E zero, LinkedList<V> shortPath)* method of the *Algorithms* class is called. This method was developed during the ESINF classes. It has a Time Complexity of $O(V^2)$.

```
public static <V, E> E shortestPath(Graph<V, E> g, V vOrig, V vDest,
                                    Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                    LinkedList<V> shortPath) {
    if(!g.validVertex(vOrig) || !g.validVertex(vDest)) {
        return null;
    }

    shortPath.clear();
    int numVerts = g.numVertices();
    boolean[] visited = new boolean[numVerts];
    @SuppressWarnings("unchecked")
    V[] pathkeys = (V[]) new Object[numVerts];
    @SuppressWarnings("unchecked")
    E[] dist = (E[]) new Object[numVerts];

    for (int i = 0; i < numVerts; i++) {
        dist[i] = null;
        pathkeys[i] = null;
    }

    shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathkeys, dist);

    E lengthPath = dist[g.key(vDest)];

    if(lengthPath == null)
        return null;

    getPath(g, vOrig, vDest, pathkeys, shortPath);
    return lengthPath;
}
```

3. US403 – Get most efficient circuit with less distance

3.1 Problem and justification

US403 - As a Traffic manager I wish to know the most efficient circuit that starts from a source location and visits the greatest number of other locations once, returning to the starting location and with the shortest total distance.

- **Acceptance criteria:**
    - Implement one of the heuristics used for this type of circuit.
- **Contributes:**
    - **Code, tests and complexity analysis:**
        - Diogo Violante, 1201284

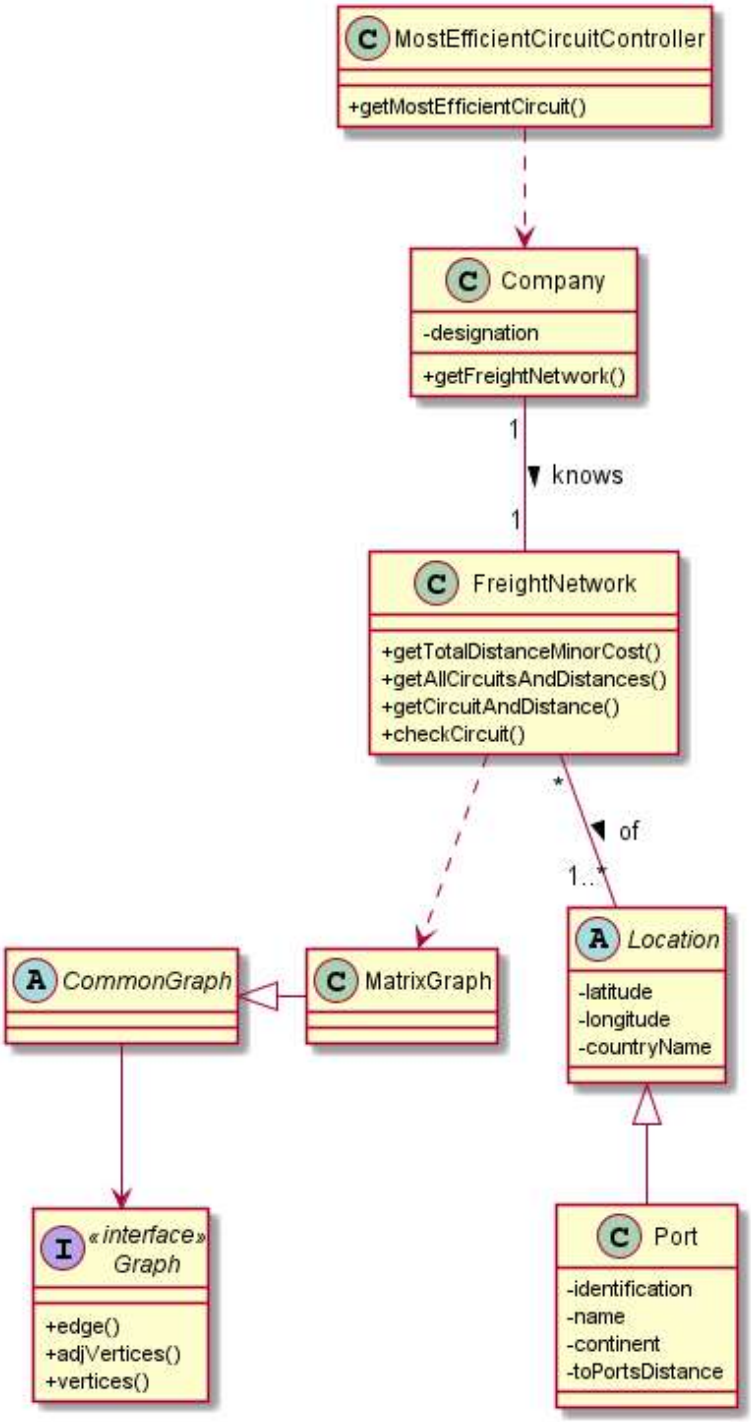
3.2 Class Diagram Analysis

The Class Diagram contains the relevant concepts from the Domain Model for the US403: Company,FreightNetwork, Location, Port and Capital.

Using good practices of OO software design, a Controller **Class was** created – MostEfficientCircuitController**.**

In order to create the get the most efficient circuit which visited the greatest number of locations once, it was needed to implement an algorithm in the class FreightNetwork, since this class extends the Graph class and uses MatrixGraph to represent the locations graph. The class Company is needed because it is the one that knows the class mentioned before.

In short, the Class Diagram was designed to ensure the solicited requisites.


MostEfficientCircuitController
+getMostEfficientCircuit()
Company
-designation
+getFreightNetwork()
1
knows
1
FreightNetwork
+getTotalDistanceMinorCost()
+getAllCircuitsAndDistances()
+getCircuitAndDistance()
+checkCircuit()
*
of
1..*
CommonGraph
MatrixGraph
Location
-latitude
-longitude
-countryName
«interface»
Graph
+edge()
+adjVertices()
+vertices()
Port
-identification
-name
-continent
-toPortsDistance

3.3 Complexity Analysis

3.3.1 Code and Type Of Algorithm

```
public Map<List<Location>, Double> getTotalDistanceMinorCost(){
    Map<List<Location>, Double> allCircuits = getAllCircuitsAndDistances();
    int size = 0;
    List<Location> biggestCircuit = new ArrayList<>();
    double distance = 0.00;

    for (List<Location> circuit : allCircuits.keySet()) {
        if (circuit.size()==size) {
            if (allCircuits.get(circuit)<distance) {
                size = circuit.size();
                biggestCircuit.clear();
                biggestCircuit.addAll(circuit);
                distance = allCircuits.get(circuit);
            }
        }
        if (circuit.size()>size) {
            size = circuit.size();
            biggestCircuit.clear();
            biggestCircuit.addAll(circuit);
            distance = allCircuits.get(circuit);
        }
    }
    biggestCircuit.add(biggestCircuit.get(0));

    Map<List<Location>, Double> result = new HashMap<>();

    result.put(biggestCircuit, distance);

    return result;
}
```

```
private Map<List<Location>, Double> getAllCircuitsAndDistances(){
    Map<List<Location>, Double> allCircuits = new HashMap<>();
    List<Location> tempLocations;
    List<Location> visited;
    List<Double> tempWeights;
    double initialDistance;


    for (Location local : freightNetwork.vertices()) {
        visited = new ArrayList<>();

        initialDistance = 0.00;
        do {
            tempLocations = new ArrayList<>();
            tempWeights = new ArrayList<>();
            getCircuitAndDistance(tempLocations, tempWeights, visited, initialDistance, local);
            initialDistance = 0.00;
            checkCircuit(local, tempLocations, visited, tempWeights, allCircuits);

        }while(tempLocations.size()!=1);
    }
    return allCircuits;
}
```

```java
private void getCircuitAndDistance(List<Location> circuit, List<Double> weights, List<Location> visited, Double initialDistance, Location local){
    if (circuit.contains(local)) {return;}
    circuit.add(local);
    int counter = 0;

    List<Location> adjs = new LinkedList<>(freightNetwork.adjVertices(local));
    int index = 0;

    for (Location location : adjs) {
        Double weight = freightNetwork.edge(local, location).getWeight();

        if ((initialDistance == 0.00 || weight <= initialDistance) && !visited.contains(location)){
            if (!circuit.contains(location)) {
                counter++;
            initialDistance = weight;
            index = adjs.indexOf(location);
            }
        }

    }
    if (counter == 0) return;
    weights.add(initialDistance);
    initialDistance = 0.00;
    getCircuitAndDistance(circuit, weights, visited, initialDistance, adjs.get(index));
}
```

```java
private void checkCircuit(Location local, List<Location> circuit, List<Location> visited, List<Double> weights, Map<List<Location>, Double> allCircuits) {
    if (circuit.size()==1){
        return;
    }

    Location lastLocation = circuit.get(circuit.size()-1);
    if(freightNetwork.edge(lastLocation, local)==null){
        circuit.remove(lastLocation);
        if (!visited.contains(lastLocation)) visited.add(lastLocation);
        return;
    }

    double initialDistance = 0.00;
    weights.add(freightNetwork.edge(lastLocation, local).getWeight());

    for (double values : weights) {
        initialDistance = initialDistance + values;
    }

    List<Location> temp = new ArrayList<>(circuit);
    allCircuits.put(temp, initialDistance);
    visited.add(lastLocation);
}
```

The method `checkCircuit()` verifies if, for a given circuit(List<Location> circuit), the last Location has an Edge with the first Location, in order to have a circuit. If not then the last Location is added to a List visited, so it isn´t visited again. Else the weight of that edge with the first one is added to a List<Double> weights and the total weight is calculated, the circuit is put on a Map<List<Location>, Double> allCircuits and the last location is added to the List visited.

This method has a Time Complexity of **O(N).**

`getCircuitAndDistance()` is a recursive method that given an initial Location, checks the adjacent Location with the least weight and proceeds to do the same with that Location, until it reaches a Location it already visited and ends. For each Location that is visited, it is added to the List circuit and its weight is added to the List weights.

This method has a Time Complexity of **O(VxE).**

`getAllCircuitsAndDistances()` checks if it exists a circuit for every Vertex in the Graph, by calling both of the methods mentioned above.

This method has a Time Complexity of **$O(V^2)$.**

`getTotalDistanceMinorCost()` is the method that calls the method above and iterates through all the circuits found and added to the map, to get the biggest circuit and if necessary the one with the least weight. It then return a map with the chosen circuit and the correspondent weight.

This method has a Time Complexity of **$O(N^2)$.**

So given all the methods used in this algorithm, we can conclude that the **Final Time Complexity** is **$O(V^2)$**, given the Time Complexity of `getAllCircuitsAndDistances()` when its called in `getTotalDistanceMinorCost()`**.**